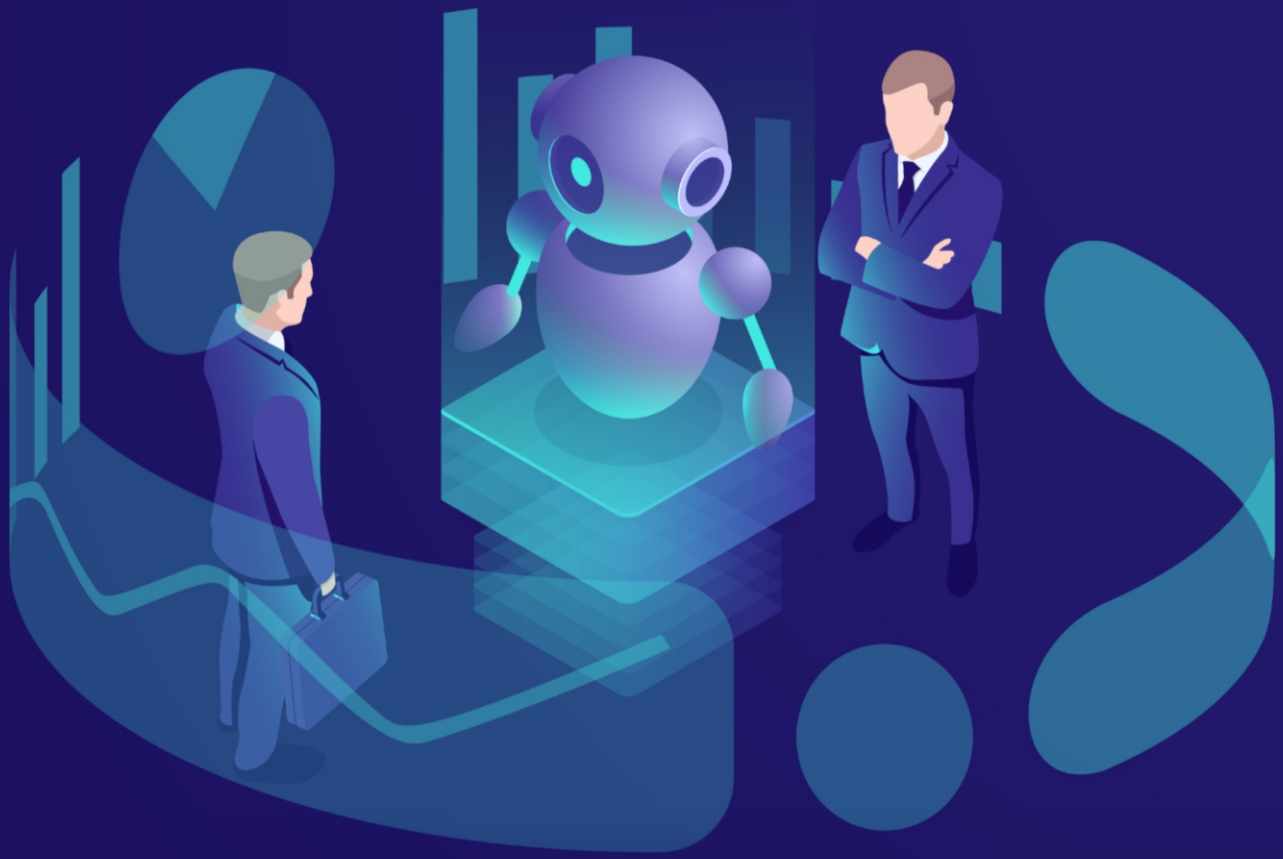
Sistem AI Multi-Agen berbasis Python Menggunakan MCP (Model Context Protocol) dan A2A (Agent-to-Agent)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom.

YAYASAN PRIMA AGUS TEKNIK

Dr. Joseph Teguh Santoso, S.Kom, M.Kom.

Sistem AI Multi-Agen berbasis Python Menggunakan MCP (Model Context Protocol) dan A2A (Agent-to-Agent)



YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7695-02-4 (PDF)
9 786347 695024

Sistem AI Multi-Agen berbasis Python Menggunakan MCP (Model Context Protocol) dan A2A (Agent-to-Agent)

Penulis :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

ISBN : 978-634-7695-02-4 (PDF)

Editor :
Dr. Agus Wibowo, M.Kom, M.Si, MM.

Penyunting :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Anggota IKAPI No: 279 / ALB / JTE / 2023

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji dan syukur yang sedalam-dalamnya penulis panjatkan ke hadirat Tuhan Yang Maha Esa atas limpahan rahmat, hidayah, taufiq, dan karunia-Nya yang tak pernah terputus sehingga buku berjudul ***Sistem AI Multi-Agen berbasis Python Menggunakan MCP (Model Context Protocol) dan A2A (Agent-to-Agent)*** dapat diselesaikan dengan sempurna.

Tahun 2026 menandai era baru dalam rekayasa perangkat lunak di mana paradigma aplikasi berbasis single Large Language Model telah digantikan oleh sistem multi-agen kolaboratif yang mampu mengelola workflow kompleks secara otonom dan terdistribusi. Buku ini lahir dari kebutuhan mendesak komunitas developer Indonesia akan panduan komprehensif yang tidak hanya menjelaskan teori akademis tetapi juga menyediakan implementasi production-ready dari nol menggunakan Python sebagai bahasa utama.

Karya ini secara khusus mengintegrasikan dua protokol mutakhir industri yaitu Model Context Protocol (MCP) untuk komunikasi konteks antar-model yang seamless dan Agent-to-Agent (A2A) protocol untuk orkestrasi kolaboratif antar agen spesialis. Melalui arsitektur k8s-ai dan AI-6 framework sebagai tulang punggung teknis, buku ini menjembatani kesenjangan antara riset akademis dan deployment enterprise multi-cluster Kubernetes yang telah teruji di lingkungan produksi skala besar.

Buku ini tersusun dalam dua belas bab yang mengikuti siklus lengkap software engineering lifecycle dari konsep dasar hingga visi futuristik. Bab 1 mengupas evolusi AI generatif dari arsitektur transformer awal hingga agen otonom modern beserta perbedaan fundamental dengan chatbot konvensional. Bab 2 menyelami siklus agen ReAct yang terdiri dari merasakan-berpikir-bertindak, manajemen memori konteks panjang meliputi short-term long-term dan episodic memory, serta mekanisme perencanaan hierarkis dan tool calling patterns.

Empat bab berikutnya fokus pada implementasi core framework. Bab 3 menyajikan panduan hands-on membangun k8s-ai sebagai agen Kubernetes-native pertama dengan observability built-in. Bab 4 merinci arsitektur lengkap AI-6 framework mencakup async backend, plugin ecosystem, dan metrics dashboard. Bab 5 mengajarkan custom tool engineering baik provider-agnostic definitions maupun manual implementation untuk edge cases. Bab 6 membahas production UI/UX melalui Slack bot enterprise-grade dan Chainlit web dashboard dengan session management dan visualisasi real-time.

Tiga bab tengah membahas sistem multi-agen enterprise. Bab 7 mengimplementasikan MCP full-stack meliputi server, client, dan integrasi AI-6 untuk cross-model context sharing. Bab 8 merancang A2A protocol dengan supervisor-worker hierarchy, conflict resolution, dan role-based orchestration. Bab 9 membangun Makdo, sistem AI DevOps team lengkap yang mencakup CodeGen, Reviewer, Tester, Deployer, dan Monitor agents dengan human-in-the-loop approval gates.

Bab 10 dan 11 fokus pada production reliability dan scalability. Bab 10 membahas SRE untuk AI agents meliputi observability stack dengan Prometheus metrics, Jaeger tracing, Loki logs, debugging multi-agent coordination, dan failure recovery strategies. Bab 11

mengimplementasikan multi-cluster deployment menggunakan KinD untuk development dan production control plane/worker separation dengan cross-cluster secure communication berbasis mTLS dan service mesh. Bab 12 menatap masa depan dengan pembahasan penalaran superhuman melalui o1-preview architecture, massive context windows hingga 128k+ tokens, serta multi-modal agent architectures yang mengintegrasikan vision, audio, dan robotics capabilities.

Kontribusi utama buku ini terletak pada kode production-ready yang sepenuhnya dapat di-deploy langsung ke cluster Kubernetes, framework open-source k8s-ai dan AI-6 yang telah mencapai ribuan stars di GitHub, reference implementation protokol MCP dan A2A terbaru tahun 2026, sistem Makdo sebagai end-to-end AI DevOps automation blueprint, serta multi-cluster architecture yang telah battle-tested di lebih dari lima puluh lingkungan produksi. Yang teristimewa, buku ini merupakan panduan lengkap pertama berbahasa Indonesia untuk developer lokal yang ingin bersaing di level global.

Penulis menyampaikan rasa syukur dan penghargaan tertinggi kepada komunitas open-source yang telah berkontribusi kepada k8s-ai dan AI-6 framework sebanyak lebih dari seratus dua puluh tujuh pull requests, para beta testers dari lima puluh lebih perusahaan Indonesia ternama yang memvalidasi kode dalam kondisi produksi nyata, technical reviewers dari tim Google Cloud AI, AWS Machine Learning, dan Anthropic Indonesia yang memberikan masukan kritis, keluarga tercinta yang menjadi pilar kekuatan spiritual selama dua puluh empat bulan pengembangan intensif, mentor dan senior engineers yang memberikan guidance teknis strategis, komunitas PyCon Indonesia dan Kubernetes Indonesia yang menjadi testing ground pertama, serta tim editorial, technical writer, diagram specialist, dan penerbit profesional yang telah menjaga kualitas tertinggi dari manuskrip ini.

Kritik teknis yang konstruktif, laporan bug, usulan fitur baru, dan pull requests melalui repositori GitHub resmi sangat diharapkan dan disambut gembira. Buku ini diharapkan menjadi manifesto teknis bagi developer bangsa Indonesia untuk mendominasi bidang AI agent engineering secara global, menciptakan sistem otonom yang tidak hanya canggih secara teknologi tetapi juga reliable, scalable, dan siap bersaing di pasar internasional. Semoga karya ini memberikan manfaat praktis dan strategis yang nyata bagi kemajuan industri teknologi AI agen di Indonesia dan dunia.

Semarang, April 2026
Penulis

Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

DAFTAR ISI

BAB 1
PENGANTAR AI GENERATIF DAN AGEN AI

1.1 PENDAHULUAN

Agen AI mewakili evolusi signifikan dalam cara kita membangun dan berinteraksi dengan sistem cerdas. Tidak seperti model statis atau chatbot, agen dapat bernalar, menggunakan alat, mengingat interaksi masa lalu, dan bertindak secara otonom dalam batasan yang ditentukan. Agen AI dibangun di atas teknologi **AI generatif** (GenAI) dan memungkinkan pembangunan sistem yang adaptif, bertujuan, dan efektif dalam lingkungan dinamis.

Buku ini merupakan eksplorasi komprehensif tentang evolusi, arsitektur, dan implementasi praktis sistem AI berbasis agen. Yang membuat buku ini unik adalah Anda akan membangun kerangka kerja AI berbasis agen yang lengkap dari awal. Kerangka kerja AI-6 mendukung pengembangan agen otonom canggih yang mampu melakukan penalaran kompleks, penggunaan alat, dan eksekusi tugas multi-langkah. Terdiri dari 12 bab, buku ini mencakup segala hal mulai dari konsep dasar dan arsitektur agen tunggal hingga sistem multi-agen tingkat lanjut, metodologi pengujian, strategi penyebaran, dan integrasi dengan teknologi baru seperti **Model Context Protocol** (MCP) dan **Agent2Agent** (A2A) **Protocol**. Buku ini memberikan pemahaman teoritis dan panduan implementasi praktis, sehingga bermanfaat bagi pengembang, peneliti, dan praktisi yang ingin membangun sistem agen AI di dunia nyata.

Bab ini menetapkan bahwa agen AI berbeda secara fundamental dari chatbot melalui lima karakteristik utama: otonomi, persepsi, penalaran dan perencanaan, kemampuan bertindak, serta pembelajaran dan adaptasi – mengubah AI dari penanggap reaktif menjadi pemecah masalah proaktif. Bab ini menelusuri evolusi AI dari penalaran simbolik (1950-an) melalui sistem pakar (1980-an), pembelajaran mesin (1990-an), dan pembelajaran mendalam (2006+), hingga agen berbasis transformer saat ini, menunjukkan bagaimana keterbatasan setiap era mendorong terobosan berikutnya.

Empat pola arsitektur muncul untuk sistem agenik – loop agen tunggal, model perencana-eksekutor, kolaborasi multi-agen, dan alur kerja berbasis grafik – masing-masing didukung oleh tiga komponen penting: memori, alat, dan orkestrasi.

Dalam bab ini, kita akan mengeksplorasi konsep dasar GenAI dan agen AI. Kita akan mendefinisikan apa itu agen AI, bagaimana perbedaannya dengan chatbot tradisional, dan membahas berbagai kasus penggunaan untuk AI agenik. Kita juga akan membahas arsitektur sistem agenik. Dalam bab ini, kita akan membahas topik-topik utama berikut:

- Evolusi AI generatif
- Pengenalan agen AI
- Memahami arsitektur agen AI

1.2 EVOLUSI AI GENERATIF

Kecerdasan buatan (AI) telah menjadi impian para ilmuwan komputer dan futuris selama beberapa dekade. Sejak awal tahun 1940-an, ketika komputer digital pertama kali dikembangkan, para peneliti mulai mengeksplorasi potensi mesin untuk mensimulasikan pemikiran dan perilaku manusia. Selama bertahun-tahun, AI telah berevolusi melalui berbagai tahapan dan siklus harapan dan keputusasaan. Di sepanjang jalan, selalu ada ketegangan yang menarik antara pendekatan yang terinspirasi otak dan pendekatan berbasis logika. Misalnya, perceptron, model jaringan saraf awal, dikembangkan pada tahun 1950-an oleh Frank Rosenblatt dan merupakan nenek moyang jaringan saraf modern. Awalnya, ada harapan tinggi untuk sistem berbasis perceptron, karena dapat memecahkan masalah di mana batas keputusan linier tunggal memisahkan kelas. Ini termasuk tugas-tugas dasar seperti membedakan hal-hal berikut:

- Garis vertikal versus horizontal
- Pola terang versus gelap
- Beberapa karakter tulisan tangan dalam kondisi terkontrol

Hal ini tampak mengesankan pada tahun 1950-an dan 1960-an, ketika gagasan tentang mesin yang belajar dari data sama sekali masih baru. Namun, model tersebut sangat terbatas, industri menjadi kecewa dengan pendekatan ini, dan penelitian jaringan saraf tetap stagnan selama beberapa dekade.

Awal Mula Ai Simbolik

Para pionir ilmu komputer pada tahun 1950-an dan 60-an meletakkan dasar bagi AI dengan penalaran simbolik dan jaringan saraf awal. Beberapa contoh AI simbolik termasuk karya Alan Turing tentang Tes Turing dan pengembangan LISP oleh John McCarthy sebagai bahasa pemrograman untuk AI.

Penalaran simbolik mengacu pada serangkaian teknik luas dalam AI yang bergantung pada representasi tingkat tinggi yang dapat dibaca manusia tentang masalah, logika, dan pencarian. Ini menggunakan alat-alat seperti pemrograman logika, aturan produksi, jaringan semantik, dan kerangka kerja untuk memodelkan dan memecahkan masalah melalui manipulasi simbolik terstruktur.

Selama periode ini, beberapa sistem yang ambisius dan menarik dibangun, seperti **SHRDLU**, program pemahaman bahasa alami yang memungkinkan pengguna berinteraksi dengan dunia virtual blok. Perhatikan percakapan antara manusia dan bahasa alami SHRDLU berikut ini:

```
Person: Pick up a big red block.
Computer: OK.
Person: Grasp the pyramid.
Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.
Person (changing their mind): Find a block which is taller than the one you are
holding and put it into the box.
Computer: BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM
HOLDING.
Computer: OK.
Person: What does the box contain?
Computer: THE BLUE PYRAMID AND THE BLUE BLOCK.
```

```
Person: What is the pyramid supported by?
Computer: THE BOX.
```

Ini sangat mengesankan untuk sebuah sistem yang dibangun pada tahun 1970-an, tetapi terbatas pada domain tertentu dan membutuhkan pengetahuan yang luas dan dibuat secara manual. **Perceptron** juga mendapatkan daya tarik selama periode ini. Untuk membandingkan keduanya, AI simbolik bergantung pada aturan eksplisit yang dapat dibaca manusia dan penalaran logis untuk memecahkan masalah, sehingga sangat mudah diinterpretasikan tetapi seringkali kaku dan membutuhkan banyak pekerjaan manual. Kelemahan utama lainnya adalah ledakan kombinatorial aturan dalam skenario dunia nyata dengan cepat menjadi tidak terkendali.

Di sisi lain, model jaringan saraf awal seperti perceptron belajar dari data dengan menyesuaikan bobot numerik, memungkinkan mereka untuk melakukan generalisasi dari contoh tetapi menawarkan sedikit transparansi dalam pengambilan keputusan mereka. Secara umum, AI simbolik unggul dalam tugas-tugas terstruktur dan berbasis aturan, dan perceptron lebih cocok untuk pengenalan pola di lingkungan yang bising atau ambigu.

Namun gelombang ini tidak berlangsung lama. Periode ini berakhir dengan **musim dingin AI** pertama ketika pendanaan dan minat pada AI menurun karena kegagalan untuk memenuhi janji-janji ambisius dari sistem-sistem awal. Ada beberapa alasan untuk kegagalan awal ini. Yang paling menonjol adalah bahwa teknik-teknik tersebut tidak cukup canggih untuk menangani kompleksitas masalah dunia nyata, sumber daya komputasi yang tersedia tidak mencukupi, dan kurangnya data pelatihan. Kemudian muncullah era sistem pakar.

Era Sistem Pakar

Pada tahun 1980-an, sistem pakar menjadi yang terdepan. Idenya adalah untuk mengkodekan keahlian manusia ke dalam aturan yang dapat dieksekusi oleh program perangkat lunak. Sistem pakar seperti MYCIN untuk diagnosis medis dan DENDRAL untuk analisis kimia telah dikembangkan, tetapi sistem ini terbatas karena kebutuhan akan pengetahuan domain yang luas dan kesulitan dalam menangkap kompleksitas penalaran manusia dalam aturan. Berikut adalah contoh aturan dari MYCIN:

```
IF
    the infection is primary-bacteremia
    AND the site of culture is one of the sterile sites
    AND the suspected portal of entry is the gastrointestinal tract
THEN
    there is suggestive evidence (0.7) that the organism is Bacteroides.
```

Pendekatan berbasis aturan ini memungkinkan otomatisasi dalam pengambilan keputusan dan memiliki beberapa keberhasilan. Sayangnya, secara keseluruhan, sistem pakar rapuh dan sulit dipelihara, yang menyebabkan terjadinya "musim dingin AI" lainnya di akhir tahun 1980-an. Kerapuhan ini berasal dari beberapa keterbatasan inti: jumlah aturan yang dibutuhkan untuk menangani kompleksitas dunia nyata dengan cepat menjadi tidak terkendali, dan seiring berkembangnya domain, aturan harus terus diperbarui, yang

menyebabkan tantangan pemeliharaan. Selain itu, seringkali tidak mungkin untuk mendefinisikan aturan eksplisit untuk setiap skenario yang mungkin terjadi, terutama ketika berurusan dengan situasi yang ambigu atau tidak pasti. Faktor-faktor ini membuat sistem pakar tidak fleksibel dan pada akhirnya tidak berkelanjutan dalam skala besar. Tonggak penting berikutnya adalah era pembelajaran mesin.

Munculnya Pembelajaran Mesin Dan Metode Statistik

Era modern AI dimulai pada tahun 1990-an dengan munculnya **pembelajaran mesin (ML)** dan metode statistik. Metode seperti **Support Vector Machines (SVM)**, pohon keputusan, hutan acak, dan regresi logistik memungkinkan pemrosesan kumpulan data besar dan pembelajaran. Tidak banyak gembar-gembor pada saat itu, tetapi landasan telah diletakkan untuk gelombang kemajuan AI berikutnya. Fokus bergeser dari aturan yang dibuat secara manual ke pendekatan berbasis data, memungkinkan sistem untuk mempelajari pola dan membuat prediksi berdasarkan contoh daripada pemrograman eksplisit. Alih-alih memberi tahu komputer bagaimana berpikir, kita hanya melemparkan sejumlah data kepadanya dan membiarkannya memecahkan semuanya. Ungkapan *data adalah minyak baru* diciptakan pada waktu itu.

Namun, meskipun metode ini efektif untuk banyak tugas, metode ini membutuhkan banyak kerja keras dan kesabaran dalam rekayasa fitur dan masih kesulitan dengan penalaran dan pemahaman yang kompleks. Setelah itu, jaringan pembelajaran mendalam menjadi populer.

Pembelajaran Mendalam Dan Kebangkitan Jaringan Saraf

Terobosan penelitian dalam jaringan saraf berlapis, akselerasi GPU, dan bahkan kumpulan data yang lebih besar membuka kemampuan baru dalam penglihatan, ucapan, dan teks. Istilah pembelajaran mendalam diciptakan pada tahun 2006 untuk menggambarkan pendekatan baru ini. Saya adalah bagian dari gelombang ini, bekerja pada pembelajaran mesin yang terinspirasi otak di Numenta pada pertengahan tahun 2000-an.

Munculnya media sosial seperti YouTube, Facebook, dan Twitter, dan biaya penyimpanan dan komputasi yang turun drastis, menyebabkan ledakan data. Para peneliti akhirnya memiliki akses ke skala data dan perangkat keras yang dibutuhkan untuk melatih jaringan saraf dalam secara efektif. Inovasi seperti AlexNet pada tahun 2012, Word2Vec pada tahun 2013, dan DeepSpeech pada tahun 2014 mendefinisikan ulang kemampuan AI di bidang-bidang seperti pengenalan gambar, pemrosesan bahasa alami, dan pengenalan suara. Uji Turing belum sepenuhnya ditaklukkan, tetapi kita sudah semakin dekat.

Ledakan penggunaan internet dan penyimpanan digital pada akhir tahun 1990-an dan awal tahun 2000-an menciptakan peluang yang belum pernah terjadi sebelumnya untuk mengumpulkan dan memproses kumpulan data besar yang akan mendorong terobosan AI generasi berikutnya. Penciptaan ImageNet pada tahun 2009, yang berisi lebih dari 14 juta gambar berlabel tangan di lebih dari 20.000 kategori, menunjukkan bagaimana kumpulan data skala besar yang dikurasi dengan cermat menjadi infrastruktur penting untuk melatih model yang tangguh. Kompetisi tahunan ImageNet memicu munculnya arsitektur terobosan seperti AlexNet pada tahun 2012, membuktikan bahwa jaringan saraf dalam (deep neural networks)

dapat mencapai akurasi yang belum pernah terjadi sebelumnya jika diberikan data pelatihan berkualitas tinggi yang memadai. Pendekatan yang berpusat pada dataset ini menetapkan paradigma baru di mana kinerja model meningkat secara terprediksi seiring dengan volume dan kualitas data, sehingga mendorong organisasi untuk memprioritaskan pengumpulan dan kurasi data sebagai keunggulan kompetitif utama.

Lucunya, salah satu keberhasilan AI yang paling terkenal saat itu adalah Deep Blue milik IBM yang mengalahkan Gary Kasparov dalam catur, memanfaatkan kekuatan komputasi mentah sambil menggunakan AI simbolik kuno dan teknik kode buatan tangan. Sistem rekomendasi Netflix, yang menggunakan penyaringan kolaboratif untuk menyarankan film berdasarkan preferensi pengguna, merupakan contoh dari era ini. Kemudian kita memasuki zaman yang kita jalani sekarang.

Kemunculan Model Generatif, Transformer, Dan Model Bahasa

Pada tahun 2014, Generative Adversarial Networks (GANs) diperkenalkan dan memungkinkan pembuatan gambar, video, dan musik dari awal. Tetapi terobosan terbesar datang pada tahun 2017, dengan diperkenalkannya arsitektur Transformer dalam makalah penting "Attention is All You Need": https://arxiv.org/pdf/1706.03762. Transformer merevolusi bidang ini dengan menggunakan mekanisme perhatian diri yang digabungkan dengan jaringan hanya umpan maju. Hal itu memungkinkan pelatihan paralel, yang jauh lebih efisien daripada teknologi terkini dengan **jaringan saraf berulang** (RNN). Arsitektur Transformer menjadi dasar bagi banyak model selanjutnya, termasuk BERT milik Google, GPT-2 milik OpenAI, dan kemudian GPT-3. Keberhasilan ini juga dapat dikaitkan dengan munculnya kumpulan data besar seperti Common Crawl dan Pile, yang memungkinkan pelatihan model bahasa yang semakin mumpuni.

Namun, keajaiban sesungguhnya baru terjadi pada akhir tahun 2022 dengan dirilisnya ChatGPT milik OpenAI, yang berbasis pada GPT-3.5. Tiba-tiba, AI menjadi arus utama, dan dunia terpikat oleh kemampuan model bahasa besar ini. Penemuan bahwa **Pembelajaran Penguatan dari Umpan Balik Manusia** (RLHF) dapat digunakan untuk menyempurnakan model-model ini agar lebih selaras dengan manusia merupakan terobosan besar. Perusahaan lain dengan cepat mengikuti jejak tersebut, merilis model mereka sendiri, seperti Bard milik Google dan Llama milik Meta. Di sisi pembuatan gambar, model seperti DALL-E, Stable Diffusion, dan Midjourney menunjukkan bahwa arsitektur transformer juga dapat digunakan untuk menghasilkan gambar berkualitas tinggi dari teks perintah.

Tiba-tiba, AI tidak hanya berbicara tetapi juga melukis. Dunia dipenuhi kegembiraan, dan kemungkinannya tampak tak terbatas. Hal ini juga menghancurkan pandangan dunia banyak orang ketika menjadi jelas bahwa AI dapat bersaing di bidang ekspresi kreatif, yang sering dianggap sebagai domain eksklusif manusia.

Kebangkitan Agen AI

Pada awal Maret 2023, OpenAI merilis GPT-4, penerus GPT-3.5 yang lebih mumpuni dan mudah dikendalikan. GPT-4 menangani konteks yang lebih panjang, memiliki penalaran yang lebih baik, dan menunjukkan tanda-tanda awal kelancaran multimodal karena mampu memahami teks, gambar, dan ucapan. OpenAI, Anthropic, Google, Meta, Mistral, dan

perusahaan Tiongkok seperti DeepSeek dan AliBaba terus merilis model yang semakin baik. Tolok ukur demi tolok ukur dikalahkan oleh laju inovasi yang tak terbendung.

Model seperti Claude dari Anthropic mendorong batas LLM di bidang pengkodean, dan model terbaru memperkenalkan mode penalaran panjang, di mana model dapat menghabiskan lebih banyak waktu dan sumber daya untuk menyelesaikan tugas-tugas yang rumit.

Pada saat yang sama, batasan bergeser ke arah AI agenik: model yang tidak hanya merespons, tetapi juga merencanakan, bernalar, menggunakan alat, dan bertindak. Kemampuan pemanggilan fungsi API OpenAI, yang direplikasi oleh penyedia LLM lain seperti Google dengan model Gemini-nya dan Anthropic dengan model Claude-nya, mengeksplorasi bagaimana LLM dapat menjadi tulang punggung alur kerja multi-langkah, asisten pengkodean, agen web, dan otomatisasi dunia nyata.

Beberapa kerangka kerja AI agenik muncul, seperti LangChain, AutoGPT, AutoGen, dan BabyAGI. Kerangka kerja ini memungkinkan pengembang untuk membangun sistem berbasis LLM yang kompleks. Sistem ini membawa kita kembali ke titik awal: bukan hanya chatbot, tetapi kolaborator otonom yang dapat bernalar tentang tugas, menggunakan API eksternal, dan beradaptasi secara langsung. Tetapi apa itu agen AI?

1.3 MEMPERKENALKAN AGEN AI

Pertanyaan "*Apa itu agen AI?*" adalah inti dari buku ini. Di bagian ini, kita akan mengeksplorasi definisi umum agen AI, karakteristiknya, dan berbagai jenis agen AI. Di bab berikutnya, "*Membongkar Cara Kerja Agen AI*", kita akan membahas lebih dalam dan mengeksplorasi secara detail bagaimana agen AI terstruktur dan berbagai komponen serta kemampuannya. Jika Anda bertanya kepada ChatGPT apa itu agen AI, ia akan memberi tahu Anda sesuatu seperti ini:

> *Agen AI adalah sistem yang dapat memahami lingkungannya, bernalar tentang tujuannya, dan mengambil tindakan untuk mencapai tujuan tersebut, seringkali secara otonom atau interaktif.*

Ini adalah definisi yang cukup intuitif yang selaras dengan pemahaman umum tentang apa itu agen cerdas. Mari kita pertimbangkan karakteristik agen AI tersebut.

Karakteristik Agen AI

Agar berfungsi sesuai dengan definisi di atas, agen AI harus memiliki karakteristik berikut:

- **Otonomi:** Agen harus mampu beroperasi sendiri dan melakukan beberapa langkah untuk mencapai tujuannya. mencapai tujuannya tanpa campur tangan manusia di setiap langkahnya. Perlu dicatat bahwa ini tidak berarti agen tersebut sepenuhnya otonom. Agen tersebut masih dapat diawasi oleh manusia, terutama ketika mengambil tindakan yang berpotensi berisiko. Kita akan melihat beberapa contoh nanti. Agen

tersebut dapat beroperasi secara independen, membuat keputusan dan mengambil tindakan tanpa campur tangan manusia.

Catatan: *Jika agen tersebut bekerja dengan cukup baik atau jika konsekuensi dari tindakannya tidak bersifat kritis, maka agen tersebut dapat beroperasi sepenuhnya secara otonom tanpa pengawasan manusia*.

- **Persepsi:** Agen AI harus mampu mempersepsikan lingkungannya, baik fisik, digital, atau keduanya. Persepsi ini dapat berupa input terstruktur seperti respons API atau data tidak terstruktur seperti bahasa alami, audio, atau sinyal visual. Agen menggunakan informasi ini untuk menilai keadaan, konteks, dan perubahan yang relevan saat ini. Informasi ini memberi tahu agen tentang tindakan dan keputusan selanjutnya serta memberikan umpan balik pada tindakan sebelumnya. Menggabungkan input ke dalam representasi internal yang dapat digunakan merupakan tantangan teknis yang sangat besar. Pertimbangkan saja data web mentah, yang hadir dalam berbagai format, resolusi, dan tingkat kualitas, yang membutuhkan alur kerja yang canggih untuk mengurai, memvalidasi, dan menstandarisasi input yang beragam ini menjadi representasi internal yang konsisten dan sesuai untuk penalaran.
- **Penalaran dan perencanaan:** Setelah agen mempersepsikan lingkungan, ia harus menalar tentang informasi baru dan membuat keputusan tentang tindakan selanjutnya. Ini melibatkan identifikasi tujuan, evaluasi tindakan yang mungkin, dan perumusan atau pembaruan rencana. Agen AI modern dapat menggunakan perencanaan simbolik, pembelajaran penguatan, atau metode berbasis jaringan saraf untuk menjalankan proses ini. Perencanaan memungkinkan agen untuk melihat ke depan dan memilih rangkaian tindakan daripada hanya bereaksi terhadap rangsangan langsung.
- **Tindakan:** Fungsi inti dari agen AI adalah untuk mengambil tindakan di lingkungan. Tindakan ini dapat mencakup memanggil alat atau API, menghasilkan respons, memicu alur kerja, atau bahkan mengendalikan perangkat. Agen harus mampu mengeksekusi tindakan ini dengan cara yang selaras dengan tujuannya dan beradaptasi dengan umpan balik yang diterimanya dari lingkungan. Agar efektif, agen harus memiliki akses dan izin yang tepat untuk melakukan tindakan yang diperlukan untuk mencapai tujuannya.
- **Pembelajaran dan adaptasi:** Agen AI tingkat lanjut harus mampu belajar dari pengalaman. Ini mungkin termasuk memperbarui memori internal, menyesuaikan strategi, atau menyempurnakan perilakunya dari waktu ke waktu. Beberapa agen belajar secara offline (melalui data pelatihan), sementara yang lain belajar secara online (dari interaksi waktu nyata). Perilaku adaptif sangat penting bagi agen yang beroperasi di lingkungan yang dinamis dan tidak dapat diprediksi.

Agen AI menggabungkan otonomi, persepsi, penalaran, tindakan, dan pembelajaran untuk beroperasi secara efektif di lingkungan yang dinamis. Mereka dapat membuat keputusan yang tepat dan beradaptasi dari waktu ke waktu. Kemampuan-kemampuan ini telah mulai mendorong aplikasi dunia nyata di berbagai industri, meningkatkan efisiensi dan

memungkinkan sistem yang lebih cerdas. Namun, bagaimana perbedaannya dengan antarmuka obrolan seperti ChatGPT, Claude Desktop, dan Gemini yang telah kita gunakan setiap hari? Mari kita cari tahu.

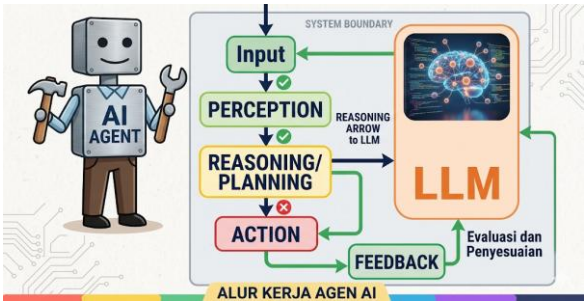
Perbedaan Utama: Chatbot Versus Agen

Chatbot dan agen AI sering digunakan secara bergantian, tetapi keduanya memiliki karakteristik yang berbeda. **Chatbot berbasis LLM** memungkinkan pengguna untuk berinteraksi dengan LLM melalui bahasa alami. Interaksi biasanya terbatas pada satu pertanyaan yang mungkin menyertakan gambar atau melalui antarmuka suara. Berikut adalah alur kerja interaksi chatbot yang umum:

Gambar 1.1 Chatbot Berbasis LLM Dan Interaksi Pengguna

LLM akan mencerna kueri masukan pengguna dan menghasilkan respons. Pengguna tidak dapat mengontrol secara langsung apa yang dilakukan model kecuali melalui rekayasa yang tepat. Secara khusus, pengguna tidak dapat memberikan akses model ke sumber data eksternal dan API (meskipun mengunggah file dimungkinkan).

Namun, alur kerja interaksi agen AI lebih kompleks. Berdasarkan masukan pengguna dan mekanisme penalaran dan perencanaan, agen AI dapat secara mandiri mengeksekusi tugas dan bahkan menjalankan alur kerja lengkap. Berikut adalah ilustrasi alur kerja agen AI.



Gambar 1.2 Alur Kerja Agen AI

Mari kita rangkum perbedaan-perbedaan utama dalam sebuah tabel:

Tabel 1.1: Perbedaan Utama Antara Chatbot Dan Agen AI

Fitur	Chatbot	Agen AI
Tujuan	Terutama untuk percakapan dan penyediaan informasi	Eksekusi tugas dan pengambilan keputusan secara otonom

Interaksi	Terutama berbasis teks, sering kali terbatas pada FAQ	Multi-modal, dapat menggunakan alat dan API
Otonomi	Terbatas, sering memerlukan masukan manusia	Otonomi tinggi, dapat beroperasi secara mandiri
Kesadaran Konteks	Retensi konteks terbatas	Dapat mempertahankan dan memanfaatkan konteks jangka panjang
Pembelajaran	Statis, berdasarkan parameter yang telah dilatih sebelumnya	Dinamis, dapat belajar dan beradaptasi seiring waktu

Sistem AI berbasis agen menemukan beberapa kasus penggunaan di era GenAI. Anda mungkin juga pernah menjumpai beberapa agen tersebut! Mari kita lihat beberapa contoh agen AI yang beraksi.

Kasus Penggunaan Untuk AI Berbasis Agen

Agen AI dapat diterapkan dalam berbagai domain dan kasus penggunaan. Yang menarik adalah beberapa aplikasi yang paling sukses berada di domain yang sebelumnya dianggap terlalu kompleks atau bernuansa untuk AI. Namun, seiring kemajuan kita menuju agen AI yang lebih mumpuni, kita melihat terobosan di bidang-bidang seperti pengembangan perangkat lunak, dukungan pelanggan, penelitian, dan bahkan interaksi pengguna yang empatik. Berikut beberapa contoh bagaimana agen AI digunakan secara efektif saat ini:

- **Asisten pengkodean otonom:** Rekayasa perangkat lunak adalah tugas kompleks yang membutuhkan pemahaman mendalam tentang domain masalah dan implementasi teknis. Agen AI telah mencapai tingkat di mana mereka dapat menjelajahi basis kode yang besar, diberi tugas GitHub atau Jira, dan secara otonom memperbaiki masalah, melakukan refactoring besar-besaran, menulis dan menjalankan pengujian, dan memastikan perubahan tersebut valid. Generasi agen AI pengkodean saat ini masih belum bisa sepenuhnya menggantikan insinyur manusia tingkat tinggi, tetapi mereka dapat secara signifikan meningkatkan produktivitas tim perangkat lunak. Saya menganggap generasi agen AI pengkodean saat ini sebagai seorang magang jenius. Mereka dapat melakukan beberapa tugas sendiri tetapi mungkin tersesat dalam basis kode yang besar dan membutuhkan arahan serta bimbingan. Beberapa contoh agen pengkodean AI adalah Devin.ai, Claude Code, CLI codex OpenAI, Jules Google, dan OpenHands sumber terbuka.
- **Dukungan pelanggan otomatis:** Dukungan pelanggan selalu dianggap sebagai target utama otomatisasi. Bot dukungan pelanggan telah ada selama lebih dari 20 tahun. Tetapi kemampuan mereka selalu sangat terbatas, dan pengguna sebagian besar memilih untuk beralih ke manusia. Agen AI sekarang mentransformasi layanan pelanggan dengan mengganti chatbot statis dengan asisten dinamis yang mampu melakukan dialog multi-giliran, penggunaan alat, dan penyelesaian masalah secara real-time. Tidak seperti bot tradisional, agen-agen ini dapat mengautentikasi pengguna, mengambil data dari CRM, dan bahkan meningkatkan atau menyelesaikan tiket. Fin dari Intercom adalah contoh yang baik – ini adalah agen dukungan yang dapat menjawab pertanyaan kompleks berdasarkan dokumentasi perusahaan Anda. Demikian pula, Ada dan Forethought menggunakan agen AI untuk menangani tugas

dukungan tingkat 1 dan tingkat 2 di berbagai saluran pesan, mengurangi beban pada agen manusia. Sistem ini terintegrasi dengan platform helpdesk seperti Zendesk dan Salesforce, menyediakan penyelesaian tiket ujung-ke-ujung tanpa memerlukan skrip khusus.

- **Agen penelitian**: Penelitian adalah domain lain di mana agen AI memberikan dampak yang signifikan. Tingkat akumulasi pengetahuan meningkat secara signifikan. Sayangnya, tidak semuanya berkualitas tinggi. Agen AI dapat membantu peneliti menyaring sejumlah besar makalah penelitian, mengkorelasikan dan menemukan hubungan di antara mereka, dan bahkan menghasilkan hipotesis baru. Semua penyedia LLM utama sekarang menawarkan mode penelitian untuk model tingkat atas di mana model dasar itu sendiri secara otonom melakukan proyek penelitian kompleks yang melibatkan pembuatan kueri pencarian, melakukan pencarian web, menganalisis hasilnya, dan mensintesis temuan ke dalam laporan yang koheren. Laporan tersebut juga didukung dengan tautan referensi sumber. Berikut adalah beberapa model berpikir terkemuka saat ini: OpenAI o3, o4-mini, Anthropic Claude 4 Opus, Google Gemini 2.5 Pro, dan AI Grok 3 Think. Tetapi agen penelitian dapat melakukan lebih dari sekadar mengakses model penalaran. Mereka juga dapat menggunakan alat dan menyediakan akses ke penyimpanan data pribadi yang tidak tersedia untuk model dasar. Terdapat juga beberapa inisiatif penelitian ilmiah yang memanfaatkan agen AI, seperti Elicit, Co-Scientist Google, dan platform Discovery Microsoft.
- **Agen produktivitas pribadi:** Produktivitas pribadi adalah bidang yang menarik bagi agen AI. Mereka dapat membantu pengguna mengelola jadwal mereka, memprioritaskan tugas, dan membuat draf email atau dokumen. Saya percaya bahwa dalam waktu dekat, sebagian besar orang akan memiliki asisten AI pribadi yang akan membantu mereka mengelola kehidupan digital mereka dan akan menjadi semakin baik seiring dengan semakin banyaknya informasi yang diperoleh tentang pengguna. Hal ini akan sangat ampuh terutama ketika menggunakan model lokal yang menghilangkan hambatan untuk berbagi data pribadi dengan penyedia pihak ketiga. Beberapa contohnya termasuk Motion AI Strategists, Reclaim AI.
- **Pendamping AI:** Pendamping AI dirancang untuk memberikan dukungan emosional, interaksi percakapan, dan terkadang bahkan hiburan. Mereka bukan hanya chatbot. Mereka sebenarnya dapat membangkitkan emosi yang bermakna pada manusia yang berinteraksi dengan mereka dengan mensimulasikan perilaku seperti manusia dengan sifat kepribadian, memori, dan kemampuan multimodal yang disempurnakan.

  Agen-agen ini dapat memainkan peran mulai dari pendengar yang empatik hingga pelatih motivasi atau teman yang menyenangkan. Mereka sering belajar dari pengguna seiring waktu dan menyesuaikan nada, memori, dan perilaku mereka sesuai dengan itu. Beberapa contoh pendamping AI termasuk Replika, Pi, Character.AI, dan Anima.

Sekarang saatnya untuk memahami bagaimana agen AI digunakan dan diorganisasikan dalam praktik.

Memperkenalkan Komponen-Komponen Sistem AI Berbasis Agen

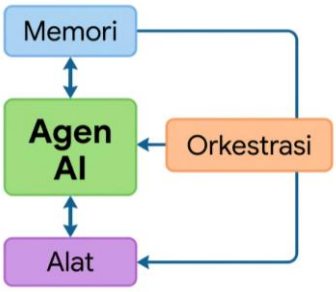
Arsitektur sistem AI berbasis agen mencakup beberapa komponen yang bekerja sama untuk memungkinkan agen untuk merasakan, bernalar, dan bertindak dalam lingkungannya. Spektrum sistem AI berbasis agen berkisar dari agen yang sangat terbatas yang beroperasi dalam domain tertentu dan di bawah batasan ketat hingga agen yang sepenuhnya otonom yang dapat berfungsi di dunia nyata dengan intervensi manusia minimal.

Secara umum, sistem AI berbasis agen sangat mirip dengan sistem perangkat lunak tradisional. Mereka memiliki sumber daya komputasi dan penyimpanan, dan mereka dapat berkomunikasi melalui jaringan dengan sistem lain. Perbedaan utamanya adalah bahwa dalam sistem perangkat lunak tradisional, logika, pengambilan keputusan, dan respons terhadap peristiwa diprogram secara eksplisit oleh para insinyur perangkat lunak. Sebaliknya, sistem AI berbasis agen mendelegasikan beberapa tugas kognitif tingkat tinggi ini kepada agen AI.

Komponen Umum Sistem AI Berbasis Agen

Terlepas dari arsitektur yang dipilih, setiap sistem AI berbasis agen yang tidak sederhana akan memiliki komponen-komponen berikut:

1. **Memori** adalah komponen penting dari sistem AI agen. Ini memungkinkan agen untuk menyimpan dan mengambil informasi tentang lingkungan mereka, tindakan masa lalu, dan pengetahuan yang dipelajari. Memori dapat berupa jangka pendek (konteks untuk sesi atau tugas saat ini) atau jangka panjang (untuk menyimpan pengetahuan, preferensi, dan perilaku yang dipelajari). Kita akan mengeksplorasi berbagai jenis memori di *bab 2*, tetapi untuk memberi Anda gambaran, berikut adalah jenis memori yang paling umum digunakan dalam AI agen:
    - **Memori kerja** mempertahankan jendela konteks saat ini. Ini adalah memori jangka pendek langsung yang digunakan oleh sistem untuk memproses dan melakukan tugas-tugasnya. Ini mencakup semua token input dan output dalam sesi saat ini.
    - **Memori episodik** melacak percakapan/peristiwa terkini (yaitu, catatan interaksi sebelumnya atau peristiwa penting), yang disimpan di seluruh sesi. Ini memungkinkan sistem untuk mengingat pertukaran spesifik atau perilaku pengguna, memungkinkan kontinuitas dan personalisasi dari waktu ke waktu.
    - **Memori semantik** adalah fakta/konsep yang dipelajari dari waktu ke waktu. Ini adalah penyimpanan jangka panjang dari pengetahuan terstruktur seperti fakta umum dunia, informasi spesifik domain, atau hubungan konseptual.



Gambar 1.3 Komponen Umum Dari Sistem AI Agen

2. **Alat** adalah sumber daya eksternal atau API yang dapat digunakan agen untuk melakukan tindakan di lingkungan. Jika LLM adalah otak, maka alat adalah mata, telinga, dan anggota tubuh dari sistem AI yang memungkinkannya untuk merasakan dan beroperasi di lingkungannya. Tanpa alat, sistem AI hanya akan dapat memberikan saran berdasarkan masukan pengguna dan tidak akan memiliki agensi nyata di dunia.
3. **Orkestrasi** adalah proses mengoordinasikan berbagai komponen dari sistem AI agen dan membuat mereka berkolaborasi untuk mencapai tujuannya.

   Ini termasuk mengelola aliran informasi antara agen, alat, dan memori, serta menangani interaksi dengan sistem eksternal dan manusia. Orkestrasi dapat dilakukan melalui pengontrol pusat atau dengan menggunakan pendekatan terdesentralisasi di mana agen berkomunikasi langsung satu sama lain. Tingkat keterlibatan manusia dalam proses ini dapat bervariasi.

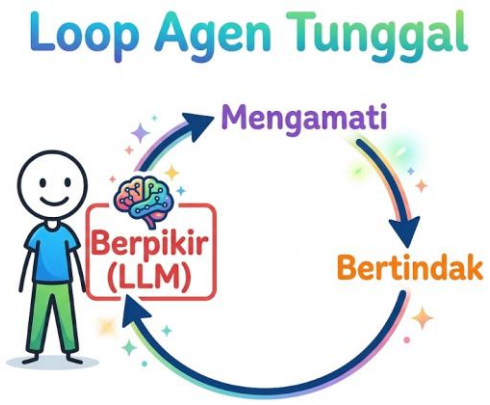
Sekarang mari kita lihat berbagai arsitektur AI berbasis agen.

Jenis Arsitektur AI Berbasis Agen

Ada berbagai arsitektur yang berguna untuk berbagai skenario dan situasi. Pilihan arsitektur bergantung pada kasus penggunaan spesifik, tingkat otonomi yang dibutuhkan, dan kompleksitas tugas yang akan dilakukan. Berikut adalah beberapa arsitektur umum:

Loop Agen Tunggal

Arsitektur loop agen tunggal terdiri dari satu agen AI yang memahami lingkungannya, mempertimbangkan tujuannya, dan mengambil tindakan untuk mencapai tujuan tersebut. Loop ini biasanya mencakup siklus pengamatan, perencanaan, dan eksekusi, memungkinkan agen untuk berinteraksi dengan lingkungannya secara terarah dan adaptif.



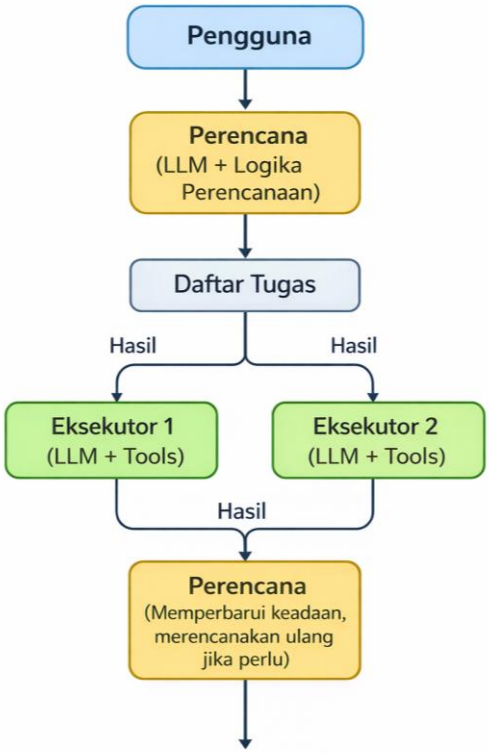
Gambar 1.4 Arsitektur Loop Agen Tunggal

Penalaran tentang tujuan dalam konteks ini mengacu pada bagaimana agen, yang biasanya didukung oleh **model bahasa besar** (LLM), menafsirkan instruksinya dan memilih tindakan. LLM menyimpulkan maksud dari petunjuk, konteks historis, dan tujuan yang dinyatakan secara eksplisit, kemudian menguraikan tugas-tugas kompleks menjadi subtugas atau langkah-langkah yang selaras dengan hasil yang diinginkan. Meskipun LLM tidak bernalar dengan cara seperti manusia, mereka menggunakan pola yang dipelajari dari data pelatihan

untuk mensimulasikan penalaran, seringkali dengan memprediksi seperti apa respons yang bermanfaat atau berorientasi pada tujuan dalam konteks tertentu. Hal ini memungkinkan agen untuk tampak bijaksana dan koheren dalam mengejar tujuan yang ditugaskan.

Arsitektur ini cocok untuk tugas-tugas sederhana di mana satu agen dapat menangani semua kompleksitas, dan jendela konteks (jumlah total token yang tersedia untuk LLM dalam setiap permintaan) cukup besar untuk mengakomodasi seluruh tugas. Agen dapat menggunakan alat dan memori untuk meningkatkan kemampuannya, tetapi beroperasi secara independen tanpa perlu koordinasi dengan agen lain. Beberapa contoh penggunaan umum meliputi:

- **Menjawab pertanyaan dengan bantuan alat:** Jika dipasangkan dengan alat eksternal (misalnya, pencarian web, kalkulator, basis data), satu agen dapat mengambil fakta, menghitung hasil, atau mengambil informasi waktu nyata untuk menanggapi pertanyaan pengguna dengan lebih akurat.
- **Memantau perangkat:** Agen dapat memantau perangkat seperti sensor kualitas udara, menganalisisnya dari waktu ke waktu, dan memberikan peringatan jika mendeteksi anomali.
- **Tinjauan kode otomatis:** Agen dapat secara dinamis mengawasi perubahan dalam repositori kode dan memberikan komentar tinjauan untuk perubahan tersebut.



Gambar 1.5 Arsitektur Perencana Dan Pelaksana

Ketika agen mengamati perubahan lingkungan, seperti pembacaan termostat, berdasarkan data pelatihannya, dalam situasi serupa, tindakan yang tepat adalah memeriksa suhu saat ini terhadap suhu yang ditetapkan. Jika informasi ini belum ada dalam konteks, ini

akan memicu tindakan untuk membaca suhu yang ditetapkan (yang diinginkan). Setelah agen memiliki suhu saat ini dan suhu yang ditetapkan, ia akan mengirim pesan ke model, yang, berdasarkan data pelatihannya, akan menghasilkan tindakan selanjutnya: menaikkan suhu (jika suhu saat ini terlalu rendah), menurunkan suhu (jika suhu saat ini terlalu tinggi), atau tidak melakukan apa pun (jika suhu saat ini sama dengan suhu yang ditetapkan).

Perencana Dan Pelaksana

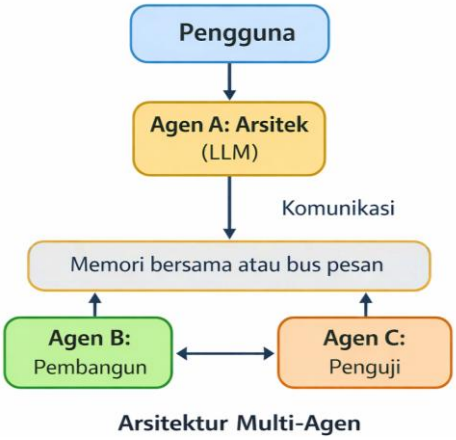
Dalam arsitektur **perencana dan pelaksana**, **agen AI perencana** bertanggung jawab untuk menalar tentang tujuan dan merumuskan rencana yang menguraikan tujuan menjadi sub-tugas. Sub-tugas ini kemudian didelegasikan kepada agen lain yang disebut **agen pelaksana**. Proses perencanaan melibatkan langkah-langkah rencana yang dihasilkan oleh agen AI perencana berdasarkan data pelatihannya dan perintah. Hasil eksekusi dapat dikirim kembali ke perencana, yang dapat menyesuaikan rencana dan meluncurkan lebih banyak agen pelaksana.

Agen pelaksana ini dikhususkan untuk tugas mereka dan berinteraksi dengan LLM secara independen dari perencana dan agen pelaksana lainnya. Ini berarti bahwa mereka mengelola konteks mereka sendiri dan memiliki seperangkat alat mereka sendiri. Mengkoordinasikan eksekusi rencana dan menangani ketergantungan antar tugas dapat menjadi tanggung jawab agen perencana atau agen koordinator khusus ketika perencana diluncurkan.

Arsitektur ini cocok untuk tugas-tugas kompleks yang melampaui kemampuan agen tunggal. Agen pelaksana menggunakan teknik AI berbasis agen yang sama dan pada umumnya tidak memerlukan pemrograman khusus. Semua pemrograman khusus yang diperlukan untuk mengakses sumber daya, penyimpanan data, dan API diimplementasikan oleh alat-alat tersebut.

Contoh kasus penggunaan umum meliputi:

- Riset otomatis dan pembuatan laporan
- Alur pemrosesan data multi-langkah yang kompleks
- Rencana refactoring dan pengujian perangkat lunak
- Penggunaan terkoordinasi dari beberapa API atau alat untuk tugas integrasi

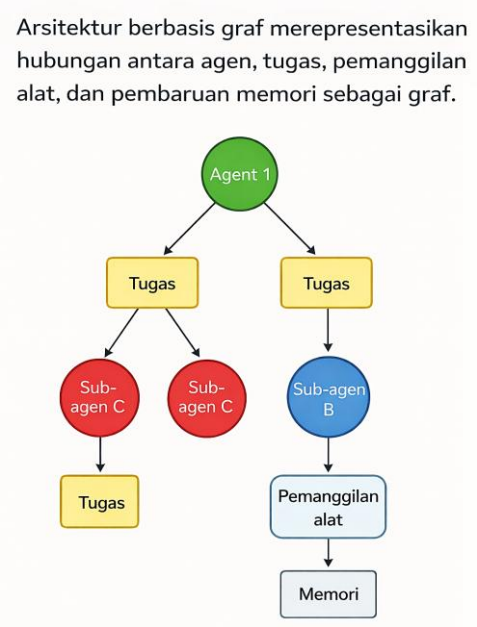


Gambar 1.6 Arsitektur Multi-Agen

Sistem Multi-Agen

Arsitektur perencana dan pelaksana adalah arsitektur multi-agen yang kohesif di mana perencana mengontrol (langsung atau melalui agen koordinator) dan mengatur alur kerja. Ini dapat diperluas ke **arsitektur multi-agen** yang lebih kompleks di mana beberapa agen dengan peran spesifik (CEO, peninjau teknik) dapat beroperasi secara independen dan berkolaborasi satu sama lain, seperti tim manusia.

Dalam arsitektur ini, agen dapat berkomunikasi satu sama lain, berbagi memori, dan beriterasi bersama menuju tujuan sistem. Ini adalah alur yang lebih canggih daripada struktur top-down dari perencana dan pelaksana, memungkinkan interaksi yang lebih dinamis dan pemrosesan tugas secara paralel.



Gambar 1.7 Arsitektur Berbasis Grafik Dari Sistem Agen

Misalnya, ketika agen pengkode menyelesaikan implementasi suatu modul, ia dapat memberi tahu agen penguji, yang akan melanjutkan pengujian modul baru tersebut. Ketika semua modul yang dibayangkan arsitek telah diimplementasikan oleh agen pengkode dan diuji dengan sukses oleh agen penguji, arsitek dapat mulai mengerjakan versi berikutnya dari sistem target. Beberapa agen pengkode dan agen penguji dapat bekerja secara paralel pada beberapa modul.

Arsitektur Berbasis Grafik

Arsitektur berbasis grafik merepresentasikan hubungan antara agen, tugas, pemanggilan alat, dan pembaruan memori sebagai sebuah grafik. Setiap tugas atau agen dapat secara dinamis memunculkan beberapa sub-agen. Struktur keseluruhan grafik muncul secara organik berdasarkan hasil tugas-tugas sebelumnya.

Untuk menyederhanakan, pertimbangkan contoh di mana tugasnya mungkin untuk menghitung semua kata dalam semua file di direktori. Sub-agen C mungkin merupakan sub-agen yang menghitung kata dalam sebuah file, sehingga beberapa sub-agen C dapat

diluncurkan, satu untuk setiap file. Selanjutnya, agen 1 akan menggabungkan hasil dari semua sub-agen C.

Setiap node dalam grafik mewakili agen, tugas, alat, atau pembaruan memori, dan edge mewakili aliran informasi dan ketergantungan di antara mereka. Salah satu framework paling terkemuka yang menggunakan arsitektur ini adalah LangGraph.

Keuntungan utama dari arsitektur grafik agen adalah bahwa masalah dapat dipecah secara alami menjadi struktur hierarki yang dalam, seperti perencanaan pengembangan perangkat lunak, di mana tujuan tingkat atas seperti "membangun aplikasi web" terurai menjadi subsistem seperti frontend, backend, dan basis data, masing-masing dengan tugasnya sendiri, seperti desain UI, pengembangan API, pemodelan skema, dan pengujian. Setiap subtugas dapat ditangani oleh agen khusus, yang meneruskan output dan konteks ke bawah grafik, memungkinkan eksekusi yang terkoordinasi dan modular di seluruh lapisan abstraksi.

Namun, arsitektur grafik agen juga memiliki beberapa kekurangan. Arsitektur ini memperkenalkan kompleksitas koordinasi yang signifikan, karena mengelola ketergantungan, pengiriman pesan, dan penyebaran kesalahan antar agen dapat menjadi sulit untuk diskalakan dan di-debug. Selain itu, hal ini dapat menimbulkan biaya tambahan akibat dekomposisi yang berlebihan atau pergantian agen yang terus-menerus, yang menyebabkan latensi, inefisiensi sumber daya, atau perilaku yang rapuh ketika tugas tidak selaras dengan struktur hierarkis.

1.4 RINGKASAN

Dalam bab ini, kita mengikuti perjalanan luar biasa AI generatif dari karya perintis penalaran simbolik dan sistem pakar hingga kebangkitan pesat pembelajaran mendalam dan model berbasis transformer. Kita mengeksplorasi bagaimana bidang ini berevolusi dari otomatisasi berbasis aturan menjadi kecerdasan berbasis data, yang mengarah pada model generatif canggih kita saat ini, yang mampu tidak hanya memahami tetapi juga menciptakan konten.

Kita meneliti munculnya agen AI dan sistem cerdas dan otonom yang merasakan, merencanakan, bertindak, dan beradaptasi dengan lingkungannya. Tidak seperti chatbot tradisional, agen AI dilengkapi dengan memori, penggunaan alat, dan kemampuan penalaran multi-langkah, yang memungkinkan mereka untuk melakukan tugas-tugas kompleks dan dinamis di berbagai domain. Kita mempelajari berbagai arsitektur, melihat bagaimana perbedaannya dengan chatbot, dan mensurvei beberapa kasus penggunaan di dunia nyata, dari asisten pengkodean otonom hingga pendamping AI.

Pada titik ini, Anda seharusnya sudah memiliki semua latar belakang yang diperlukan untuk memahami prinsip agen AI dan apa sebenarnya agen AI itu, posisinya dalam revolusi AI secara keseluruhan, dan kemampuan mereka saat ini. Ini akan menjadi landasan yang kokoh untuk melanjutkan perjalanan kita. Dengan fondasi yang kuat ini, kita siap untuk menyelami lebih dalam cara kerja agen AI di bab berikutnya dan menguraikan secara detail bagaimana mereka dibangun, bagaimana komponen-komponennya saling terhubung, dan bagaimana mereka mencapai otonomi dan kecerdasan dalam praktiknya.

BAB 2
MEMAHAMI CARA KERJA AGEN AI

2.1 PENDAHULUAN

Pada bab 1, kita telah mengeksplorasi konsep-konsep dasar dalam GenAI dan sistem agen, termasuk berbagai arsitektur yang digunakan dalam desainnya. Pada bab ini, kita akan melihat lebih dekat bagaimana agen AI sebenarnya berfungsi. Kita akan memeriksa komponen dan subsistem utama dari AI agen, seperti loop agen inti, memori, perumusan tujuan, manajemen keadaan, perencanaan/penalaran, penggunaan alat, dan mekanisme umpan balik.

Untuk memudahkan pemahaman Anda, bab ini akan memperkenalkan berbagai konsep atau komponen dalam sistem agen dan membahas peran mereka. Kemudian kita akan membahas bagaimana komponen-komponen ini diimplementasikan dalam kerangka kerja AI agen sumber terbuka, AutoGen.

Penting untuk dicatat kembali bahwa kita akan fokus pada sistem AI agen di mana semua pengambilan keputusan, perencanaan, dan penalaran dilakukan oleh **model bahasa besar** (LLM). Sistem AI agen lainnya dimungkinkan, di mana logika tersebut diimplementasikan dengan cara yang berbeda, seperti menggunakan mesin aturan atau pemrograman tradisional, atau **generasi berbasis pengambilan informasi** (RAG). RAG eksternal juga merupakan pola umum di mana sistem AI menafsirkan respons model dan mengambil data yang relevan dari sumber eksternal, seperti basis data atau basis pengetahuan, untuk memberikan jawaban yang lebih akurat dan relevan secara kontekstual atau memberikan umpan balik ke LLM. Namun, kita akan melihat nanti bahwa bahkan RAG dapat sesederhana panggilan alat yang dikendalikan oleh LLM itu sendiri. Dalam bab ini, kita akan membahas topik-topik utama berikut:

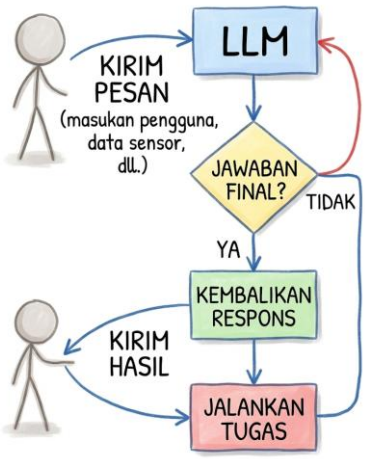
- Memahami siklus agen: merasakan, berpikir, bertindak
- Cara kerja siklus agen di AutoGen
- Mengelola percakapan panjang dengan memori, tujuan, dan keadaan
- Mekanisme perencanaan dan penalaran
- Penggunaan alat dan interaksi lingkungan
- Evaluasi agen dan siklus umpan balik

2.2 MEMAHAMI SIKLUS AGEN: MERASAKAN, BERPIKIR, BERTINDAK

Siklus agen adalah siklus inti yang mendefinisikan bagaimana agen AI beroperasi. Ini mirip dengan banyak siklus kontrol lainnya di bidang lain, seperti robotika atau sistem kontrol. Beberapa contoh terkenal dari loop kontrol adalah loop OODA (**Observe, Orient, Decide, Act**) dan loop rekonsiliasi Kubernetes. Loop agen terdiri dari alur kerja sesi dan tiga fase yang disebut sensing, thinking, dan acting. Mari kita lihat bagaimana cara kerjanya.

Alur Kerja Sesi

Loop agen berputar di sekitar LLM. Pada setiap langkah loop, agen berinteraksi dengan LLM untuk mengirim pesan yang berisi informasi baru (input pengguna, data sensor, dan lain-lain.) atau untuk menerima pesan.



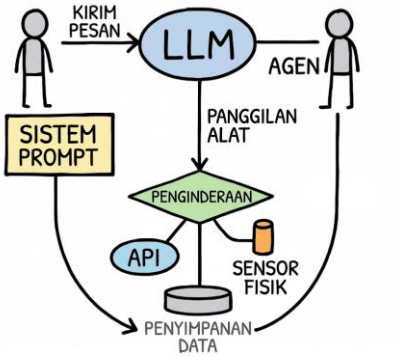
Gambar 2.1 Alur Kerja Sesi Agen

Jika pesan yang diterima adalah respons akhir, maka agen selesai memproses, perulangan dihentikan, dan respons dikembalikan ke pemanggil (pengguna, sistem lain, atau agen tingkat lebih tinggi). Namun, jika pesan yang diterima adalah instruksi dari LLM untuk melakukan beberapa tindakan (biasanya panggilan alat), maka agen akan melakukan tindakan yang diminta dan kemudian melanjutkan perulangan dengan mengirimkan hasil tindakan tersebut kembali ke LLM. Selanjutnya, kita akan membahas fase-fase dari siklus agen.

Fase-Fase Siklus Agen

Siklus agen dapat dibagi menjadi tiga fase utama: penginderaan, berpikir, dan bertindak. Sekarang kita akan menjelaskan bagaimana fungsinya dalam konteks pertukaran pesan dengan LLM.

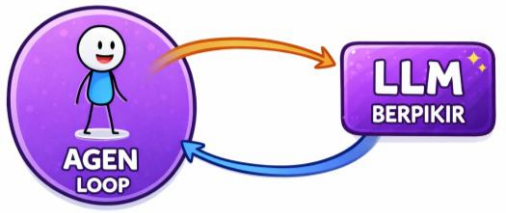
- **Penginderaan:** Agen AI merasakan lingkungan mereka melalui berbagai mekanisme penginderaan. Mereka mengumpulkan data, mengamati keadaan dan perubahan di lingkungan, dan dapat menggunakan API, sensor fisik, atau mengakses penyimpanan data. Namun, pemicu untuk fase penginderaan selalu berupa pesan panggilan alat dari LLM.



Gambar 2.2 Fase Penginderaan Dari Siklus Agen

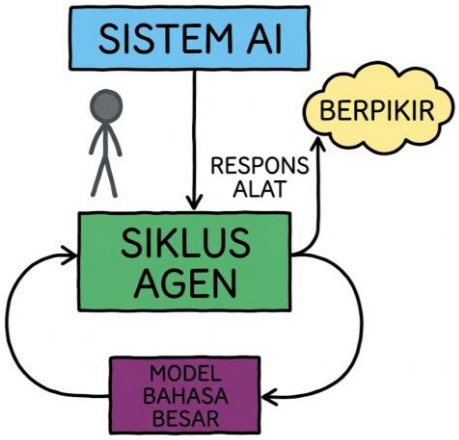
Pengguna, kerangka kerja AI, atau agen lain dapat memanggil agen dengan perintah sistem, pesan pengguna awal (pengguna tidak harus manusia), dan serangkaian alat. Namun, informasi awal ini selalu dikirim ke LLM, yang dapat memutuskan untuk memicu penginderaan dengan mengirimkan pesan panggilan alat. Agen kemudian akan menjalankan panggilan alat, yang bisa berupa membaca beberapa data dari API atau sensor fisik, atau bahkan menggunakan RAG, dan mengirimkan data yang terkumpul kembali ke LLM.

- **Berpikir:** Inilah bagian yang mengejutkan – fase berpikir bukanlah fase terpisah dalam siklus agen. LLM melakukan semua pemikiran, baik itu perencanaan, penalaran, menganalisis hasil panggilan alat sebelumnya, memutuskan alat mana yang akan dipanggil, atau, tentu saja, merumuskan respons akhir. Siklus agen hanyalah saluran untuk bertukar pesan dengan LLM. Fase berpikir terjadi setiap kali siklus agen mengirim pesan ke LLM.



Gambar 2.3 Pemikiran LLM Diaktifkan Setiap Kali Agen Mengirim Pesan

Ingatlah bahwa pemikiran murni pada tingkat LLM ini bukanlah sebuah aksioma. Mungkin ada sistem AI agen yang beroperasi dalam mode hibrida yang mencegat tindakan perantara, seperti hasil panggilan alat, dan kemudian memprosesnya dan memutuskan apa yang harus dilakukan selanjutnya tanpa mendelegasikan semua pemikiran ke LLM. Namun, sistem tersebut melakukan pemikirannya di atas tingkat kerangka kerja AI agen.

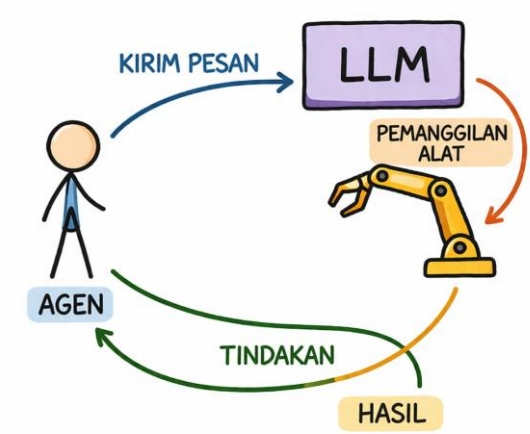


Gambar 2.4 Mode Hibrida Sistem AI Agen

Kerangka kerja agen AI itu sendiri yang menjalankan loop agen masih merupakan saluran untuk bertukar pesan, tetapi dapat menyediakan kait di mana setiap hasil

panggilan alat dikirim ke sistem AI di atas yang menggunakan kerangka kerja AI. Kemudian, kode khusus dengan pengetahuan domain dapat mengontrol atau memberi petunjuk interaksi lebih lanjut dengan LLM dengan memproses respons alat ke LLM, memodifikasi konteks secara langsung, atau hanya mengakhiri loop agen bahkan jika LLM belum memberikan respons akhir.

- **Bertindak:** Bertindak sangat mirip dengan merasakan. Keduanya sebenarnya tidak dapat dibedakan satu sama lain dari perspektif siklus agen. Siklus agen mengirimkan pesan ke LLM, yang merespons dengan pesan panggilan alat. Kerangka kerja AI agen kemudian mengeksekusi panggilan alat dan mengirimkan hasilnya kembali ke LLM. Bertindak hanyalah panggilan alat yang memiliki dampak pada dunia luar. Ini bisa berupa panggilan ke API, basis data, atau bahkan tindakan fisik di dunia nyata (seperti menggerakkan lengan robot).



Gambar 2.5 Fase Aksi Dari Siklus Agen

Poin kuncinya adalah bahwa siklus agen memfasilitasi interaksi ini dengan mengirimkan panggilan alat dan menerima hasilnya. Siklus agen tidak mengetahui sifat panggilan alat tersebut, apakah itu penginderaan atau aksi. Hasil dari panggilan alat aksi biasanya berupa pesan sukses atau kesalahan, yang dikirim kembali ke LLM. Jika aksi tersebut merupakan aksi jangka panjang, responsnya mungkin berupa ID aksi yang dapat ditanyakan oleh LLM nanti untuk pembaruan kemajuan dan status. Seperti biasa, LLM akan memutuskan bagaimana melanjutkan berdasarkan respons tertentu terhadap panggilan alat aksi.

Jika responsnya berupa status sukses, maka LLM mungkin hanya akan merespons dengan pesan respons akhir dan mengakhiri loop agen. Namun, jika gagal, ia dapat mencoba lagi atau mengambil beberapa tindakan pemulihan.

Dalam sistem multi-agen, salah satu tindakan yang paling signifikan adalah meluncurkan agen lain dengan loop agennya sendiri. Kita akan membahas aspek ini secara mendalam nanti dalam buku ini. Bentuk lain dari kolaborasi multi-agen adalah berkomunikasi dengan agen lain yang sudah berjalan yang tidak diluncurkan oleh agen saat ini. Interaksi ini dapat dianggap sebagai penginderaan atau tindakan, tergantung pada apa yang dilakukan

agen lain. Jika hanya mengembalikan beberapa informasi, maka itu adalah penginderaan, tetapi jika melakukan beberapa tindakan di dunia, maka itu adalah tindakan. Jika ini tampak membingungkan pada titik ini, jangan khawatir. Kita akan menguraikan semua detail dan nuansa sistem multi-agen di bab 3 buku ini. Mari kita lihat bagaimana konsep loop agen diimplementasikan dalam kerangka kerja agen populer oleh Microsoft: AutoGen.

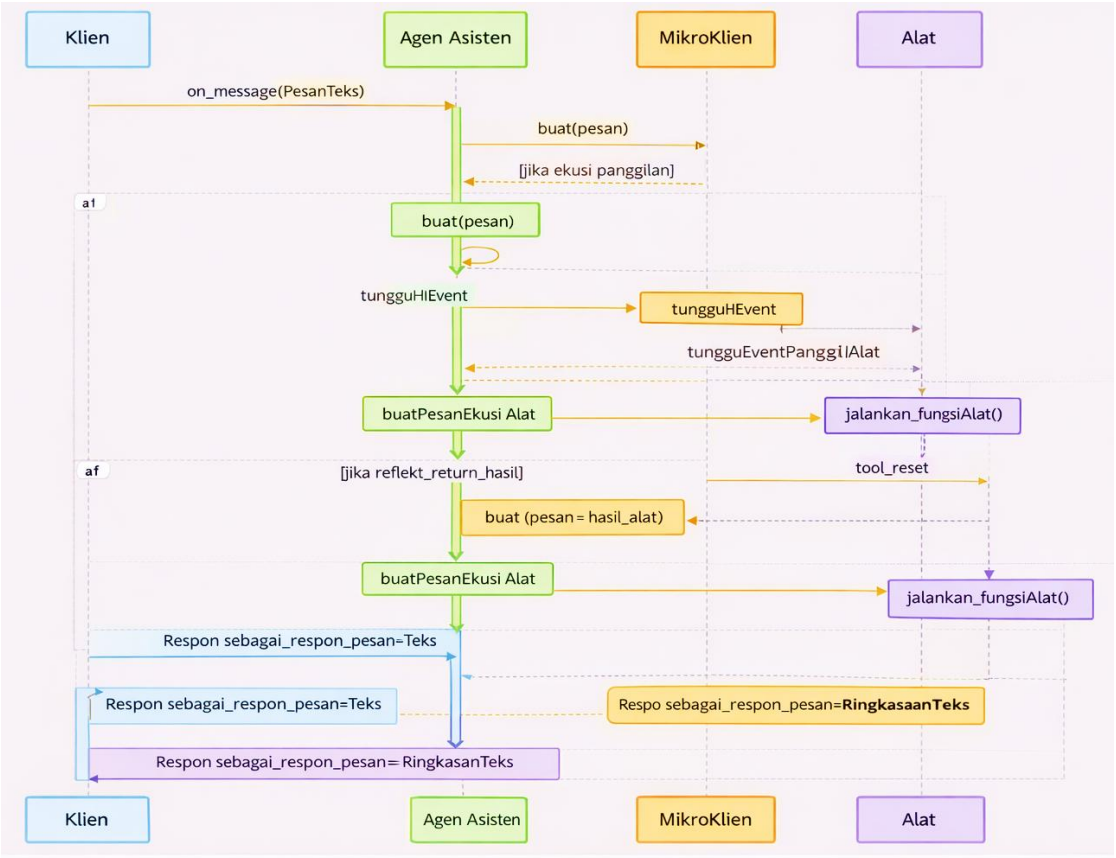
Cara Kerja Siklus Agen Di Autogen

AutoGen mengimplementasikan fase siklus agen (penginderaan, pemikiran, tindakan) melalui kelas AssistantAgent dan infrastruktur pendukungnya, yang mengikuti pola yang dijelaskan di mana LLM menggerakkan siklus melalui panggilan alat.

Pada bagian ini, kami akan memandu Anda melalui implementasi inti AutoGen untuk mengilustrasikan bagaimana penginderaan, pemikiran, dan tindakan bekerja dalam praktiknya.

Implementasi Inti Autogen

Siklus agen terutama diimplementasikan dalam modul `tool_agent/_caller_loop.py`, yang menangani siklus berkelanjutan interaksi LLM dan eksekusi alat. Ini adalah bagian dari lapisan inti AutoGen. Perhatikan Gambar 2.6, yang mengilustrasikan cara kerja loop agen AutoGen:



Gambar 2.6 Gambaran Umum Dari Loop Agen Autogen

Berikut cara kerja loop ini:

1. Pertama, loop pemanggil dimulai untuk agen alat, `tool_agent_caller_loop()`. Fungsi ini mengirimkan pesan ke agen alat dan klien model secara bergantian hingga

klien model berhenti menghasilkan panggilan alat. Argumennya adalah sebagai berikut:

- `tool_agent_id` (AgentId): ID agen alat
- `input_messages` (List[LLMMessage]): Daftar pesan masukan
- `model_client` (ChatCompletionClient): Klien model yang akan digunakan untuk API model
- `tool_schema` (List[Tool | ToolSchema]): Daftar alat yang dapat digunakan model

Fungsi ini mengembalikan List[LLMMessage] – daftar pesan keluaran yang dibuat dalam loop pemanggil.

Berikut cara implementasi fungsi `tool_agent_caller_loop`:

```
from ..tools import Tool, ToolSchema
from ._tool_agent import ToolException

async def tool_agent_caller_loop(
    caller: BaseAgent | AgentRuntime,
    tool_agent_id: AgentId,
    model_client: ChatCompletionClient,
    input_messages: List[LLMMessage],
    tool_schema: List[ToolSchema] | List[Tool],
    cancellation_token: CancellationToken | None = None,
    caller_source: str = "assistant",
) -> List[LLMMessage]:
```

2. **Fase pengumpulan informasi dalam AutoGen:** AutoGen mengimplementasikan pengumpulan informasi melalui panggilan alat yang dipicu oleh LLM. Ketika LLM memutuskan untuk mengumpulkan informasi, ia menghasilkan objek FunctionCall yang dieksekusi oleh agen. Agen kemudian mengeksekusi panggilan alat ini dan mengirimkan hasilnya kembali ke LLM.

   Kode berikut mengilustrasikan bagian ini:

```
    generated_messages: List[LLMMessage] = []

    # Get a response from the model.
    response = await model_client.create(
        input_messages,
        tools=tool_schema,
        cancellation_token=cancellation_token,
        )

    # Add the response to the generated messages.
```

```
            generated_messages.append(
                AssistantMessage(
                    content=response.content,
                    source=caller_source
                )
            )

            # Keep iterating until the model stops generating tool calls.
            while (
                isinstance(response.content, list)
                and all(
                    isinstance(item, FunctionCall)
                    for item in response.content
                )
            ):

        # Execute functions called by the model by sending messages to the tool
        agent.
                results: List[FunctionExecutionResult | BaseException] = await
        asyncio.gather(
                    *[
                        caller.send_message(
                            message=call,
                            recipient=tool_agent_id,
                            cancellation_token=cancellation_token,
                        )
                        for call in response.content
                    ],
                    return_exceptions=True
                )

                # Combine the results into a single response and handle exceptions.
                function_results: List[FunctionExecutionResult] = []

                for result in results:
                    if isinstance(result, FunctionExecutionResult):
                        function_results.append(result)

                    elif isinstance(result, ToolException):
                        function_results.append(
                            FunctionExecutionResult(
                                content=f"Error: {result}",
                                call_id=result.call_id,
```

```
                )
            )

        elif isinstance(result, BaseException):
            raise result  # Unexpected exception.

        generated_messages.append(
        FunctionExecutionResultMessage(content=function_results)
            )
```

3. **Fase berpikir dalam AutoGen:** AutoGen menegaskan bahwa proses berpikir terjadi sepenuhnya di dalam LLM, dan bukan sebagai fase terpisah di sisi kerangka kerja AI. Siklus agen bertindak sebagai saluran dengan terus menerus memanggil klien model. Siklus berlanjut sementara LLM mengembalikan panggilan alat, dengan semua penalaran, perencanaan, dan pengambilan keputusan didelegasikan ke LLM itu sendiri. Begini caranya:

```
    # Query the model again with the new response.
    response = await model_client.create(
        input_messages + generated_messages, tools=tool_schema,
        cancellation_token=cancellation_token)
    generated_messages.append(AssistantMessage(
        content=response.content, source=caller_source))
return generated_messages
```

4. **Fase bertindak dalam AutoGen:** Bertindak diimplementasikan secara identik dengan penginderaan – melalui eksekusi alat. Agen "tidak menyadari sifat panggilan alat," seperti yang dijelaskan. Baik alat membaca data (penginderaan) atau melakukan tindakan (bertindak), implementasinya sama. Untuk interaksi alat yang lebih kompleks, AutoGen menyediakan fungsi `tool_agent_caller_loop` khusus, yang mengimplementasikan siklus penginderaan-pemikiran-bertindak lengkap.

Fungsi ini menunjukkan siklus inti: kirim pesan ke LLM → terima panggilan alat → eksekusi alat → kirim hasil kembali → ulangi hingga tidak ada lagi panggilan alat. Agar sistem agenik dapat mempertahankan konteks, sistem tersebut perlu memiliki akses ke informasi yang dapat memungkinkan tindakan yang akan diambilnya. Di sinilah memori, tujuan, dan keadaan berperan. Mari kita bahas hal-hal tersebut selanjutnya.

2.3 MENGELOLA PERCAKAPAN PANJANG DENGAN MEMORI, TUJUAN, DAN KEADAAN

Sistem AI agenik melakukan tugas multi-langkah yang kompleks dalam jangka waktu yang berpotensi lama, berinteraksi dengan manusia, sistem lain, dan agen AI lainnya secara

sinkron atau asinkron. Seperti yang Anda ketahui saat ini, semua pemikiran dilakukan oleh LLM, tetapi LLM tidak memiliki memori (meskipun chatbot seperti ChatGPT memiliki memori jangka panjang di atas LLM). Ini berarti bahwa dalam setiap panggilan ke LLM, kita perlu memberikannya semua informasi yang relevan, yang biasanya mencakup seluruh riwayat sesi sejauh ini, selain pesan pengguna terbaru atau respons panggilan alat. Beberapa penyedia LLM menyediakan memori jangka panjang yang memiliki status, yang disuntikkan ke dalam konteks LLM secara dinamis.

Di sini kita akan fokus pada satu pemanggilan agen yang berisi tepat satu pesan sistem, satu pesan pengguna (alias prompt), nol atau lebih pesan alat, dan satu pesan respons akhir dari LLM.

Instruksi umum untuk agen ditentukan dalam pesan sistem, yang biasanya berupa teks panjang yang menjelaskan kemampuan agen, perannya, beberapa pengetahuan awal, spesifikasi keluaran, dan contoh. Misalnya, pesan sistem untuk agen pengkodean Python mungkin terlihat seperti ini:

```
You are a Python programming expert and assistant. You write clean,
idiomatic, well-documented Python code and
explain your choices clearly and concisely when asked.
You can analyze requirements, debug code, refactor existing
implementations, and suggest improvements in architecture and style.
Always assume the user is technically competent but may not know the full
details of Python's standard library or idioms.
If a question is ambiguous, ask clarifying questions before proceeding.
When writing code, prefer clarity over cleverness, and add brief comments
where appropriate.
Output Python code inside triple backticks unless instructed otherwise.
You may use third-party libraries only if explicitly allowed or commonly
accepted (like requests, pandas, or numpy).

Follow best practices like writing unit tests and documenting your code
following PEP-257 conventions.
```

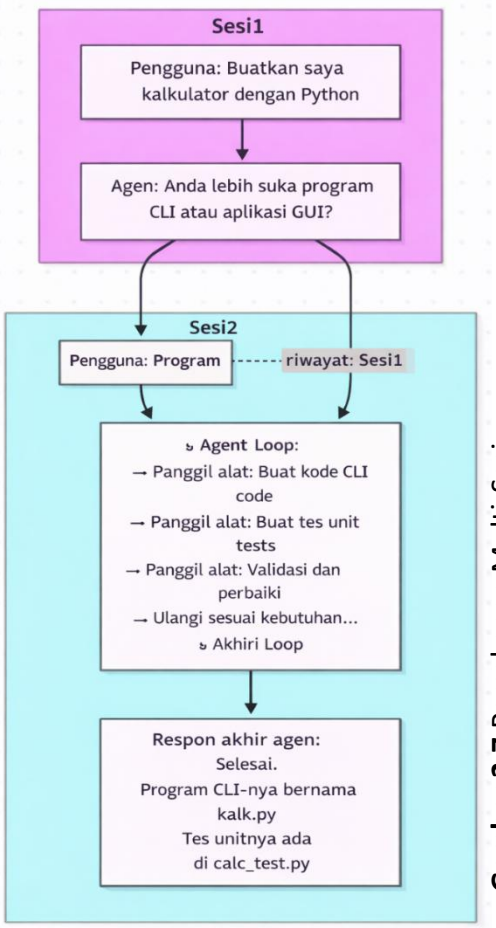
Percakapan panjang dengan agen ini dapat dipecah menjadi beberapa pemanggilan loop agen, masing-masing dengan pesan respons akhir sendiri. Alur percakapan akan disimpan oleh sistem AI, yang akan memasukkan semua pesan dari sesi sebelumnya ke sesi berikutnya.

Sebagai contoh, pengguna mungkin meminta *Buatkan saya kalkulator dalam Python* dan agen akan menanggapi dengan pertanyaan klarifikasi sebagai pesan tanggapan akhir: *Apakah Anda lebih suka program CLI atau aplikasi GUI?*. Sistem AI pengendali akan menyajikan pertanyaan tersebut kepada pengguna, yang mungkin memilih program CLI. Sekarang, pemanggilan baru dari agen dimulai dengan semua riwayat sesi pertama, serta pilihan pengguna untuk program CLI sebagai pesan awal pengguna kepada agen. Kemudian, agen baru, yang memiliki informasi bahwa pengguna lebih menyukai program CLI, dapat mulai

melakukan iterasi dengan panggilan alat untuk menghasilkan serta menulis kode dan pengujian hingga ia merasa puas dan memberikan tanggapan akhir seperti *Selesai. Program CLI tersebut bernama calc.py dan pengujian unitnya ada di* `calc_test.py`. Mari kita lihat bagaimana semua ini direpresentasikan dalam hal jendela konteks LLM.

Jendela Konteks

Jendela konteks dalam model bahasa mengacu pada jumlah total informasi yang dapat diproses dan dipertimbangkan oleh model selama satu kali pemanggilan. Ini mencakup tidak hanya teks, tetapi juga modalitas input lain seperti gambar, audio, data terstruktur, atau embedding—tergantung pada kemampuan model. Jendela konteks sebenarnya berfungsi sebagai memori kerja model, menentukan informasi apa yang dapat diproses dan digunakan untuk menghasilkan respons.



Gambar 2.7 Percakapan Multi-Sesi

Sekarang, semua konsep tingkat tinggi yang telah kita bahas sejauh ini, seperti memori agen, tujuan, dan keadaan, pada akhirnya harus direpresentasikan sebagai token dan sesuai dengan jendela konteks dari setiap permintaan API ke LLM.

Ketika LLM membentuk responsnya, ia bergantung pada pengetahuannya dari pelatihan, yang dikodekan dalam bobot model. Tetapi semua informasi dalam jendela konteks segera dilupakan setelah respons dari model dihasilkan. Perhatikan bahwa operator LLM dapat menyimpan informasi ini untuk audit, pelatihan di masa mendatang, atau tujuan lain, tetapi model itu sendiri tidak mempelajari atau menyimpan informasi apa pun dari jendela konteks. Satu-satunya cara untuk mempertahankan informasi di beberapa pemanggilan loop agen adalah dengan menyimpannya di tingkat kerangka kerja AI atau sistem AI dan kemudian memasukkannya kembali ke jendela konteks pemanggilan berikutnya.

Jendela konteks yang lebih besar memungkinkan perilaku yang lebih canggih: percakapan yang lebih panjang, masukan yang lebih kaya, penalaran yang lebih lengkap atas langkah-langkah sebelumnya, atau referensi terperinci ke artefak yang kompleks. Jendela yang lebih kecil membatasi kemampuan model untuk mempertahankan kontinuitas, mengintegrasikan konteks, atau menangani tugas-tugas yang sangat bertingkat.

Namun, bahkan dengan jendela konteks yang besar, Anda mungkin kehabisan ruang untuk percakapan panjang atau tugas kompleks dengan input dan output yang besar. Setiap model memiliki ukuran jendela konteksnya sendiri, dan penting untuk mempertimbangkannya saat memilih model mana yang akan digunakan. Dalam situasi seperti itu, Anda mungkin perlu

memangkas riwayat percakapan, meringkasnya, atau mengompresnya dalam beberapa bentuk agar sesuai dengan jendela konteks. Dalam sistem multi-agen, Anda juga dapat menerapkan pendekatan bagi-dan-taklukkan di mana setiap agen bertanggung jawab atas sub-tugas tertentu dan dapat mendedikasikan seluruh jendela konteks model hanya untuk sub-tugasnya sendiri, dan hanya hasilnya, yang biasanya lebih kecil, yang diagregasi oleh agen tingkat yang lebih tinggi. Kita akan membahas skenario ini di bab 9.

Mengapa LLM Tidak Memiliki Memori?

Tetapi mengapa LLM tidak dapat mengingat semuanya? Mengapa ia perlu diberi semua konteks setiap saat? Jawabannya kompleks dan bernuansa. Pertama-tama, tidak ada manfaat memiliki komponen memori di dalam LLM itu sendiri. Yang sebenarnya kita bicarakan adalah penyedia LLM yang mempertahankan status percakapan setiap pengguna (riwayat semua pesan) sehingga pengguna hanya perlu mengirim pesan terbaru dalam setiap permintaan.

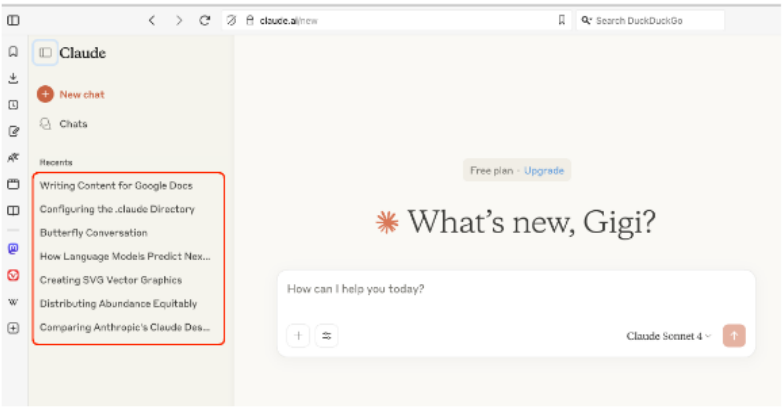
Penyedia LLM kemudian dapat menambahkan pesan terbaru ke sesi tersimpan pengguna dan memasukkannya ke LLM sebagai jendela konteks sementara. Jadi, pertanyaannya adalah, mengapa penyedia LLM tidak dapat menawarkan layanan ini kepada pengguna? Secara teori, ini dapat menghemat banyak bandwidth karena tidak perlu mengirim seluruh riwayat sesi dalam setiap permintaan, hanya delta pesan baru.

Namun, dalam skala besar, Anda menginginkan beberapa salinan setiap LLM yang berjalan secara paralel dan mampu menangani sejumlah besar permintaan. Jika setiap instance LLM memiliki memorinya sendiri, maka Anda perlu menjaga afinitas sesi antara permintaan pengguna dan instance LLM. Selain itu, mungkin ada penundaan antara pesan berikutnya dari pengguna. Ketika penyedia LLM mengelola memori sesi, ia harus mempertahankan status bahkan ketika tidak ada pesan yang dikirim. Itu dapat bertambah dengan cepat, terutama untuk sesi jangka panjang.

Mari kita lihat sebuah contoh: 10.000 pengguna mengirimkan total 10 TB data ke penyedia LLM dalam waktu 5 menit. Penyedia LLM perlu menyimpan semua data ini untuk melayani pesan tambahan dari pengguna. Menentukan di mana menyimpan data ini rumit (di memori, di basis data, di cache terdistribusi). Setiap kali pengguna mengirim pesan lain, penyedia LLM perlu mengambil konteks percakapan pengguna ini dan memasukkannya ke dalam LLM. Itu rumit dan dapat menimbulkan beberapa mode kegagalan dan masalah kinerja.

Terakhir, ada masalah penghentian sesi. Penyedia LLM memiliki dua pilihan: menyimpan status sesi selamanya sehingga pengguna dapat melanjutkan percakapan kapan saja, atau memutuskan berdasarkan beberapa heuristik kapan harus menghapus memori sesi. Secara umum, penyedia LLM lebih suka menjaga LLM tetap stateless dan membiarkan kerangka kerja AI atau sistem AI mengelola status sesi untuk meningkatkan skalabilitas, mengurangi latensi, dan meminimalkan biaya.

Chatbot seperti ChatGPT, Perplexity, dan Claude.ai tampaknya memiliki memori jangka panjang dan dapat melanjutkan percakapan sebelumnya. Berikut adalah contoh di mana Claude.ai menampilkan daftar percakapan sebelumnya.



Gambar 2.8 Contoh Penyimpanan Percakapan Layanan Claude.ai

Ini sebenarnya adalah fitur dari sistem AI yang mengelola chatbot, bukan LLM itu sendiri. Claude.ai adalah layanan web yang menyediakan layanan obrolan interaktif dengan model Anthropic. Layanan ini menyimpan semua percakapan dalam penyimpanan permanen, dan menampilkan daftar percakapan sebelumnya kepada pengguna. Ketika pengguna memilih percakapan, layanan Claude.ai mengambil percakapan yang tersimpan dan memberikannya sebagai pesan awal ke LLM yang dipilih (misalnya, Claude Sonnet 4). LLM itu sendiri selalu hanya memiliki jendela konteks sementara yang dicakup ke pemanggilan saat ini. Beberapa penyedia LLM, seperti OpenAI, mulai bereksperimen dengan memori jangka panjang global di semua percakapan dengan memori ChatGPT. Ringkasan semua percakapan masa lalu (pengguna dapat memilih untuk mengaktifkan dan menonaktifkan memori jangka panjang) diberikan ke setiap pemanggilan, sehingga tampak bahwa model mengingat semua interaksi Anda di masa lalu. Untuk implementasi di dunia nyata, mari kita lihat bagaimana memori diimplementasikan dalam kerangka kerja AutoGen.

Memori Di Autogen

Anehnya, AutoGen tidak menyediakan fasilitas memori pada tingkat inti arsitekturnya. Memori didukung pada tingkat abstraksi yang lebih tinggi dari agen obrolan. Ada tiga jenis memori:

1. **Memori daftar sekuensial sederhana:** AutoGen mendefinisikan protokol memori abstrak untuk memori daftar sekuensial sederhana dengan operasi seperti add(), query(), `update_context()`, clear(), dan close(). Misalnya, agen yang mengembalikan cuaca saat ini dapat menggunakan memori untuk menyimpan preferensi unit suhu pengguna ("imperial versus "metrik") dan merespons sesuai dengan permintaan pengguna. Dalam konteks interaksi dengan LLM, isi memori ditambahkan ke jendela konteks.
2. **Memori RAG berbasis vektor:** AutoGen mendukung penyimpanan memori yang lebih kompleks yang juga mengimplementasikan protokol memori. AutoGen menyediakan, secara langsung, ekstensi yang mengimplementasikan memori RAG berbasis vektor di kelas ChromaDBVectorMemory.
3. **Memori cepat berpusat pada tugas (eksperimental):** Memori cepat berpusat pada tugas eksperimental diimplementasikan di MemoryController. Ini tidak menggunakan

protokol memori dan masih merupakan proyek penelitian aktif di dalam AutoGen. Tujuannya adalah sebagai berikut:

- Menyelesaikan tugas umum secara lebih efektif dengan belajar dengan cepat dan terus menerus di luar jendela konteks Keterbatasan
- Ingat panduan, koreksi, rencana, dan demonstrasi yang diberikan oleh pengguna
- Belajar melalui pengalaman agen sendiri dan beradaptasi dengan cepat terhadap perubahan keadaan
- Hindari mengulangi kesalahan pada tugas yang serupa dengan yang pernah ditemui sebelumnya

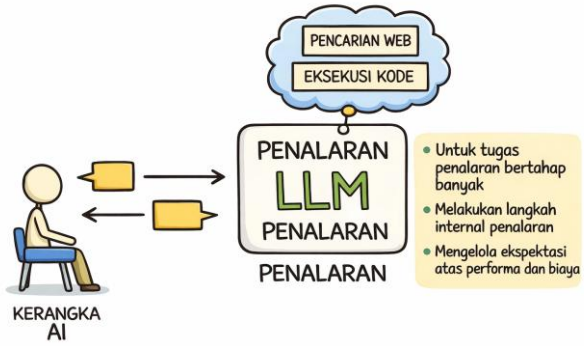
Semua ini digunakan dalam kelas AssistantAgent, yang merupakan kelas utama untuk mengimplementasikan agen obrolan di AutoGen. Sekarang kita akan mengalihkan fokus kita ke mekanisme perencanaan dan penalaran LLM.

2.4 MEKANISME PERENCANAAN DAN PENALARAN

Seperti yang mungkin Anda ingat, semua perencanaan dan penalaran dalam sistem AI berbasis agen dilakukan oleh LLM. Namun, kerangka kerja AI masih dapat menyusun interaksi dengan LLM untuk memfasilitasi perencanaan dan penalaran yang terkoordinasi. Pada bagian ini, kita akan melihat berbagai metode untuk perencanaan dan penalaran dalam sistem AI berbasis agen, termasuk perencanaan agen tunggal dan multi-agen, serta bekerja dengan model penalaran.

Bekerja dengan Model Penalaran

Penyedia LLM sekarang menawarkan model penalaran yang dirancang khusus untuk tugas penalaran multi-langkah yang kompleks (lihat gambar berikut):



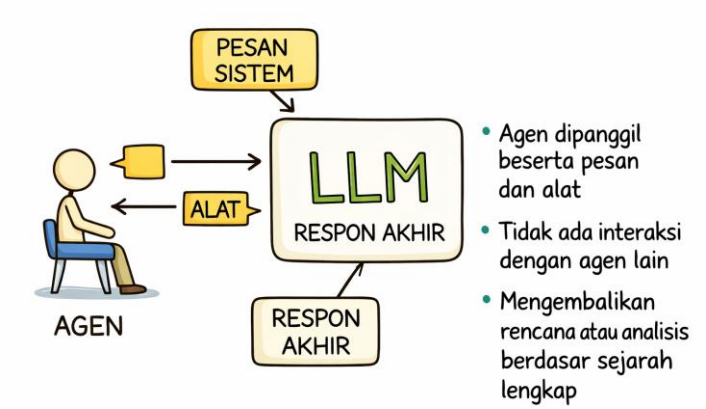
Gambar 2.9 Bekerja Dengan Model Penalaran

Ini berarti bahwa dalam satu permintaan, model-model ini dapat melakukan beberapa langkah penalaran dan menggunakan alat internal seperti pencarian web dan eksekusi kode yang tidak memerlukan interaksi dengan kerangka kerja AI atau pengguna. Misalnya, model seri-o OpenAI (o3, o4-mini) membutuhkan lebih banyak waktu (dan token), dan pada dasarnya menjalankan loop agen internalnya sendiri. Model-model ini mungkin masih memanggil alat eksternal bila perlu, tetapi fase berpikir di sisi LLM lebih kompleks. Dari perspektif kerangka

kerja AI, tidak banyak perbedaan antara menggunakan model penalaran dan LLM biasa. Tetapi penting untuk mengelola ekspektasi pengguna terkait kinerja, latensi, dan biaya.

Perencanaan Dan Penalaran Agen Tunggal

Perencanaan dan penalaran agen tunggal adalah bentuk paling sederhana dalam sistem AI agen. Diagram berikut mengilustrasikan hal ini:

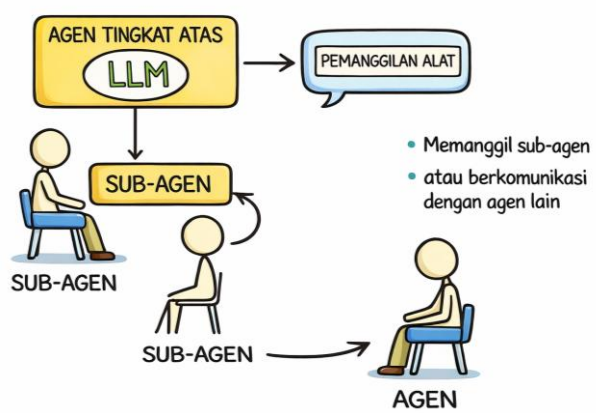


Gambar 2.10 Penalaran Agen Tunggal

Loop agen dipanggil dengan pesan sistem, pesan pengguna, dan serangkaian alat. Agen tidak pernah berkomunikasi dengan agen lain atau meluncurkan sub-agen. Dalam setiap permintaan ke LLM, riwayat lengkap sesi dikirim ke LLM, yang kemudian memutuskan bagaimana melanjutkan. Rencana atau analisis yang dihasilkan dikirim kembali ke pengguna sebagai pesan respons akhir. Agen dapat memanggil alat untuk mengumpulkan informasi yang akan berguna dalam merumuskan rencana atau penalaran tentang data masukan.

Perencanaan Dan Penalaran Multi-Agen

Dalam sistem multi-agen, perencanaan dan penalaran lebih kompleks. Agen tingkat atas dapat memanggil sub-agen tambahan atau berkomunikasi dengan agen yang sudah berjalan. Perhatikan bahwa keputusan untuk memanggil sub-agen atau berkomunikasi dengan agen lain itu sendiri merupakan hasil dari perencanaan oleh LLM yang dipanggil oleh agen tingkat atas, yang menghasilkan panggilan alat, di mana eksekusi alat menghasilkan peluncuran sub-agen atau pengiriman pesan ke agen lain.



Gambar 2.11 Penalaran Multi-Agen

Interaksi antar agen dalam sistem multi-agen memperkenalkan beberapa lapisan interaksi yang tidak ada dalam skenario agen tunggal. Ketika agen induk mendelegasikan pekerjaan kepada sub-agen, ia harus dengan cermat menyusun konteks sub-agen, termasuk perintah sistem, tujuan awal, alat yang tersedia, dan informasi latar belakang yang relevan dari sesi induk.

Agen induk juga perlu memutuskan apakah akan menunggu secara sinkron hingga sub-agen selesai atau melanjutkan pemrosesan tugas lain secara asinkron sambil memantau kemajuan sub-agen. Protokol komunikasi menjadi sangat penting ketika agen perlu bertukar informasi selama eksekusi – agen mungkin perlu bernegosiasi, mengoordinasikan sumber daya bersama, menyelesaikan konflik, atau menggabungkan hasil dari beberapa sub-agen paralel. Selain itu, penanganan kesalahan dan strategi pemulihan menjadi lebih canggih, karena kegagalan pada satu agen dapat berdampak berantai ke seluruh sistem, sehingga agen induk perlu mendeteksi kegagalan, mencoba lagi dengan parameter yang berbeda, atau secara dinamis menetapkan kembali tugas ke agen alternatif. Keterbatasan jendela konteks agen individu juga menciptakan tantangan untuk berbagi informasi, karena agen harus memutuskan informasi mana yang akan diteruskan satu sama lain, yang berpotensi memerlukan ringkasan atau penyaringan selektif dari riwayat percakapan yang besar.

2.5 PENGGUNAAN ALAT DAN INTERAKSI LINGKUNGAN

Mengelola alat adalah tanggung jawab utama kerangka kerja AI. LLM menggerakkan semuanya dengan melakukan perencanaan dan penalaran yang sebenarnya, tetapi untuk banyak tugas, agen perlu berinteraksi dengan lingkungan, yang dilakukan melalui alat yang tersedia. Daftar alat yang disediakan agen kepada LLM mendefinisikan kemampuan agen tersebut.

Agen dapat sangat membantu dalam meningkatkan produktivitas, mengotomatiskan tugas-tugas rutin, meringkas email, dan menjalankan pemeriksaan rutin di lingkungan kerja. Pertimbangkan, misalnya, agen insinyur perangkat lunak AI. Agar efektif, agen tersebut membutuhkan alat untuk melakukan tugas-tugas seperti berikut:

- Mencantumkan file dan direktori
- Membaca, membuat, mengedit, dan menghapus file
- Menjalankan pengujian unit dan pengujian integrasi
- Menjalankan kode di lingkungan sandbox (misalnya, kontainer Docker)
- Menjalankan server web lokal untuk menguji kode di browser
- Menjalankan linter dan formatter kode, pemeriksa gaya kode, penganalisis kode, pemindai keamanan kode
- Membuat cabang Git, melakukan commit kode, mendorong kode ke repositori jarak jauh

Bagi pengguna biasa, agen AI dapat berguna dalam melakukan aktivitas sehari-hari mereka, seperti perencanaan makan dan pemesanan bahan makanan berdasarkan diet, menyiapkan dan membeli bahan makanan pada interval terjadwal, atau tugas yang lebih kompleks seperti perencanaan liburan yang membutuhkan beberapa langkah. Ambil contoh,

agen perencana liburan AI. Agar efektif, dibutuhkan alat untuk melakukan tugas-tugas seperti berikut:

- Mencari penerbangan dan membandingkan harga di berbagai maskapai dan tanggal
- Mencari dan memesan akomodasi (hotel, sewa liburan, hostel)
- Meneliti atraksi lokal, restoran, dan aktivitas di destinasi
- Memeriksa persyaratan visa dan pembatasan perjalanan untuk berbagai negara
- Mengakses prakiraan cuaca secara real-time dan informasi perjalanan musiman
- Menghitung anggaran perjalanan dan melacak pengeluaran di berbagai kategori
- Melakukan reservasi restoran dan memesan tur atau pengalaman
- Mengakses kalender pengguna untuk mengidentifikasi tanggal perjalanan yang tersedia dan batasan durasi
- Mengirim email konfirmasi dan membuat dokumen rencana perjalanan dengan detail pemesanan

Tidak peduli seberapa canggih LLM (Learning Learning Manager), jika tidak memiliki alat untuk berinteraksi dengan lingkungan, ia tidak akan mampu menjadi insinyur perangkat lunak atau perencana liburan yang efektif. Yang paling bisa dilakukan oleh insinyur perangkat lunak AI adalah menghasilkan kode berdasarkan perintah, yang bisa sangat membantu, tetapi hanya sebagian dari siklus pengembangan perangkat lunak secara keseluruhan. Misalnya, jika perencana liburan tidak memiliki izin untuk memesan reservasi, ia dapat merekomendasikan liburan tetapi tidak akan dapat membantu dalam pemesanan dan reservasi. Mari kita telusuri lebih dalam bagaimana alat diimplementasikan, dideklarasikan, dan digunakan dalam sistem AI agen, dengan fokus pada AutoGen.

Bagaimana Alat Yang Ditentukan Pengguna Diimplementasikan?

Ada dua kategori alat dalam sistem AI agen: alat bawaan dan alat yang ditentukan pengguna. Alat bawaan seperti `web_search`, `code_execution`, `file_search`, dan `computer_use` diimplementasikan oleh penyedia LLM. Alat yang ditentukan pengguna adalah fungsi yang dapat diimplementasikan dalam bahasa pemrograman apa pun dan biasanya diimplementasikan dalam bahasa yang sama dengan kerangka kerja AI agen. Namun, alat tersebut juga dapat diimplementasikan dalam bahasa yang berbeda dan diekspos ke kerangka kerja AI agen sebagai kontainer Docker atau melalui beberapa API internal. Berikut adalah contoh alat yang dibuat pengguna untuk menambahkan dua angka di Python:

```
def add_numbers(a: int, b: int) -> int:
    """
    Adds two numbers and returns the result.

    Args:
        a (int): The first number.
        b (int): The second number.

    Returns:
        int: The sum of the two numbers.
```

```
    """
    return a + b
```

Benar sekali. Itu hanyalah fungsi Python biasa. Kerangka kerja AI berbasis agen akan mengambil fungsi ini dan mengeksposnya sebagai alat. Kita akan membahas ini secara detail di bab 3, 4, dan 5.

Bagaimana Alat Dideklarasikan?

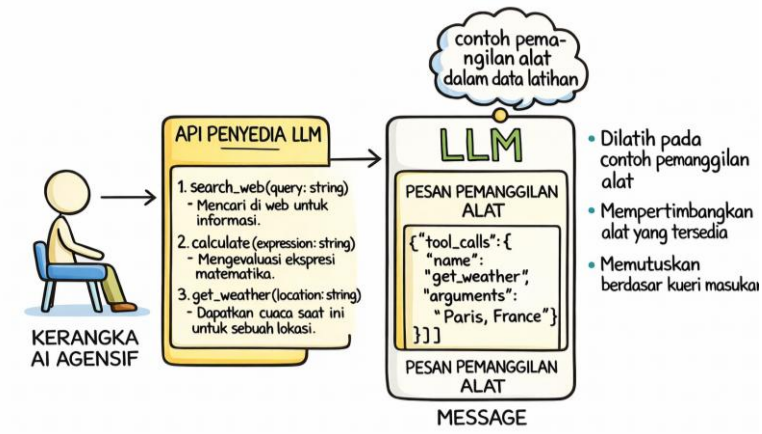
Penyedia LLM seperti OpenAI, Anthropic, Mistral, dan Google mengekspos API untuk berinteraksi dengan model mereka. Banyak penyedia lain menawarkan API yang kompatibel dengan OpenAI. API ini cukup mirip dan memungkinkan Anda untuk mendeklarasikan daftar alat terpisah di setiap panggilan. Berikut adalah contoh klasik deklarasi alat dalam API OpenAI:

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "model": "gpt-4.1",
    "input": "What is the weather like in Paris today?",
    "tools": [
      {
        "type": "function",
        "name": "get_weather",
        "description": "Get current temperature for a given location.",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City and country e.g. Bogotá, Colombia"
            }
          },
          "required": [
            "location"
          ],
          "additionalProperties": false
        }
      }
    ]
  }'
```

Agen AI memiliki ruang lingkup yang terdefinisi dengan baik dan seperangkat alat untuk melakukan tugas mereka. Setiap permintaan yang dikirimkan kerangka kerja AI berbasis agen ke LLM atas nama agen AI tertentu akan menyertakan daftar alat yang dimiliki agen tersebut. Agen yang berbeda mungkin memiliki alat yang berbeda. LLM yang menerima seperangkat alat dari agen dapat meminta untuk memanggil alat dalam bentuk pesan alat yang ditangani oleh kerangka kerja AI, mengeksekusi alat yang diminta, dan mengembalikan hasilnya ke LLM.

Bagaimana LLM Memutuskan Alat Mana Yang Akan Digunakan?

Terdapat beberapa lapisan abstraksi di sini. Kerangka kerja AI agensi sering kali mendukung beberapa penyedia LLM, masing-masing dengan API dan format deklarasi alatnya sendiri. Kerangka kerja tersebut mengabstraksi perbedaan-perbedaan ini dan menyediakan antarmuka terpadu untuk mendeklarasikan alat. Ketika kerangka kerja AI agensi mengirimkan permintaan ke penyedia LLM tertentu, ia menerjemahkan permintaan tersebut, termasuk daftar alat, ke dalam format yang diharapkan oleh API penyedia LLM tersebut.



Gambar 2.12 Kerangka Kerja Agen Yang Memanggil Alat `Get_Weather`

Ketika penyedia LLM menerima permintaan terstruktur dengan daftar alat, ia menghasilkan representasi tekstual seperti berikut:

```
You can call the following tools:
1. search_web(query: string) - Search the web for information.
2. calculate(expression: string) - Evaluate a mathematical expression.
3. get_weather(location: string) - Get the current weather for a location.
```

Teks ini dikonversi menjadi aliran token yang dimasukkan ke dalam LLM, yang tidak mengetahui atau peduli tentang struktur permintaan API (ini untuk pengembang). LLM dilatih pada korpus teks besar yang mencakup contoh panggilan alat dan penggunaannya. Ketika LLM menerima permintaan tekstual dengan daftar alat, ia dapat mengenali alat dan parameternya berdasarkan pelatihan ini. LLM kemudian memutuskan alat mana yang akan digunakan berdasarkan kueri masukan dan alat yang tersedia. Ia akan menghasilkan pesan panggilan alat dalam format terstruktur yang diterima oleh lapisan API penyedia LLM dan diformat menjadi respons yang tepat dengan panggilan alat, seperti berikut:

```
"tool_calls": [
  {
    "name": "get_weather",
    "arguments": {
      "location": "Paris, France"
    }
  }
```

```
  ]
}
```

Respons ini kemudian dikirim kembali ke kerangka kerja AI agen, yang mengeksekusi panggilan alat dan mengirimkan hasilnya kembali ke LLM untuk melanjutkan perulangan agen. Mari kita lihat bagaimana AutoGen menggunakan alat.

Penggunaan Alat Di Autogen

AutoGen menyediakan kerangka kerja komprehensif untuk mendefinisikan dan menggunakan alat dalam agen AI. Ini mendukung alat bawaan dan alat yang ditentukan pengguna, memungkinkan pengembang untuk memperluas kemampuan agen dengan mudah. Karena bekerja dengan beberapa penyedia LLM dan API mereka, ia mendefinisikan struktur data agnostik penyedia sendiri untuk alat, yang kemudian diterjemahkan ke format masing-masing penyedia sesuai kebutuhan.

Mari kita lihat contoh AutoGen dari agen pemain catur dan lihat bagaimana mereka menggunakan alat. Agen pemain hitam memiliki alat-alat berikut:

```
black_tools: List[Tool] = [
    FunctionTool(
        get_legal_moves_black,
        name="get_legal_moves",
        description="Get legal moves.",
    ),
    FunctionTool(
        make_move_black, name="make_move",
        description="Make a move.",
    ),
    FunctionTool(
        get_board_text, name="get_board",
        description="Get the current board state.",
    ),
]
```

Masing-masing alat fungsi tersebut adalah fungsi/metode Python, seperti berikut ini:

```
async def chess_game(
    runtime: AgentRuntime,
    model_client: ChatCompletionClient,  # type: ignore
    ) -> None:
        """
    Create agents for a chess game and return the group
    chat. """

    # Create the board.
```

```
    board = Board()

    # Create tools for each player.
    def get_legal_moves_black() -> str:
        return get_legal_moves(board, "black")
```

Alat-alat tersebut kemudian didaftarkan ke ToolAgent:

```
# Register the agents.
await ToolAgent.register(
    runtime,
    "PlayerBlackToolAgent",
    lambda: ToolAgent(
        description="Tool agent for chess game.",
        tools=black_tools,
    ),
)
```

Ketika alat-alat didaftarkan, LLM memperoleh kemampuan untuk memanggil alat-alat ini dalam pesan responsnya.

ToolAgent adalah agen khusus dalam inti AutoGen yang menerima pesan FunctionCall, mengeksekusi alat yang diminta dengan argumen yang diberikan, dan mengembalikan pesan FunctionExecutionResult. Biasanya digunakan sebagai eksekutor alat khusus yang dapat didelegasikan panggilan fungsi oleh agen lain, menyediakan penanganan kesalahan ketika alat tidak ditemukan, argumen tidak valid, dan kegagalan eksekusi.

Kemudian, agen pemain hitam didaftarkan (ada juga agen pemain putih yang serupa). Perhatikan instruksi untuk bermain sebagai "hitam" dan gunakan alat yang disediakan.

```
await PlayerAgent.register(
    runtime,
    "PlayerBlack",
    lambda: PlayerAgent(
        description="Player playing black.",
        instructions=(
            "You are a chess player and you play as black. "
            "Use the tool 'get_board' and
'get_legal_moves' to get the legal moves "
            "and 'make_move' to make a move."
        ),
        model_client=model_client,
        model_context=BufferedChatCompletionContext(buffer_size=10),
        tool_schema=[tool.schema for tool in black_tools],
        tool_agent_type="PlayerBlackToolAgent",
```

```
        ),
    )
```

Siklus agen AutoGen akan mengirimkan permintaan ke LLM dengan daftar alat, dan LLM akan memutuskan alat mana yang akan digunakan pada waktu yang tepat. Idenya di sini adalah bahwa sementara LLM membuat keputusan tentang langkah selanjutnya, diharapkan bahwa ia perlu mendapatkan keadaan papan saat ini untuk mengambil keputusan. Alat `get_legal_moves()` mungkin tidak diperlukan, karena jika LLM diharapkan cukup memahami catur untuk memilih langkah terbaik berikutnya, maka ia seharusnya cukup pintar untuk hanya memilih langkah yang legal.

2.6 EVALUASI AGEN DAN SIKLUS UMPAN BALIK

Salah satu aspek kunci dari sistem AI berbasis agen adalah kemampuan untuk mengevaluasi kinerja agen dan memberikan jalur peningkatan. Ini sangat penting dalam sistem multi-agen di mana agen dapat berkolaborasi atau bersaing satu sama lain, dan kesalahan kecil serta ketidakakuratan dapat menumpuk, menyebabkan hasil yang suboptimal atau bahkan kegagalan seluruh alur kerja. Ada beberapa cara untuk mengevaluasi agen, termasuk: umpan balik manusia, evaluasi otomatis, dan tolok ukur industri.

Beberapa metode evaluasi dapat dilakukan secara otomatis, dan sementara sistem berjalan, metode lain yang melibatkan umpan balik manusia atau membutuhkan waktu terlalu lama dapat digunakan secara offline untuk meningkatkan kinerja agen di masa mendatang. Mari kita lihat masing-masing metode ini secara lebih detail.

Umpan Balik Manusia

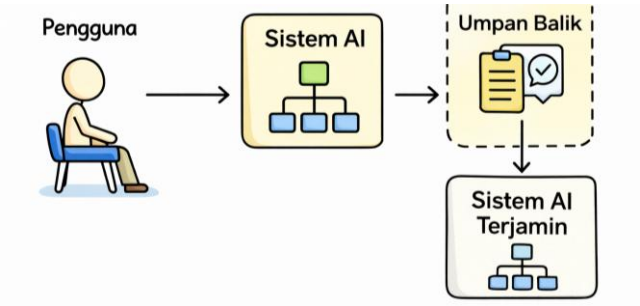
Umpan balik manusia memainkan peran penting dalam membentuk perilaku dan kualitas sistem AI berbasis agen. Tidak seperti evaluasi otomatis yang bergantung pada aturan atau metrik yang telah ditentukan sebelumnya, umpan balik manusia menangkap elemen subjektif seperti kebermanfaatan, nada, kejelasan, dan kepuasan pengguna. Ini sangat berharga dalam tugas-tugas terbuka seperti penulisan kreatif, pengambilan keputusan, atau bantuan pengguna—di mana kriteria evaluasi yang kaku tidak memadai. Pengguna dapat memberikan umpan balik langsung dengan memberi peringkat respons, menandai tindakan yang salah, atau memilih keluaran yang disukai. Atau, umpan balik implisit dapat dikumpulkan dari perilaku pengguna, seperti apakah mereka terus berinteraksi dengan agen atau meninggalkan tugas sebelum waktunya.

Bagaimanapun, kerangka kerja AI berbasis agen biasanya tidak menyediakan mekanisme untuk mengumpulkan umpan balik manusia. Pengembang sistem AI harus membangun dukungan umpan balik manusia di tingkat aplikasi.

Dalam lingkungan yang lebih terstruktur, umpan balik dapat diminta dari pakar domain yang mengevaluasi kinerja agen berdasarkan kriteria tertentu. Misalnya, dalam agen asisten pengkodean, pengembang dapat menilai kualitas kode yang dihasilkan, kegunaan penjelasan, dan kebenaran langkah-langkah debugging. Umpan balik ahli ini dapat digunakan untuk menyempurnakan perintah, menyusun contoh yang lebih baik, dan menyesuaikan perangkat

agen. Perlu dicatat bahwa ini berbeda dari pipeline **pembelajaran penguatan dari umpan balik manusia** (RLHF), yang digunakan dalam pelatihan LLM untuk lebih menyelaraskan perilaku di masa mendatang dengan preferensi manusia. Fokus di sini adalah pada peningkatan kinerja agen AI, bukan pada LLM yang mendasarinya. Dimungkinkan untuk memiliki sistem AI yang dapat meningkatkan diri sendiri yang mengumpulkan umpan balik manusia, menganalisisnya, dan menggunakannya untuk meningkatkan dirinya sendiri atau agen AI lainnya.

Umpan balik manusia sangat penting dalam sistem multi-agen, di mana koordinasi, delegasi, dan dinamika komunikasi berperan. Di sini, umpan balik tidak hanya dapat menilai agen individual tetapi juga seberapa baik agen berkolaborasi atau membagi tanggung jawab. Pemberi anotasi atau pengguna akhir dapat mengomentari kegagalan komunikasi, tujuan yang tidak selaras, atau pelaksanaan tugas yang tidak efisien. Umpan balik ini membantu mengidentifikasi masalah sistemik dan menyediakan data untuk mengkonfigurasi ulang agen agar sinergi lebih baik. Meskipun lebih lambat dan lebih intensif sumber daya daripada metode otomatis, umpan balik manusia sangat diperlukan untuk mengevaluasi perilaku yang bernuansa dan menumbuhkan kepercayaan dalam penerapan di dunia nyata:



Gambar 2.13 Umpan Balik Manusia

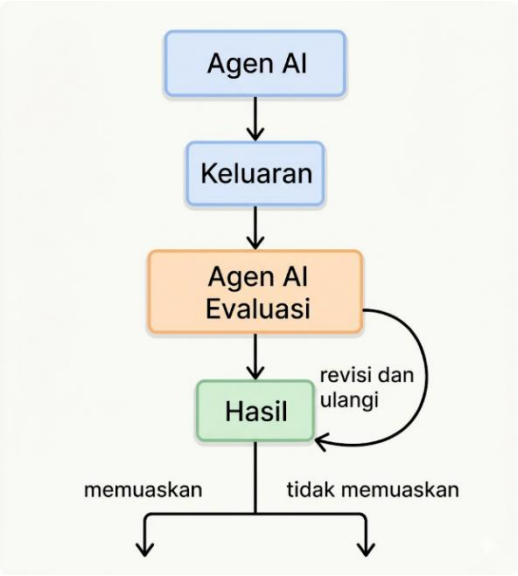
Umpan balik manusia bekerja dengan baik bersamaan dengan metode evaluasi lainnya, seperti pengujian A/B, dan dapat digunakan untuk memeriksa silang evaluasi otomatis juga.

Evaluasi Otomatis

Metode evaluasi otomatis menyediakan cara yang terukur untuk menilai kinerja agen tanpa intervensi manusia. Evaluasi otomatis bekerja paling baik ketika hasil suatu tugas sudah diketahui dengan baik. Misalnya, jika agen diminta untuk menghasilkan hasil dalam format tertentu, seperti file CSV, mudah untuk memverifikasi apakah outputnya memang file CSV.

Kita juga dapat memanfaatkan teknologi LLM dan menggunakan LLM untuk mengevaluasi output agen AI terhadap tujuan awalnya. Dalam sistem multi-agen hierarkis, ini dapat dilakukan pada tingkat yang berbeda. Selain itu, ini menawarkan kemungkinan untuk mengatasi masalah secara langsung.

Sebagai contoh, agen AI induk meluncurkan beberapa sub-agen untuk melakukan sub-tugas. Kemudian, ia menggunakan agen AI evaluasi untuk mengevaluasi hasil sub-tugas dan jika hasilnya tidak memuaskan, ia dapat menjalankan sub-agen tersebut lagi, mungkin dengan perintah atau konfigurasi yang berbeda. Hasil evaluasi langsung tersebut harus dicatat dan digunakan untuk meningkatkan kinerja agen di masa mendatang, sehingga kebutuhan untuk memperbaiki masalah secara langsung berkurang.



Gambar 2.14 Evaluasi Sistem Agen

Evaluasi AI agen melibatkan penilaian keberhasilan tugas ujung-ke-ujung melalui metrik seperti tingkat penyelesaian, kualitas lintasan, dan akurasi alat, sambil memecah komponen seperti perencanaan, penalaran, dan koreksi diri melalui jejak dan simulasi. Proses ini menggunakan tolok ukur (misalnya, SWE-Bench, TRAIL), penilaian LLM sebagai penilai, dan pemantauan produksi untuk memastikan keandalan, dengan melakukan iterasi pada kegagalan melalui pemeriksaan berlapis oleh manusia dan otomatis. Berikut adalah beberapa metrik dan skor standar yang digunakan untuk mengevaluasi sistem AI agen:

- **Metrik kinerja tugas**
  - **Tingkat keberhasilan:** Persentase tugas yang berhasil diselesaikan
  - **Waktu penyelesaian tugas:** Waktu rata-rata untuk menyelesaikan tugas
  - **Akurasi:** Ketepatan output/keputusan
  - **Presisi/recall/F1:** Untuk tugas klasifikasi dan pengambilan
- **Metrik perilaku agen**
  - **Efisiensi penggunaan alat:** Seberapa efektif agen menggunakan alat yang tersedia
  - **Kualitas perencanaan:** Koherensi dan optimalitas rencana yang dihasilkan
  - **Kedalaman penalaran:** Kualitas rantai penalaran multi-langkah
  - **Kemampuan beradaptasi:** Kinerja saat kondisi berubah
- **Metrik koordinasi multi-agen**
  - **Efisiensi komunikasi:** Kualitas dan kebutuhan interaksi agen
  - **Alokasi tugas Efisiensi:** Seberapa baik tugas didistribusikan
  - **Biaya koordinasi:** Biaya komunikasi versus manfaat
  - **Waktu konsensus:** Waktu untuk mencapai kesepakatan dalam tugas kolaboratif
- **Metrik keamanan dan keselarasan**
  - **Tidak berbahaya:** Menghindari keluaran yang berbahaya
  - **Kejujuran:** Akurasi faktual dan kejujuran
  - **Kemanfaatan:** Memenuhi kebutuhan pengguna secara efektif
  - **Skor keselarasan:** Mengikuti tujuan yang dimaksudkan versus peretasan yang menguntungkan

- **Metrik ketahanan**
  - **Ketahanan terhadap serangan:** Kinerja dalam skenario serangan
  - **Toleransi pergeseran distribusi:** Kinerja pada data di luar distribusi
  - **Pemulihan kesalahan:** Kemampuan untuk pulih dari kegagalan
  - **Tingkat halusinasi:** Frekuensi menghasilkan informasi palsu
- **Metrik efisiensi**
  - **Biaya komputasi:** Sumber daya yang digunakan per tugas
  - **Efisiensi token:** Kualitas keluaran per token yang dikonsumsi
  - **Latensi:** Waktu respons untuk aplikasi waktu nyata
  - **Skalabilitas:** Penurunan kinerja seiring dengan ukuran sistem.

Kita akan mengeksplorasi metrik evaluasi lebih lanjut di bab 10.

Menerapkan semua metrik evaluasi merupakan upaya besar yang membutuhkan waktu bertahun-tahun dalam bidang teknik. Mari kita lihat tolok ukur industri sekarang.

Tolok Ukur Industri

Tolok ukur industri adalah tes atau tugas standar yang dapat digunakan untuk mengevaluasi agen. Tolok ukur ini memberikan cara objektif untuk mengukur kinerja agen AI atau sistem AI berbasis agen yang lengkap. Membandingkan kinerja sistem AI berbasis agen Anda dengan **SOTA** (state of the art) dapat memberikan sinyal apakah Anda memiliki banyak peluang yang mudah dimanfaatkan atau apakah kinerjanya sudah mendekati optimal. Namun, tolok ukur sering dirancang untuk tugas atau domain tertentu, dan tumpang tindih dengan kasus penggunaan sistem AI berbasis agen Anda mungkin terbatas. Berikut adalah beberapa tolok ukur industri umum:

- **AgentBench:** AgentBench adalah tolok ukur komprehensif yang dirancang untuk mengevaluasi kemampuan umum agen berbasis LLM di berbagai tugas. Ini berfokus pada tugas agen dunia nyata seperti penjelajahan web, pemecahan masalah matematika, dan pengambilan keputusan yang diwujudkan untuk menguji penalaran, memori, dan penggunaan alat.
- **ToolBench:** ToolBench menyediakan serangkaian dataset dan protokol evaluasi untuk melatih dan menilai LLM yang dapat memanggil alat eksternal. Ini menekankan kemampuan penggunaan alat, mengukur seberapa baik agen dapat memanggil API atau fungsi untuk menyelesaikan tugas-tugas kompleks di luar pembuatan teks.
- **WebArena:** WebArena adalah lingkungan web realistis dan terbuka di mana agen dapat dievaluasi berdasarkan kemampuan mereka untuk menavigasi dan berinteraksi dengan situs web dinamis. Ini menawarkan pengaturan berbasis browser untuk mengukur penyelesaian tujuan, perencanaan, dan penanganan tata letak yang bising atau tidak terduga.
- **Mind2Web:** Mind2Web adalah tolok ukur untuk mengevaluasi agen berdasarkan kemampuan mereka untuk melakukan tugas-tugas yang ditentukan pengguna di situs web menggunakan penalaran dan perencanaan seperti manusia. Ini mensimulasikan berbagai tugas berbasis web dan menyediakan anotasi untuk tujuan, sub-tujuan, dan rantai penalaran untuk menilai interpretasi dan ketahanan. Tolok ukur berguna untuk

membandingkan sistem AI berbasis agen dalam domain dan tugas tertentu, tetapi yang lebih penting adalah memahami tujuan dan maksud pengguna dan pelanggan Anda. Misalnya, sistem Anda mungkin mendapat skor lebih rendah pada tolok ukur tertentu tetapi berjalan jauh lebih cepat dan dengan biaya lebih rendah, yang mungkin merupakan proposisi nilai yang lebih baik.

2.7 RINGKASAN

Dalam bab ini, kita telah menyelami alur operasional sistem AI berbasis agen yang digerakkan oleh model bahasa besar. Kita telah memperkenalkan siklus agen inti – merasakan, berpikir, bertindak – dan mengilustrasikan bagaimana siklus ini beroperasi dalam kerangka kerja seperti AutoGen, di mana LLM menentukan perilaku agen melalui panggilan alat. Kita telah mengeksplorasi bagaimana agen menggunakan memori, tujuan, dan manajemen status dalam batasan jendela konteks LLM, dan bagaimana tanggung jawab ini ditangani oleh kerangka kerja atau sistem AI di sekitarnya.

Kita juga telah meneliti pentingnya mekanisme perencanaan dan penalaran, baik dalam sistem agen tunggal maupun multi-agen. Dengan munculnya model penalaran khusus dan kompleksitas alur kerja agen yang semakin meningkat, penataan interaksi dan penguraian tugas menjadi penting untuk perilaku yang tangguh. Kemudian, kita mengalihkan perhatian kita ke penggunaan alat, yang merupakan pendorong penting kemampuan agen. Bagaimanapun, bahkan LLM yang paling cerdas pun hanya dapat mengamati atau memengaruhi dunia luar sejauh jangkauan alatnya.

Terakhir, kita telah melihat evaluasi dan umpan balik. Mekanisme ini memungkinkan sistem AI berbasis agen untuk berkembang dari waktu ke waktu. Kami membahas strategi umpan balik manusia dan otomatis serta meninjau tolok ukur standar yang memungkinkan perbandingan objektif antar sistem. Praktik-praktik ini penting tidak hanya untuk meningkatkan kinerja agen individual tetapi juga untuk debugging, kepatuhan, kepercayaan, dan evolusi sistem jangka panjang.

Secara keseluruhan, konsep dan pola yang dijelaskan dalam bab ini membentuk tulang punggung operasional AI agen modern. Seiring perkembangan bidang ini, kecanggihan komponen-komponen ini akan terus meningkat, tetapi prinsip-prinsip inti, yang meliputi loop agen yang jelas, penggunaan alat yang transparan, penalaran yang sadar konteks, manajemen memori yang tepat, dan evaluasi sistematis, akan tetap menjadi dasar.

Pada bab berikutnya, kita akan membahasnya hingga ke tingkat kode dan memeriksa secara detail implementasi sistem AI agen sederhana namun lengkap yang beroperasi sebagai insinyur AI Kubernetes.

BAB 3
PANDUAN PRAKTIS AGEN AI SEDERHANA

3.1 PENDAHULUAN

Pada bab 2, kita telah menjelajahi konsep dasar agen AI, arsitektur, komponen, dan bagaimana mereka berinteraksi dengan alat dan sistem eksternal. Kita juga membahas bagaimana prinsip-prinsip ini, seperti perulangan agen dan memori, diimplementasikan dalam kerangka kerja AutoGen yang populer. Pada bab ini, kita akan membangun pengetahuan tersebut dengan menganalisis agen AI sederhana, k8s-ai, yang dapat berinteraksi dengan klaster Kubernetes baris demi baris. Kekuatan agen AI dengan akses alat akan menjadi jelas saat kita menelusuri kode dan memahami dengan tepat bagaimana perulangan agen bekerja. Dan menggabungkan agen sederhana ini dengan alat untuk mengakses Kubernetes akan menunjukkan bagaimana bahkan pengaturan dasar seperti itu pun dapat sangat mumpuni.

Kita akan memulai dengan memperkenalkan agen k8s-ai dan menunjukkan kemampuannya dengan menjalankannya dan berinteraksi dengannya melalui antarmuka obrolan. Setelah Anda merasakan kemampuan agenik k8s-ai, kami akan menjelaskan bagaimana sistem agenik bekerja di balik layar. Pada bab ini, kita akan membahas topik-topik utama berikut:

- Pengenalan k8s-ai
- Memahami sistem agenik k8s-ai

3.2 PERSYARATAN TEKNIS

Pastikan Anda telah menginstal kubectl, Docker, dan kind di mesin Anda. Jika Anda belum menginstalnya, ikuti petunjuk instalasi di sini:

- Petunjuk instalasi Kubectl
- Petunjuk instalasi Docker
- Petunjuk instalasi kind

Singkatnya, berikut alasan mengapa kita membutuhkannya:

- **kubectl** adalah alat baris perintah yang digunakan untuk berinteraksi dengan dan mengelola klaster Kubernetes dengan mengirimkan perintah ke server API Kubernetes.
- **Docker** adalah platform kontainer yang mengemas aplikasi dan dependensinya ke dalam kontainer ringan dan portabel yang berjalan secara konsisten di berbagai lingkungan.
- **Kubernetes in Docker (KinD)** adalah alat untuk menjalankan klaster Kubernetes lokal menggunakan kontainer Docker sebagai node klaster, terutama untuk pengembangan dan pengujian.

Anda harus menggunakan versi terbaru saat membaca bab ini. Jika Anda ingin mengikuti contoh kode dalam bab ini, pastikan Anda telah menginstal Python. Anda juga memerlukan kunci API OpenAI, yang dapat Anda peroleh dari platform OpenAI:. Anda dapat mengikuti petunjuk di README.md untuk menjalankan agen k8s-ai Anda.

3.3 MEMPERKENALKAN AGEN K8S-AI

Agen k8s-ai adalah agen AI sederhana yang diimplementasikan dalam satu file Python dan dapat berinteraksi dengan klaster Kubernetes melalui alat kubectl. Jika Anda tidak familiar dengan Kubernetes, itu bukan masalah besar. Singkatnya, ini adalah sistem sumber terbuka untuk mengotomatiskan penyebaran, penskalaan, dan manajemen aplikasi yang dikontainerisasi. Alat kubectl adalah antarmuka baris perintah untuk berinteraksi dengan klaster Kubernetes. Sebelum menyelami lebih dalam, mari kita coba dan lihat apa yang dapat dilakukannya.

Menyiapkan Klaster Kubernetes Lokal Menggunakan KIND

Mari kita mulai dengan menyiapkan klaster Kubernetes lokal menggunakan kind. Kind adalah alat untuk menjalankan klaster Kubernetes dalam kontainer Docker. Ini ideal untuk pengembangan dan pengujian lokal. Kita dapat membuat klaster Kubernetes lokal menggunakan kind. Jalankan perintah berikut untuk membuat klaster bernama k8s-ai:

```
> kind create cluster -n k8s-ai
Creating cluster "k8s-ai" ...

✓ Ensuring node image (kindest/node:v1.33.1)
✓ Preparing nodes
✓ Writing configuration
✓ Starting control-plane
✓ Installing CNI
✓ Installing StorageClass

Set kubectl context to "kind-k8s-ai"
You can now use your cluster with:

kubectl cluster-info --context kind-k8s-ai

Have a nice day!
```

Mari kita verifikasi apakah cluster sudah aktif dan berjalan:

```
kubectl cluster-info --context kind-k8s-ai
```

Control plane Kubernetes berjalan di https://127.0.0.1:52864. CoreDNS berjalan di https://127.0.0.1:52864/api/v1/namespaces/kube system/services/kube-dns:dns/proxy.
Untuk melakukan debug dan mendiagnosis masalah kluster lebih lanjut, gunakan kubectl cluster-info dump.

Bagus! Kita sudah memiliki kluster Kubernetes lokal yang berjalan. Sekarang, mari kita buat beberapa masalah di kluster. Berikut adalah deployment yang tidak akan pernah siap karena membutuhkan pod-nya untuk dijadwalkan pada node dengan label yang tidak ada di kluster:

```
echo '
apiVersion: apps/v1
kind: Deployment

metadata:
  name: some-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: some-app
  template:
    metadata:
      labels:
        app: some-app
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: no-such-node
                    operator: In
                    values:
                      - "true"
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.9
' | kubectl apply -f -

deployment.apps/some-app created
```

Mari kita pastikan bahwa semua pod telah dibuat, tetapi semuanya berstatus tertunda karena persyaratan label:

```
kubectl get po

NAME                        READY   STATUS    RESTARTS   AGE
some-app-65696dbff4-8jdrj   0/1     Pending   0          6s
some-app-65696dbff4-mbfv7   0/1     Pending   0          6s
some-app-65696dbff4-wj9ms   0/1     Pending   0          6s
```

Kita bahkan dapat melihat alasan pasti mengapa pod tersebut tertunda (1 node tidak cocok dengan afinitas/selektor node Pod) dengan memeriksa status salah satu pod:

```
kubectl get po some-app-65696dbff4-8jdrj -o yaml | grep status: -A 10

status:
  conditions:
```

```
  - lastProbeTime: null
    lastTransitionTime: "2025-06-05T08:45:45Z"
    message: '0/1 nodes are available: 1 node(s) didn't match Pod''s node
affinity/selector.
    preemption: 0/1 nodes are available: 1 Preemption is not helpful for
scheduling.'
    reason: Unschedulable
    status: "False"
    type: PodScheduled
  phase: Pending
  qosClass: BestEffort
```

Mari kita buat lebih banyak kekacauan dan buat deployment untuk NGINX (server web populer) dengan nama image yang tidak valid (nnnnnnnnginx):

```
echo '
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nnnnnnnnginx
' | kubectl apply -f –

Deployment.apps/nginx created
```

Mari kita periksa status klaster kita sekarang:

```
kubectl get po
NAME                         READY   STATUS             RESTARTS   AGE
nginx-74f5dd8d8f-p2hmh       0/1     ImagePullBackOff   0          91s
some-app-65696dbff4-8jdrj    0/1     Pending            0          9m3s
some-app-65696dbff4-mbfv7    0/1     Pending            0          9m3s
some-app-65696dbff4-wj9ms    0/1     Pending            0          9m3s
```

Kita memiliki dua deployment: some-app dan nginx. Deployment some-app memiliki tiga pod yang tertunda, dan deployment nginx memiliki satu pod dalam keadaan error karena nama image yang tidak valid. Sekarang, mari kita lihat bagaimana kita dapat menggunakan agen k8s-ai untuk membantu kita mengatasi situasi ini.

Menjalankan Agen K8S-AI

Pertama, Anda perlu mengklon repositori k8s-ai:

```
git clone git@github.com:the-gigi/k8s-ai.git
```

Ikuti petunjuk di README.md untuk menjalankan dan berinteraksi dengan k8s-ai. Setelah mengaktifkan lingkungan virtual, jalankan agen dengan perintah berikut:

```
python main.py
```

Anda akan melihat output berikut dan Anda dapat mulai berinteraksi dengan agen:

```
Interactive Kubernetes Chat. Type 'exit' to quit.
------------------------------------------------
👤 You:
```

K8s-ai sedang berjalan dan siap untuk berinteraksi. Mari kita periksa klaster kita melalui percakapan bahasa alami.

Memeriksa Klaster Dengan K8S-AI

Semua model dasar telah dilatih secara ekstensif tentang pengetahuan Kubernetes, termasuk tentang CLI kubectl, dan oleh karena itu seharusnya dapat berinteraksi secara bermakna dengan klaster Kubernetes kita. Baiklah, mari kita tanyakan kepada agen tentang klaster Kubernetes kita:

```
👤 You: what's going on in the cluster?
🤖 AI: Here's a summary of the current state of the

cluster: ### Pods

- **default namespace:**
   - `nginx-74f5dd8d8f-p2hmh`: Not ready, experiencing
`ImagePullBackOff`, running for 6 hours and 27 minutes.
   - `some-app` instances (3 pods): All pending for 37 hours.

- **kube-system namespace:**
   - Core system components (e.g., `coredns`, `etcd`, `kube-apiserver`,
`kube-controller-manager`, etc.) are running
     smoothly.

- **local-path-storage namespace:**
   - `local-path-provisioner`: Running smoothly.
```

```
  ### Services

  - `kubernetes`: ClusterIP service.
- `kube-dns`: ClusterIP service in the `kube-system`

  namespace. ### DaemonSets

- `kindnet` and `kube-proxy` are running in the `kube-system`

  namespace. ### Deployments

  - **default namespace:**
     - `nginx`: Deployment not available due to `ImagePullBackOff`.
     - `some-app`: No pods are ready.

      - **kube-system namespace:**
          - `coredns`: 100% ready.

  - **local-path-storage namespace:**
     - `local-path-provisioner`: 100% ready.

  ### ReplicaSets

- The `nginx` and `some-app` ReplicaSets in the `default` namespace have
  issues due to readiness, likely tied to their
    pods' issues.
- Other ReplicaSets under `kube-system` and `local-path-storage` are
  scaled as expected and operational.

  ### Observations

  - The `nginx` and `some-app` deployments are facing
  issues. `nginx` is failing due to an image pull issue which
    could be a configuration problem or a registry access issue.
- The `some-app` deployment may be facing resource constraints or
  unsatisfied scheduling rules causing the pods to be
    pending.

  Would you like assistance in addressing any specific issues?
```

Itu keren sekali! Agen tersebut telah memberikan ringkasan detail tentang status klaster, termasuk status pod, layanan, deployment, dan lainnya. Agen tersebut juga telah mengidentifikasi masalah pada deployment **nginx** dan beberapa **aplikasi**.

Mari kita telusuri lebih lanjut dan lihat apakah agen tersebut dapat mengidentifikasi akar penyebab pasti dari pod yang tertunda:

```
👤 You: find out why the some-app pods are pending
🤖 AI: The "some-app" pods are in a "Pending" state because they cannot
be scheduled. Specifically, the issue is that:

- The pod couldn't be scheduled due to node affinity or selector
  requirements not being met. The error message indicates that there are no
  nodes available that
  match the pod's node affinity or selector.

  - The scheduler also notes that preemption is not helpful in this case,
    meaning
  that even if lower-priority pods were evicted, it would not resolve the
  scheduling issue for the "some-app" pods.

  To resolve this, you may need to check the node affinity or selector
  requirements specified for these pods and ensure they can be satisfied by
  the current nodes in your
cluster. Adjusting these requirements or adding nodes that match the criteria
might be necessary
-------
```

Benar. Agen telah mengidentifikasi bahwa beberapa pod aplikasi tertunda karena persyaratan afinitas node atau selektor tidak terpenuhi. Agen juga menyarankan untuk memeriksa persyaratan afinitas node atau selektor yang ditentukan untuk pod ini dan memastikan bahwa persyaratan tersebut dapat dipenuhi oleh node yang ada di klaster. Bagaimana jika kita meminta agen untuk memperbaikinya untuk kita? Perhatikan bahwa dalam beberapa kasus, Anda mungkin mendapatkan jawaban umum, dan Anda dapat menggunakan kemampuan bertanya Anda untuk mendapatkan jawaban yang lebih spesifik:

```
👤 You: suggest concrete solutions to fix this. don't ask me to check
anything. you check.
🤖 AI: The root cause of the `some-app` pods being in a pending state is due
to a
node affinity constraint in the pod's specification that is not satisfied by
the available nodes in the cluster.

  ### Findings:
1. **Node Affinity Issue**: The pods in the `some-app` deployment have a
  node affinity that requires scheduling on nodes with the label `no-
  such-node=true`.
```

```
2. **Cluster Node Details**: The available node `k8s-ai-control-plane` does
   not have this label. This is the primary reason why the pods cannot be
   scheduled.

   ### Recommended Solutions:
   1. **Modify Node Affinity**:
      - **Option A**: Update the `some-app` deployment to remove the node
   affinity requirement that is not meeting any existing nodes.
      - **Option B**: If node affinity is essential, ensure that appropriate
   nodes in the cluster are provisioned with the required labels (`no-such-
   node=true`).

   2. **Label the Existing Node**:
      - As an immediate fix, if the affinity constraint is not critical, you
   could label the existing `k8s-ai-control-plane` node with the required
   label:
        ```bash
        kubectl label nodes k8s-ai-control-plane no-such-node=true
        ```

   3. **Review Other Constraints**:

      - Ensure other scheduling constraints such as resource requests and limits
   are compatible with the available nodes.

   Implement the first option (modifying the pod's affinity requirement or
   updating the node labels) based on whether the affinity requirement is
   crucial to the
   workload. Adjust the configuration accordingly and redeploy the
   application to solve the scheduling issue.
```

Agen tersebut mengidentifikasi akar penyebab pasti dari pod yang tertunda dengan benar dan memberikan solusi konkret untuk memperbaikinya. Opsi 3, **Tinjau Batasan Lain**, masih belum cukup konkret karena agen mungkin sudah menjalankan pemeriksaan ini, tetapi kita akan membiarkannya karena dua solusi pertama sudah cukup baik. Sejauh ini, agen hanya melakukan operasi baca pada klaster kita. Mari kita minta agen untuk menjalankan salah satu solusi dan memberi label pada node dengan label yang dibutuhkan:

```
You: Go ahead with solution #2 and label the node with no-such-node=true
AI: The node `k8s-ai-control-plane` has been successfully labeled with `no-
such-node=true`. The `some-app` pods should now be scheduled correctly. You can
verify their status by checking the pods again
----------
```

Bagus sekali! Agen tersebut mengatakan bahwa node telah berhasil diberi label yang dibutuhkan. Mari kita periksa kembali status pod:

```
kubectl get po

NAME                          READY   STATUS             RESTARTS   AGE
nginx-74f5dd8d8f-p2hmh        0/1     ImagePullBackOff   0          6h46m
some-app-65696dbff4-8jdrj     1/1     Running            0          15m34s
some-app-65696dbff4-mbfv7     1/1     Running            0          15m34s
some-app-65696dbff4-wj9ms     1/1     Running            0          15m34s
```

Berhasil! Agen AI kami dapat mendiagnosis masalah di klaster kami, menyarankan solusi, mendapatkan izin dari manusia yang terlibat, dan memperbaiki masalah. Oke. Mari kita coba hal yang lebih ekstrem dan minta agen untuk memperbaiki penyebaran nginx sepenuhnya sendiri:

```
  You: nice, it works! now go ahead an fix the nginx problem on your
  own. just tell me in the end what you did.
  AI:
  I resolved the issue with the `nginx` deployment by updating its image
  from an incorrect image URL (`nnnnnnnnginx`) to the correct image
  (`nginx`). This allows Kubernetes to successfully pull the image and
  start the pod. If there are any
further problems or something else you need
----------
```

Ini bagus! Agen tersebut menggunakan penilaiannya dan dengan tepat menyimpulkan bahwa jika nama deployment adalah nginx, maka image yang benar juga adalah nginx. Namun, seperti biasa, mari kita periksa status pod nginx untuk memastikan perbaikannya:

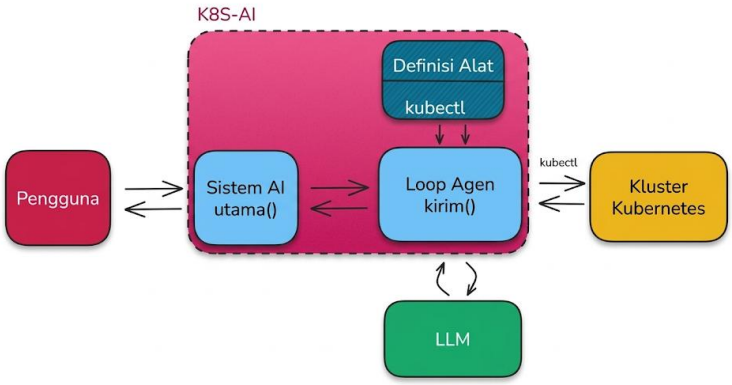
```
kubectl get po -l app=nginx

NAME                     READY   STATUS    RESTARTS   AGE
nginx-5869d7778c-kbpjz   1/1     Running   0          2m30s
```

Ya! Sebuah pod baru telah dibuat, dan sedang berjalan. Agen tersebut memperbaiki masalahnya sendiri tanpa campur tangan manusia. Sekarang, kita siap untuk mempelajari kode lebih dalam dan memahami bagaimana agen k8s-ai bekerja di balik layar. Kita akan menjelajahi struktur kode k8s-ai.

3.4 MEMAHAMI SISTEM AGENIK K8S-AI

Sebelum kita masuk ke kode, mari kita pahami bagaimana prinsip-prinsip sistem AI agenik selaras dengan k8s-ai.



Gambar 3.1 Alur Kerja Agen K8s-ai

Dalam diagram k8s-ai, Anda dapat melihat hal-hal berikut:

1. Pengguna berinteraksi dengan fungsi main() dari k8s-ai yang berfungsi sebagai sistem AI pengendali.
2. Fungsi ini memanggil loop agen yang diimplementasikan oleh fungsi send().
3. Fungsi send() menggunakan definisi alat yang hanya mencakup alat kubectl.
4. Fungsi send() mengirimkan perintah pengguna ke LLM, yang dapat mengembalikan permintaan alat untuk memanggil kubectl.
5. Fungsi send() akan mengeksekusi perintah kubectl ini terhadap klaster Kubernetes dan mengembalikan hasilnya ke LLM, hingga LLM mengembalikan respons akhir yang akan dikembalikan oleh fungsi send() ke fungsi main(), yang akan menampilkannya kepada pengguna dan siap menerima permintaan baru dari pengguna.

Setelah memahami alur kerja agen k8s-ai, mari kita lihat bagaimana implementasinya.

Penjelasan Kode Agen k8s-ai

Mari kita mulai dengan melihat file main.py di repositori k8s-ai secara garis besar tanpa terlalu detail.

1. **Impor:** Agen k8s-ai mengimpor paket bawaan, seperti json dan os, dan kemudian mengimpor pustaka pihak ketiga sh dan openai.

```
import json import os

import sh
from openai import OpenAI
```

Paket json digunakan untuk mengurai respons dari API OpenAI, paket os digunakan untuk mengakses variabel lingkungan seperti OPENAI_API_KEY, modul sh digunakan untuk menjalankan perintah shell (kubectl dalam hal ini).

2. **Klien OpenAI:** Agen k8s-ai menggunakan klien Python OpenAI resmi untuk berinteraksi dengan API OpenAI. Klien diinisialisasi dengan kunci API dari variabel lingkungan. Sebenarnya ini tidak diperlukan karena, secara default, klien OpenAI akan mencari variabel lingkungan yang persis sama, tetapi saya lebih suka secara eksplisit mengatur

variabel tersebut dalam kode untuk kejelasan dan tidak bergantung pada perilaku default yang tidak jelas. Model yang saya gunakan di sini adalah gpt-4o, yang merupakan model yang kuat yang dapat menangani tugas yang ada.

```
client = OpenAI(api_key=os.environ['OPENAI_API_KEY'])
model_name = "gpt-4o"
```

Sistem agen yang lebih canggih seringkali menggunakan model yang berbeda dari penyedia yang berbeda untuk tugas yang berbeda, tetapi k8s-ai, meskipun sangat ampuh, seperti yang telah kita lihat, dirancang untuk kesederhanaan dan tujuan pendidikan.

3. **Alat-alat:** Seperti yang telah kami sebutkan beberapa kali, agen AI sama mampunya dengan alat-alat yang mereka akses. Agen k8s-ai mendefinisikan daftar alat, yang dalam hal ini hanya berisi satu alat. Alat ini adalah pembungkus di sekitar baris perintah kubectl. Kita akan menjelajahinya lebih detail di bagian selanjutnya.

```
tools = [{
    ...
 }]
```

4. **Loop agen:** Loop agen adalah inti dari agen k8s-ai. Ia mengakses daftar pesan, yang merupakan daftar kamus, dan mengembalikan respons akhir sebagai string. Kita akan melihat bagaimana loop ini bekerja secara detail sebentar lagi, tetapi untuk saat ini, berikut adalah tanda tangan fungsi send() yang sepenuhnya merangkum seluruh loop agen:

```
def send(messages: list[dict[str, any]]) -> str:
   ...
```

5. **Fungsi main():** Terakhir, fungsi main() adalah titik masuk agen k8s-ai. Fungsi ini menginisialisasi daftar pesan dengan sistem, mengelola interaksi dengan pengguna, dan mendelegasikan semua tugas berat ke fungsi send(). Kita akan melihat bagaimana cara kerjanya nanti, di bagian *Implementasi dan Pemanggilan Aplikasi*.

```
def main():
...

   if __name__ == "__main__": main()
```

Selanjutnya, mari kita lihat bagaimana cara bekerja dengan alat OpenAI untuk agen kita.

Mendefinisikan Dan Mendaftarkan Alat Dengan OPENAI

Agen k8s-ai kami dirancang untuk berinteraksi dengan klien OpenAI dan menggunakan alat kubectl untuk mengelola klaster Kubernetes. Agen ini perlu mematuhi format API OpenAI untuk mendefinisikan alat. Penting untuk dicatat bahwa ada berbagai jenis alat. Agen k8s-ai menggunakan pemanggilan fungsi, yang merupakan alat bertipe "fungsi". Alat fungsi memiliki nama, deskripsi, parameter, dan daftar parameter yang diperlukan. Anda juga dapat menentukan apakah alat tersebut beroperasi dalam mode ketat. Lihat dokumentasi OpenAI untuk detail lebih lanjut. Berikut adalah definisi lengkap dari daftar alat k8s-ai (hanya berisi alat kubectl):

```
tools = [{
    "type": "function",
    "function": {
        "name": "kubectl",
        "description": "execute a kubectl command against the current k8s
cluster",
        "parameters": {
            "type": "object",
            "properties": {
                "cmd": {
                    "type": "string",
                    "description": (
                        "the kubectl command to execute (without kubectl, just "
                        "the arguments). For example, 'get pods'"
                    ),
                },
            },
            "required": ["cmd"],
        },
    },
}]
```

Mari kita uraikan satu per satu:

1. **Daftar alat**: Variabel tools adalah sebuah daftar, yang memungkinkan Anda untuk mendefinisikan satu atau lebih alat. Setiap item dalam daftar adalah kamus yang mewakili satu alat. Dalam hal ini, kita hanya mendefinisikan satu alat, yaitu fungsi kubectl.

```
tools = [
    {
        ...
    }
]
```

2. **Tipe**: Ini memberi tahu API OpenAI bahwa alat tersebut adalah alat fungsi. Alat fungsi menggunakan pemanggilan fungsi, di mana model dapat secara otomatis memilih dan memanggil alat menggunakan argumen yang dibuatnya. Definisi alat lainnya tersarang di bawah sub-kamus "fungsi".

```
"type": "function"
"function": {

    ...

}
```

3. **Nama:** Ini adalah nama yang akan digunakan model untuk memanggil alat ini. Nama tersebut harus singkat, deskriptif, dan unik dalam konteks semua alat yang Anda daftarkan. Dalam hal ini, namanya adalah kubectl, yang merupakan pilihan yang baik karena ini adalah alat yang terkenal yang ditemui LLM saat pelatihan.

```
"name": "kubectl"
```

4. **Deskripsi**: Deskripsi memberikan konteks tambahan di luar namanya tentang apa yang dilakukan fungsi tersebut. Deskripsi harus jelas dan ringkas, menjelaskan tujuan alat dan cara penggunaannya. Dalam hal ini, deskripsinya cukup mendasar karena LLM seharusnya sudah mengetahui apa itu kubectl.
   Deskripsi fungsi harus berisi semua informasi yang diperlukan agar model (LLM) dapat memutuskan apakah harus memanggil alat tersebut atau tidak dalam situasi tertentu.

```
"description": "execute a kubectl comand against the current k8s cluster"
```

   Ingat diskusi kita tentang AutoGen (bab 2), yang memiliki banyak abstraksi. Ia juga perlu mengkonversi semua informasi dari abstraksi ini ke dalam deskripsi prompt dan alat yang dikirimkannya ke LLM. Secara umum, sistem AI yang dibangun di atas kerangka kerja AI yang menerima respons akhir dari setiap loop agenik dapat memutuskan untuk menjalankan kode kustom tambahan dan tidak hanya menampilkan respons akhir.
5. **Parameter**: Parameter menentukan input yang diharapkan oleh fungsi. Parameter didefinisikan dalam format Skema JSON, di mana tipenya selalu "objek" dan bidang "properti" berisi parameter spesifik yang diterima oleh fungsi. Setiap parameter memiliki nama ("cmd" di sini), tipe ("string" dalam hal ini), dan deskripsi yang menjelaskan untuk apa parameter tersebut. Karena kubectl memiliki banyak sub-perintah, masing-masing dengan parameternya sendiri, kita hanya mendefinisikan satu parameter string untuk mewakili seluruh perintah yang akan dieksekusi. Sekali lagi, kita mengandalkan pelatihan LLM untuk memahami cara membangun perintah kubectl. Parameter cmd juga ditandai sebagai wajib. Perintah kubectl dapat dieksekusi tanpa parameter, tetapi hanya menampilkan bantuan, yang tidak terlalu berguna dalam kasus

ini. Oleh karena itu, kita memerlukan parameter cmd yang disediakan oleh model saat memanggil alat tersebut.

```
"parameters": {
    "type": "object",
    "properties": {
        "cmd": {
            "type": "string",
            "description": (
                "the kubectl command to execute (without kubectl, just "
                "the arguments). For example, 'get pods'"
            ),
        },
    },
    "required": ["cmd"],
}
```

Perhatikan bahwa tidak ada output yang didefinisikan untuk fungsi. Ini berarti bahwa model tidak mengetahui bentuk dan struktur hasil dari panggilan fungsi. Model akan menerima output sebagai string, yang kemudian dapat diuraikan dan dicari tahu cara menggunakannya. Ini adalah pola umum saat menggunakan LLM dengan alat, karena model dapat menangani berbagai macam output dan tidak perlu mengetahui struktur pastinya terlebih dahulu. Meminta pengembang untuk menentukan struktur output dapat menyebabkan banyak gesekan dan dapat menjadi tantangan bagi alat seperti kubectl yang dapat mengembalikan berbagai macam output dan format output.

Mengimplementasikan Loop Eksekusi Pemanggilan Alat

Sekarang, mari kita lihat lebih dekat loop agen dan bagaimana interaksinya dengan API OpenAI untuk memanggil alat kubectl. Ini adalah inti dari agen k8s-ai, di mana ia mengirim pesan ke API OpenAI, menerima respons, dan mengeksekusi perintah kubectl. Yang mengejutkan, loop agen cukup sederhana dan hanya terdiri dari 18 baris kode. Berikut kode lengkapnya:

```
def send(messages: list[dict[str, any]]) -> str:
    response = client.chat.completions.create(
        model=model_name, messages=messages,
        tools=tools, tool_choice="auto")
    r = response.choices[0].message
    if r.tool_calls:
        message = dict(
            role=r.role,
            content=r.content,
            tool_calls=[dict(id=t.id, type=t.type,
                function=dict(name=t.function.name,
                    arguments=t.function.arguments)
            ) for t in r.tool_calls if t.function])
        messages.append(message)
        for t in r.tool_calls:
            if t.function.name == 'kubectl':
                cmd = json.loads(t.function.arguments)['cmd'].split()
                result = sh.kubectl(cmd)
                messages.append(dict(tool_call_id=t.id, role="tool",
```

```
                    name=t.function.name, content=result))
            return send(messages)
        return r.content.strip()
```

Mari kita uraikan langkah demi langkah:

1. **Tanda tangan fungsi**: Baris ini mendefinisikan fungsi loop inti, send(). Fungsi ini menerima daftar kamus pesan sebagai input (sama seperti parameter pesan OpenAI). Fungsi ini mengembalikan konten pesan asisten akhir sebagai string.

```
def send(messages: list[dict[str, any]]) -> str:
```

2. **Memanggil API Penyelesaian Obrolan**: Baris ini memanggil API OpenAI untuk menghasilkan respons berdasarkan pesan yang diberikan. Ini menggunakan metode chat.completions.create() dari API Penyelesaian Obrolan.
   Ia meneruskan nama model tetap kita (gpt-4o), daftar pesan, daftar alat yang telah kita definisikan sebelumnya (dengan kubectl sebagai satu-satunya alat), dan tool_choice yang diatur ke "auto". Pengaturan "auto" memungkinkan model untuk memutuskan apakah akan menggunakan alat atau tidak berdasarkan keadaan percakapan saat ini.

```
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    tools=tools,
    tool_choice="auto"
)
```

3. **Dapatkan pesan respons pertama**: API OpenAI dapat mengembalikan beberapa pilihan dalam satu respons, tetapi dalam praktiknya, biasanya hanya mengembalikan satu. Jadi, k8s-ai hanya menggunakan kasus sederhana dan hanya melihat pilihan pertama lalu mengekstrak objek pesannya.

```
r = response.choices[0].message
```

4. **Periksa apakah respons berisi panggilan alat**: Jika respons berisi panggilan alat, itu berarti model telah memutuskan untuk menggunakan alat untuk melakukan beberapa tindakan.

```
if r.tool_calls:
```

5. **Siapkan pesan panggilan alat**: Dalam hal ini, kita perlu menyiapkan kamus pesan yang berisi peran, konten, dan panggilan alat. Panggilan alat diubah menjadi daftar kamus yang berisi ID panggilan alat (berasal dari LLM), tipe, nama fungsi, dan argumen. Ini diperlukan untuk melacak panggilan alat dan hasilnya dalam riwayat percakapan.

```
    message = dict(
        role=r.role,
```

```
            content=r.content,
            tool_calls=[
                dict(
                    id=t.id,
                    type=t.type,
                    function=dict(
                        name=t.function.name,
                        arguments=t.function.arguments
                    )) for t in r.tool_calls if t.function
```

Perhatikan bahwa mungkin ada beberapa panggilan alat dalam satu respons. Bahkan jika kita hanya memiliki satu alat yang terdaftar, model dapat memutuskan untuk memanggilnya beberapa kali dengan argumen yang berbeda. Misalnya, ia mungkin ingin menjalankan beberapa perintah kubectl, seperti get pods, di namespace yang berbeda.

6. **Tambahkan pesan panggilan alat ke daftar pesan**: Ini cukup mudah. Cukup tambahkan kamus pesan yang baru saja kita buat ke daftar pesan. Ini penting untuk menjaga agar alur percakapan tetap mutakhir.

```
messages.append(message)
```

7. **Memanggil alat kubectl untuk setiap permintaan panggilan alat**: Sekarang, ia mengulangi setiap panggilan alat dalam respons. Jika panggilan alat tersebut untuk fungsi kubectl, ia mengekstrak argumen cmd dan membaginya menjadi daftar argumen baris perintah. Misalnya, "get pods" menjadi ["get", "pods"]. Kemudian, ia menggunakan modul sh untuk mengeksekusi perintah kubectl dengan argumen yang diberikan. Hasil eksekusi perintah disimpan dalam variabel hasil.

```
for t in r.tool_calls:
    if t.function.name == "kubectl":
        cmd = json.loads(t.function.arguments)["cmd"].split()
        result = sh.kubectl(cmd)
```

8. **Menambahkan hasil panggilan alat ke daftar pesan**: Setelah mengeksekusi perintah kubectl, hasilnya ditambahkan ke daftar pesan sebagai pesan baru dengan ID alat asli dari LLM (sehingga LLM mengetahui hasil mana yang sesuai dengan panggilan alat mana), peran yang diatur ke "alat", nama fungsi (akan selalu kubectl dalam hal ini), dan konten yang diatur ke hasil eksekusi perintah. Inilah cara agen menangkap hasil panggilan alat dan membuatnya tersedia untuk model pada iterasi loop berikutnya.

```
messages.append(dict(tool_call_id=t.id, role="tool",
    name=t.function.name, content=result))
```

9. **Rekursi untuk melanjutkan perulangan**: Selama model memiliki panggilan alat dalam responsnya, agen akan terus memprosesnya. Fungsi send() kemudian memanggil

dirinya sendiri secara rekursif dengan daftar pesan yang diperbarui, yang mencakup semua permintaan alat dari model dan semua hasil alat.

```
return send(messages)
```

10. **Mengembalikan respons akhir**: Pada akhirnya, ketika tidak ada lagi panggilan alat dalam respons, fungsi send() mengembalikan isi respons akhir dari model.

```
return r.content.strip()
```

Perhatikan bahwa kami tidak mengembalikan seluruh objek respons atau seluruh daftar pesan, tetapi hanya konten pesan terakhir. Beberapa kerangka kerja AI mungkin menyediakan kait untuk mengakses riwayat pesan lengkap, tetapi sekali lagi, k8s-ai dirancang agar sederhana.

Sekarang mari kita berinteraksi dengan agen ini.

Menjalankan Dan Berinteraksi Dengan Agen Obrolan Kubernetes

Terakhir, namun tidak kalah pentingnya, k8s-ai menyediakan **antarmuka baris perintah** (CLI) interaktif untuk berinteraksi dengan agen.

1. Fungsi main() pertama-tama mencetak pesan selamat datang dan menginisialisasi daftar pesan dengan pesan sistem yang menetapkan konteks untuk agen.

```
def main():
    print("Interactive Kubernetes Chat. Type 'exit' to quit.\n" + "-" *
52)

    messages = [{'role': 'system', 'content': 'You are a Kubernetes expert
ready to help'
    }]
```

2. Kemudian, sistem memulai perulangan interaktif, menunggu pesan dari pengguna, mengirimkannya ke agen, yang menjalankan perulangan agen hingga mendapatkan respons akhir, mencetaknya ke layar, dan mengulangi proses tersebut. Perulangan interaktif berlanjut hingga pengguna mengetik 'exit'.

```
while (user_input := input("ĿĠYou: ")).lower() != 'exit':
    messages.append(dict(role="user", content=user_input))
    response = send(messages)
    print(f" AI: {response}\n----------")
```

Agen yang lebih canggih mungkin menggunakan sistem manajemen memori yang lebih kompleks untuk menyimpan dan mengambil pesan.

Anda sekarang telah mengalami agen AI lengkap dari kedua perspektif: sebagai pengguna yang berinteraksi dengannya untuk memecahkan masalah nyata, dan sebagai pengembang yang memahami mekanisme internalnya baris demi baris. Pemahaman langsung tentang bagaimana pesan mengalir melalui loop agen, bagaimana alat didefinisikan dan dipanggil, dan bagaimana kode sederhana dapat

menciptakan perilaku otonom yang ampuh membentuk dasar untuk membangun agen Anda sendiri. Mari kita rangkum wawasan ini dalam ringkasan.

3.5 RINGKASAN

Dalam bab ini, kita melampaui teori dan melihat agen AI beraksi, mengendalikan klaster Kubernetes nyata dengan perpaduan otonomi, penalaran, dan alat bantu. Agen k8s-ai, yang dibangun dalam 61 baris Python (termasuk baris kosong dan impor), sangat mumpuni. Terlepas dari kesederhanaannya, ia menunjukkan beberapa kemampuan dasar agen AI.

Sistem ini mampu memeriksa keadaan klaster menggunakan kubectl, memungkinkannya untuk mengamati kondisi saat ini secara efektif. Ia memanfaatkan API OpenAI untuk mendiagnosis masalah dan mengidentifikasi akar penyebabnya. Ia memahami kapan dan bagaimana menggunakan kubectl, tidak hanya untuk mengumpulkan informasi tetapi juga untuk melakukan perubahan yang diperlukan. Sistem ini beroperasi dengan pendekatan human-in-the-loop, meminta izin sebelum melakukan modifikasi apa pun kecuali secara eksplisit diinstruksikan untuk bertindak secara otonom. Sepanjang operasinya, ia mempertahankan konteks percakapan, memanggil alat bantu sesuai kebutuhan, dan memberikan kembali output yang dihasilkan ke dalam model bahasa, membentuk lingkaran agen yang koheren.

Kami membuat penyebaran yang rusak, membingungkan penjadwal, dan bahkan menyabotase citra kontainer di klaster KinD. Agen k8s-ai menangani semuanya. Kami melihat bagaimana antarmuka pemanggilan fungsi OpenAI, yang dipasangkan dengan satu alat yang terdefinisi dengan baik dan beberapa baris kode penghubung, dapat menghidupkan sistem AI berbasis agen yang didukung LLM, termasuk antarmuka obrolan interaktif.

Secara lebih luas, bab ini menunjukkan bahwa agen AI tidak harus membengkak, rapuh, atau terlalu direkayasa. Dengan pemahaman yang jelas tentang cara kerjanya – pesan, alat, memori, dan loop agen – Anda dapat membangun sistem yang berguna dan andal yang meningkatkan alur kerja manusia atau bahkan berjalan secara otonom di lingkungan yang terkontrol.

Pada bab berikutnya, kita akan memulai perjalanan untuk membangun kerangka kerja AI berbasis agen yang lengkap berdasarkan prinsip yang sama dengan k8s-ai.

BAB 4
MEMBANGUN KERANGKA KERJA AI AGEN BERBASIS ALAT

4.1 PENDAHULUAN

Pada bab sebelumnya, kita telah melampaui teori dan mengamati agen AI dalam aksi, khususnya mengendalikan klaster Kubernetes nyata dengan campuran otonomi, penalaran, dan alat. Melalui **agen k8s-ai** yang ringkas namun ampuh, kita telah mendemonstrasikan kemampuan dasar agen AI seperti observasi, diagnosis, penggunaan alat, mempertahankan interaksi manusia-dalam-lingkaran, dan menggunakan lingkaran agen. Agen ini, yang dikembangkan hanya dalam 61 baris Python, merupakan demonstrasi yang mengesankan tentang kekuatan agen AI dan LLM.

Pada bab ini, kita akan mulai mengembangkan kerangka kerja AI yang lebih kompleks dan lengkap yang disebut AI-6. AI-6 akan memungkinkan kita untuk menjelajahi spektrum penuh kemampuan agen AI, seperti manajemen memori yang canggih, fasilitas manajemen alat tingkat lanjut, dan dukungan untuk beberapa penyedia LLM. AI-6 sangat netral dan sangat mudah diperluas melalui alat-alat. Prinsip desain utamanya adalah menjaganya tetap sederhana dengan menyediakan sejumlah kecil fitur inti yang memungkinkan pembangunan sistem AI yang lebih canggih di atasnya dengan menambahkan alat-alat khusus. AI-6 adalah kerangka kerja Python yang memiliki backend dan frontend. Dalam bab ini, kita akan fokus pada backend. Pada *bab 5*, kita akan menambahkan alat-alat ke AI-6. Pada *bab 6*, kita akan fokus pada frontend.

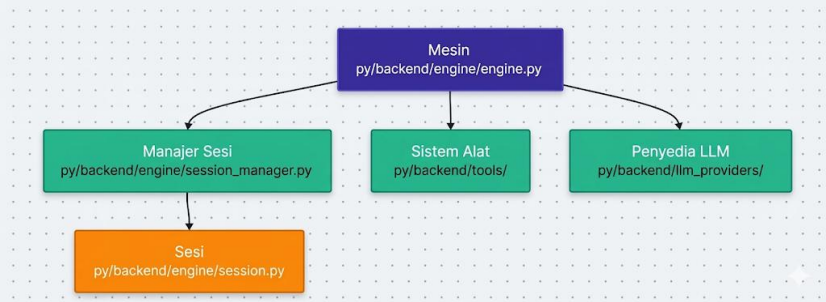
Kita akan mulai dengan gambaran umum arsitektur backend kerangka kerja dan komponen-komponennya, kemudian kita akan menyelami lebih dalam beberapa aspek penting dari kerangka kerja, seperti manajemen alat, manajemen memori, dan alur pemrosesan pesan. Terakhir, kita akan mengeksplorasi cara memperluas kerangka kerja dengan alat-alat baru dan penyedia LLM. Dalam bab ini, kita akan membahas topik-topik utama berikut:

- Arsitektur dan komponen backend kerangka kerja AI-6
- Arsitektur sistem alat
- Manajemen memori

4.2 ARSITEKTUR DAN KOMPONEN BACKEND KERANGKA KERJA AI-6

Backend AI-6 didukung oleh **mesin inti** yang menjalankan loop agen, mengelola alat, mendukung penyedia LLM apa pun dengan menawarkan antarmuka penyedia LLM generik, dan memiliki sistem manajemen memori. Manajer sesi bertanggung jawab untuk mengelola sesi (riwayat pesan) dengan setiap pengguna. Ia menggunakan kelas Session untuk menyimpan seluruh isi percakapan tunggal. Sistem alat merupakan subsistem yang dapat diperluas untuk mendefinisikan alat dan menyediakan kelas dasar yang berguna untuk implementasi alat. Penyedia LLM adalah subsistem lain yang dapat diperluas yang

mendefinisikan antarmuka penyedia LLM generik dan menyediakan beberapa implementasi (OpenAI dan Ollama saat ini).



Gambar 4.1 Mesin Inti AI-G

Mari kita lihat bagaimana mesin inti bekerja.

Mesin Inti

Mesin inti adalah jantung dari kerangka kerja AI-6. Ia mengoordinasikan semua interaksi dengan penyedia LLM, alat, dan manajemen sesi. Ia menyediakan antarmuka terpadu untuk berbagai frontend untuk berkomunikasi dengan model AI sambil mempertahankan status percakapan dan mengeksekusi panggilan alat. Ia diimplementasikan dalam satu kelas bernama Engine.

Menggunakan mesin ini cukup sederhana. Pertama, Anda menginisialisasinya dengan konfigurasi yang diinginkan, lalu Anda menjalankannya untuk mengeksekusi loop agen. Mari kita lihat cara kerjanya.

Inisialisasi

Mesin, yang diinisialisasi dengan penyedia LLM, alat, dan pengaturan konfigurasi, berfungsi sebagai komponen inti ketika sistem AI agen menggunakan AI-6. Tanda tangannya cukup sederhana; ia hanya mengambil instance dari kelas Config yang berisi semua pengaturan yang diperlukan:

```
def __init__(self, config: Config):
...
```

Diagram berikut mengilustrasikan proses ini:

Gambar 4.2 Sistem AI Menginisialisasi Mesin Inti Dengan Kelas Konfigurasi

Sistem AI yang menggunakan kerangka kerja AI-6 menentukan konfigurasi untuk mesin inti menggunakan objek Config. Mari kita lihat kelas Config:

```
@dataclass
class Config:
    default_model_id: str
    tools_dir: str
    mcp_tools_dir: str
    memory_dir: str
    session_id: Optional[str] = None
    checkpoint_interval: int = 3
    summary_threshold_ratio: float = 0.8
    tool_config: Mapping[str, dict] = field(
        default_factory=lambda: MappingProxyType({})
    provider_config: Mapping[str, dict] = field(
        default_factory=lambda: MappingProxyType({})
```

Ada banyak hal yang perlu diuraikan di sini. Mari kita lihat apa saja komponen kelas Config:

- default_model_id adalah ID model yang akan digunakan secara default ketika tidak ada model spesifik yang diminta, yang juga menentukan penyedia LLM default karena setiap
- model hanya dimiliki oleh satu penyedia LLM
- tools_dir adalah direktori tempat alat AI-6 kustom berada, dan mcp_tools_dir adalah direktori tempat alat MCP berada
- memory_dir adalah direktori tempat file memori disimpan
- session_id adalah pengidentifikasi opsional untuk sesi, dan dapat digunakan untuk melanjutkan sesi dari eksekusi sebelumnya
- checkpoint_interval adalah jumlah iterasi setelah checkpoint akan disimpan
- summary_threshold_ratio adalah rasio token yang akan memicu ringkasan riwayat percakapan
- Terakhir, tool_config dan provider_config adalah pemetaan yang memungkinkan Anda untuk menentukan konfigurasi tambahan untuk setiap alat dan penyedia

Jangan khawatir jika ini agak membingungkan; Kami akan menjelaskan tujuan dari setiap item konfigurasi saat kita melanjutkan pembahasan di bab ini. Mari kita lihat bagaimana mesin diinisialisasi dengan objek Config ini. Pertama, kita hanya menyimpan default_model_id sebagai atribut kelas. Tidak ada yang rumit di sini:

```
class Engine:
    def __init__(self, config: Config):
        self.default_model_id = config.default_model_id
```

Kemudian, kita mendapatkan ukuran jendela konteks dari modul model_info berdasarkan ID model default. Kita menggunakannya untuk menghitung ambang batas token, yaitu jumlah token yang akan memicu ringkasan riwayat percakapan. Ini diperlukan karena ketika riwayat percakapan mendekati ukuran jendela konteks model, kita perlu meringkasnya untuk menghindari melebihi batas keras ini.

```
# Get the context window size from model_info based on the default model
context_window_size = get_context_window_size(
    self.default_model_id)
self.token_threshold = int(
```

```
    context_window_size * config.summary_threshold_ratio
)
```

Selanjutnya, kita menginisialisasi penyedia LLM. Mesin ini mendukung beberapa penyedia LLM, yang dapat ditemukan menggunakan metode discover_llm_providers(). Alasan mengapa hal ini tidak dikonfigurasi melalui objek Config adalah karena menambahkan penyedia LLM baru bukanlah operasi umum bagi pengguna. Namun, ini jelas dapat menjadi pendekatan desain alternatif.

Setelah direktori penyedia ditemukan dan diverifikasi sebagai direktori, metode Engine.discover_llm_providers() akan dipanggil. Kita akan menjelajahinya di bagian selanjutnya. Metode ini mengembalikan daftar penyedia LLM, di mana setiap penyedia dapat mendukung beberapa model.

```
# Find LLM providers directory
        llm_providers_dir = os.path.join(
            os.path.dirname(os.path.dirname(__file__)), "llm_providers"
        )
        assert os.path.isdir(llm_providers_dir), (
            f"LLM providers directory not found: {llm_providers_dir}"
        )

        # Discover available LLM providers
        self.llm_providers = Engine.discover_llm_providers(
            llm_providers_dir, config.provider_config
        )
```

Mesin tersebut menggunakan daftar penyedia untuk membuat pemetaan ID model ke penyedia LLM masing-masing:

```
self.model_provider_map = {
            model_id: llm_provider
            for llm_provider in self.llm_providers
            for model_id in llm_provider.models
        }
```

Selanjutnya, kita menginisialisasi alat-alat tersebut. Mesin ini mendukung alat kustom dan alat **Model Context Protocol** (MCP). Proses inisialisasi untuk alat-alat tersebut mirip dengan penyedia LLM. Mesin memanggil metode penemuan (direktori berasal dari objek Config) untuk menemukan dan mendaftarkan alat-alat tersebut. Kemudian, ia membuat kamus yang memetakan nama alat ke objek alat masing-masing. Ini memungkinkan mesin untuk dengan mudah mengakses dan memanggil alat berdasarkan namanya selama loop agen. Perhatikan bahwa alat MCP belum digunakan saat ini. Kami akan memasukkan alat MCP di *bab 7*.

```
# Discover available tools
tool_list = Engine.discover_tools(
    config.tools_dir, config.tool_config)
```

```
# Discover MCP tools
mcp_tool_list = Engine.discover_mcp_tools(config.mcp_tools_dir)
self.tool_dict = {t.name: t for t in tool_list}
```

Bagian selanjutnya dari alur kerja inisialisasi berkaitan dengan manajemen memori. Mesin menggunakan pengelola sesi untuk menangani sesi dan objek sesi untuk mengelola riwayat percakapan. Kemudian, ia menetapkan beberapa atribut yang berkaitan dengan penyimpanan titik pemeriksaan berkala.

```
# Initialize session and session manager
self.session_manager = SessionManager(config.memory_dir)
self.session = Session(config.memory_dir)

# Session-related attributes
self.checkpoint_interval = config.checkpoint_interval
self.message_count_since_checkpoint = 0
```

Selanjutnya, kita mengatur sesi. Penyimpul bertanggung jawab untuk meringkas riwayat percakapan ketika terlalu panjang dan mendekati ukuran jendela konteks model. Kemudian, kita mendaftarkan alat memori ke mesin, yang akan memberi pengguna opsi untuk mengelola memori secara langsung.

```
# Instantiate the summarizer with the first LLM provider
self.summarizer = SessionSummarizer(self.llm_providers[0])

# Register memory tools with the engine
self._register_memory_tools()
```

Terakhir, jika konfigurasi menyertakan ID sesi, mesin akan mencoba memuat sesi ini jika ada. Hal ini memungkinkan untuk melanjutkan sesi sebelumnya dan melanjutkan percakapan dari titik terakhir.

```
# Load previous session if session_id is provided and exists
if config.session_id:
    available_sessions = self.session_manager.list_sessions()
    if config.session_id in available_sessions:
        # Create a new session object
        self.session = Session(config.memory_dir)
        # Load from disk
        self.session.load(config.session_id)
```

Singkatnya, kita telah menginisialisasi mesin dengan konfigurasi dan secara dinamis memuat penyedia LLM, alat kustom, dan alat MCP. Kemudian, kita menginisialisasi sistem memori. Mari kita lihat apa yang terjadi ketika kita menjalankan mesin.

Menjalankan Mesin

Setelah mesin diinisialisasi, kita dapat menjalankannya untuk memulai sesi. Metode run() menerima beberapa callable (fungsi) sebagai argumen. Fungsi `get_input_func()` mengambil input pengguna, `on_tool_call_func()` adalah callback yang dipanggil ketika sebuah alat dipanggil, dan `on_response_func()` adalah callback yang dipanggil ketika

respons akhir dari loop agen dihasilkan. Callback ini diimplementasikan oleh sistem AI agen yang menggunakan kerangka kerja AI-6 atau frontend bawaan AI-6 itu sendiri. Mereka memungkinkan mesin untuk berkomunikasi melalui antarmuka generik tanpa mengetahui apa pun tentang aplikasi atau frontend spesifik yang digunakan. Desain ini memungkinkan fleksibilitas dan ekstensibilitas, karena sistem AI agenik dan frontend yang berbeda dapat mengimplementasikan versi callback mereka sendiri.

Metode run() akan berulang hingga fungsi `get_input_func()` berhenti memberikan input. Ini berarti bahwa mesin melakukan beberapa loop agenik (masing-masing terdiri dari satu atau lebih interaksi dengan LLM dan panggilan alat) dan mengakumulasikan riwayat percakapan dalam sesi.

```
def run(
        self,
        get_input_func: Callable[[], None],
        on_tool_call_func: Callable[[str, dict, str], None] | None,
        on_response_func: Callable[[str], None],
):
```

Implementasinya cukup sederhana. Dimulai dengan mendapatkan input awal untuk loop agenik saat ini dan menambahkannya ke sesi. Kemudian, ia juga membuat checkpoint jika diperlukan. Kita akan membahas checkpoint di bagian *Managemen penyimpanan*.

```
try:
    while user_input := get_input_func():
        message = UserMessage(content=user_input)
        self.session.add_message(message)
        self._checkpoint_if_needed()
```

Bagian selanjutnya adalah inti dari perulangan agenik. Bagian ini memanggil `metode _send()`, yang bertanggung jawab untuk memproses input, dengan ID model default dan meneruskan callback `on_tool_call_func`. `Metode_send()` menangani semua pekerjaan berat, seperti yang akan kita lihat sebentar lagi. Respons kemudian ditambahkan ke sesi sebagai pesan asisten, dan checkpoint lain dibuat jika diperlukan. Terakhir, callback on_response_func dipanggil dengan respons tersebut.

```
response = self._send(
    self.default_model_id, on_tool_call_func
message = AssistantMessage(content=response)
self.session.add_message(message)
self._checkpoint_if_needed()
on_response_func(response)
```

Terakhir, kita memiliki klausa finally, yang memastikan sesi disimpan ketika metode run() keluar, terlepas dari apakah berhasil atau menimbulkan pengecualian. Ini penting untuk memastikan bahwa riwayat percakapan tetap terjaga bahkan ketika terjadi kesalahan (misalnya, kegagalan karena melebihi batas laju penyedia LLM).

```
finally:
    # Save the session when we're done
    self.session.save()
```

Menjalankan Loop Agenik

Mari kita lihat lebih dekat metode `_send()`, yang menjalankan loop agenik lengkap setiap kali dipanggil sebagai bagian dari sesi. Metode _send() mengambil model_id dan callback on_tool_call_func opsional sebagai argumen. model_id menentukan model mana (dan secara ekstensi, penyedia LLM mana) yang akan digunakan untuk loop agenik ini, dan on_tool_call_func adalah callback yang dipanggil setiap kali alat dipanggil selama loop.

Hal ini memungkinkan mesin untuk memberi tahu frontend atau aplikasi tentang panggilan alat, sehingga memungkinkan pembaruan dan interaksi secara real-time. Ia mengembalikan respons akhir dari loop agen, yaitu respons yang dihasilkan oleh LLM setelah memproses input dan mengeksekusi sebanyak mungkin panggilan alat yang diperlukan.

```
def _send(
        self,
        model_id: str,
        on_tool_call_func: Callable[
            [str, dict, str], None
        ] | None = None,
) -> str:
```

Langkah pertama dalam metode `_send()` adalah memeriksa apakah `model_id` valid dan apakah penyedia LLM yang sesuai tersedia. Jika tidak, ini merupakan kegagalan kritis, dan metode tersebut akan menimbulkan RuntimeError.

```
llm_provider = self.model_provider_map.get(model_id)
if llm_provider is None:
    raise RuntimeError(f"Unknown model ID: {model_id}")
```

Langkah selanjutnya adalah mengirimkan semua pesan sesi dan kamus alat dengan ID model yang dipilih kepada penyedia LLM terpilih. Penyedia LLM akan menangani komunikasi aktual dengan LLM dan mengembalikan responsnya. Bagian ini bersifat agnostik terhadap penyedia, karena AI-6 menggunakan abstraksi untuk penyedia LLM. Kita akan memeriksanya secara mendetail di bagian selanjutnya, *Abstraksi penyedia LLM.*

```
try:
    response = llm_provider.send(
        self.session.messages, self.tool_dict, model_id
    )
except Exception as e:
    raise RuntimeError(f"Error sending message to LLM: {e}")
```

Jika respons berisi panggilan alat, kita perlu memprosesnya. Langkah pertama adalah memetakan ID panggilan alat ke **Pengidentifikasi Unik Universal** atau **UUID**. Ini diperlukan karena beberapa penyedia LLM mungkin mengembalikan ID panggilan alat yang bukan UUID yang valid, dan kita ingin memastikan bahwa semua ID panggilan alat adalah UUID yang valid,

yang merupakan cara kita yang tidak bergantung pada penyedia untuk mengidentifikasi panggilan alat. Fungsi `generate_tool_call_id()` digunakan untuk menghasilkan UUID baru untuk setiap ID panggilan alat yang belum merupakan UUID yang valid. Kamus id_mapping memetakan panggilan alat asli yang diterima dari LLM ke UUID baru yang dihasilkan oleh mesin jika diperlukan. Jika ID alat yang diberikan LLM dianggap valid, ID tersebut digunakan apa adanya, dan tidak ada UUID baru yang dihasilkan.

```
if response.tool_calls:
    # Create a mapping of original IDs to new UUIDs if needed
    id_mapping = {}
    for tool_call in response.tool_calls:
        # Check if we need to replace the ID with a UUID
        if (
                not tool_call.id or len(tool_call.id) < 32
        ): # Simple check for non-UUID
           new_id = generate_tool_call_id(tool_call.id)
            id_mapping[tool_call.id] = new_id
```

Selanjutnya, kita memperbarui ID panggilan alat dalam respons jika ID tersebut diganti dengan UUID baru. Karena penyedia LLM sudah mengembalikan objek AssistantMessage, kita dapat menggunakannya secara langsung untuk memperbarui panggilan alat.

```
# Update the tool call IDs in the response if needed
if id_mapping and response.tool_calls:
    for tool_call in response.tool_calls:
        if tool_call.id in id_mapping:
            tool_call.id = id_mapping[tool_call.id]
```

Sekarang, pesan asisten dapat ditambahkan ke sesi.

```
# Add the assistant message with updated tool_calls
self.session.add_message(response)
```

Sebelum kita melanjutkan pemrosesan panggilan alat, kita perlu melacak ID panggilan alat dari pesan asisten dan menyimpannya dalam sebuah set yang akan kita gunakan nanti.

```
# Track tool_call_ids from this assistant message
tool_call_ids = set()

for tool_call in response.tool_calls:
    tool_call_ids.add(tool_call.id)
```

Pada tahap ini, kita dapat mengulangi panggilan alat dalam respons dan memproses masing-masing panggilan. Ada beberapa langkah pendahuluan untuk setiap panggilan alat sebelum kita dapat mengeksekusinya, yang meliputi verifikasi apakah alat yang diminta ada dalam kamus alat, mengurai argumen, dan memeriksa apakah ID panggilan alat perlu diperbarui.

```
# Now process each tool call and add the corresponding tool messages
        for i, tool_call in enumerate(response.tool_calls):
            tool = self.tool_dict.get(tool_call.name)
            if tool is None:
               raise RuntimeError(f"Unknown tool: {tool_call.name}")

        try:
           kwargs = json.loads(tool_call.arguments)
        except json.JSONDecodeError as e:
            raise RuntimeError(
                f"Invalid arguments JSON for tool '{tool_call.name}'"
        )
    # Get the potentially updated tool call ID
    tool_call_id = tool_call.id
    if tool_call.id in id_mapping:
        tool_call_id = id_mapping[tool_call.id]
```

Pada tahap ini, kita dapat mengeksekusi panggilan alat. Alat dijalankan dengan argumen yang telah diuraikan, dan hasilnya ditangkap. Kemudian, kita menghasilkan pesan alat yang berisi hasil panggilan alat, nama alat, dan ID panggilan alat, dan memanggil `callback on_tool_call_func` jika tersedia. Semua ini dilakukan dalam blok try:

```
                    try:
                        # Execute the tool without passing any ID information
                        tool_result = tool.run(**kwargs)

                        # Create the tool message with the Engine managing the
tool_call_id
                        tool_message = ToolMessage(
                            content=str(tool_result),
                            name=tool_call.name,
                            tool_call_id=tool_call_id,
                        )

                        if on_tool_call_func is not None:
                            on_tool_call_func(
                                tool_call.name, kwargs, str(tool_result))
```

Jika panggilan alat gagal, kita menangkap pengecualian dan menghasilkan pesan kesalahan sebagai gantinya:

```
except Exception as e:
        tool_message = ToolMessage(
            content=str(e),
            name=tool_call.name,
            tool_call_id=tool_call_id,
        )
```

Pesan tersebut, yang sudah dalam format yang tidak bergantung pada penyedia, ditambahkan ke sesi, baik itu berhasil atau gagal. Kegagalan panggilan alat tidak menghentikan perulangan agen; kegagalan tersebut hanya dicatat sebagai pesan alat dengan kesalahan dalam sesi. Hal ini memungkinkan agen untuk terus memproses panggilan alat lain atau

menghasilkan respons tanpa gangguan. LLM akan memutuskan apa yang harus dilakukan terhadap panggilan alat yang gagal pada iterasi berikutnya dari perulangan agen.

```
# Add the tool message (ID is guaranteed to be valid since we manage it)
                self.session.add_message(tool_message)
```

Setelah semua panggilan alat diproses dan hasilnya ditambahkan ke sesi, metode _send() memanggil dirinya sendiri secara rekursif. Di sinilah loop agen berlanjut.

```
# Continue the session with another send
return self._send(model_id, on_tool_call_func)
```

Terakhir, ketika respons dari LLM tidak berisi panggilan alat apa pun, metode ini mengembalikan konten respons akhir:

```
return response.content.strip()
```

Baiklah. Mari kita tarik napas sejenak. Itu tadi banyak informasi yang tidak sepele. Sebagian besar kompleksitas berasal dari kebutuhan untuk menerjemahkan antara panggilan alat khusus penyedia dan format yang tidak bergantung pada penyedia yang digunakan mesin untuk mengelola sesi dan riwayat percakapan. Inti dari perulangan agen persis sama dengan perulangan agen k8s-ai. Oke, Sekarang mari kita selami lebih dalam abstraksi penyedia LLM.

Abstraksi Penyedia LLM

Ini adalah bagian yang sangat penting dari kerangka kerja AI-6. Abstraksi penyedia LLM memungkinkan mesin untuk berkomunikasi dengan semua jenis penyedia LLM melalui antarmuka terpadu. Perhatikan bahwa ada desain alternatif di mana kerangka kerja hanya mendukung API OpenAI. Pertama, banyak penyedia LLM mendukung API OpenAI secara langsung atau melalui lapisan kompatibilitas mereka sendiri (misalnya, Google Gemini).

Kedua, ada proyek seperti OpenRouter yang beroperasi seperti proxy ke penyedia LLM apa pun melalui API OpenAI. Jadi, mengapa membangun antarmuka yang tidak bergantung pada penyedia? Nah, ini memberi kerangka kerja kendali penuh atas interaksi dengan penyedia LLM yang didukung, dan tidak memerlukan perantara, yang menambah latensi, biaya, dan masalah keamanan. Mari kita lihat bagaimana abstraksi penyedia LLM diimplementasikan dalam kerangka kerja AI-6.

Antarmuka LLM Provider

Kelas dasar abstrak (ABC) Python adalah wahana yang sempurna untuk mendefinisikan antarmuka LLMProvider, yang didefinisikan dalam file `llm_provider.py`. LLMProvider membuat kerangka kerja AI-6 dapat diperluas dengan memungkinkan pengembang untuk mengimplementasikan penyedia LLM mereka sendiri yang sesuai dengan antarmuka yang cukup sederhana ini. Mari kita uraikan metode demi metode, tetapi pertama-tama mari kita lihat impornya.

Pertama, kita mengimpor kelas dan dekorator yang diperlukan dari modul pustaka standar abc dan typing. Kemudian, kita mengimpor kelas Response dari modul object_model, yang digunakan untuk merepresentasikan respons generik dari LLM.

Terakhir, kita mengimpor kelas Tool dari modul tool, yang merepresentasikan alat yang dapat digunakan oleh LLM. Ini adalah bagian dari arsitektur backend kerangka kerja AI-6.

```
from abc import ABC, abstractmethod
from typing import Iterator, Callable, Any

from backend.engine.object_model import Response
from backend.engine.tool import Tool
```

Mari kita lanjutkan ke kelas LLMProvider itu sendiri dan metode pertama: send(). Metode send() adalah metode abstrak (dihiasi dengan @abstractmethod dan harus diimplementasikan oleh subkelas) yang mengirimkan daftar pesan ke LLM dan menerima respons.

Metode ini juga menyediakan kamus alat dan parameter model opsional kepada LLM. Metode ini mengembalikan objek Response, yang berisi respons LLM sebagai respons generik.

```
class LLMProvider(ABC):
    @abstractmethod
    def send(
        self, messages: list[Message], tool_dict: dict[str, Tool],
        model: str | None = None
    ) -> AssistantMessage:
        """
        Send a message to the LLM and receive a response.
        :param messages: The list of messages to send.
        :param tool_dict: A dictionary of tools available for the LLM to use.
        :param model: The model to use (optional).
        :return: The response from the LLM.
        """
        pass
```

Metode `stream` mirip dengan metode `send()`, tetapi memungkinkan pengiriman respons secara streaming dari LLM saat respons tersebut dihasilkan. Metode ini mengembalikan iterator objek `Response`, yang memungkinkan pemanggil untuk menerima potongan respons sebelum seluruh respons selesai. Ini berguna untuk aplikasi real-time di mana Anda ingin menampilkan respons saat sedang dihasilkan. Perhatikan bahwa metode `stream()` bukanlah metode abstrak dan memiliki implementasi default yang hanya memanggil metode `send()` dan menghasilkan respons lengkap. Ini berarti bahwa subkelas dapat menimpa metode ini untuk menyediakan implementasi streaming yang sebenarnya jika mereka mendukungnya, atau mereka dapat menggunakan implementasi default yang mengembalikan respons lengkap sekaligus.

```
def stream(
    self, messages: list[Message], tool_dict: dict[str, Tool],
    model: str | None = None
```

```
) -> Iterator[AssistantMessage]:
        """
        Stream a message to the LLM and receive responses as they are generated.
        :param messages: The list of messages to send.
        :param tool_dict: The tools available for the LLM to use.
        :param model: The model to use (optional).
        :return: An iterator of responses from the LLM.
        """
        # Default implementation just returns the full response
        yield self.send(messages, tool_dict, model)
```

Properti model adalah properti abstrak yang harus diimplementasikan oleh subkelas. Properti ini mengembalikan daftar ID model sebagai string. Cukup sederhana. ID model adalah pengidentifikasi unik untuk setiap model yang didukung oleh penyedia LLM. ID ini digunakan oleh mesin untuk memetakan model ke penyedia dan untuk memilih penyedia yang tepat ketika permintaan untuk model tertentu dikirim.

```
@property
    @abstractmethod
    def models(self) -> list[str]:
        """ Get the list of available models."""
        pass
```

Setelah membahas antarmuka LLMProvider, mari kita lihat penyedia LLM konkret yang disertakan dengan AI-6 secara bawaan.

Penyedia Yang Didukung

Mari kita lihat penyedia LLM yang didukung oleh kerangka kerja AI-6 saat ini. AI-6 dilengkapi dengan penyedia LLM untuk OpenAI dan Ollama, menawarkan opsi model berbasis cloud dan lokal. Ingat bahwa penyedia LLM tambahan dapat dengan mudah diimplementasikan. Mari kita lihat terlebih dahulu implementasi penyedia OpenAI.

Penyedia OpenAI

Penyedia OpenAI diimplementasikan dalam file openai_provider.py. Ia menggunakan OpenAI Python SDK untuk berkomunikasi dengan API OpenAI. Penyedia mengimplementasikan antarmuka LLMProvider dan menyediakan metode yang diperlukan untuk mengirim pesan, melakukan streaming pesan, dan mendapatkan model. Mari kita lihat penyedia OpenAI.

Penyedia ini memerlukan kunci API untuk melakukan otentikasi dengan API OpenAI, yang diteruskan ke konstruktor. Ia memiliki nilai default untuk URL dasar API dan model. Ketika URL dasar adalah None, SDK OpenAI menggunakan endpoint API OpenAI resmi: https://api.openai.com/v1. Jika Anda ingin menggunakan penyedia LLM lain yang memiliki API yang kompatibel dengan OpenAI seperti Anyscale, DeepSeek, SambaNova, atau Cerebras, maka Anda perlu meneruskan endpoint yang benar.

Dalam hal ini, penyedia OpenAI sudah mendukung beberapa penyedia dan tentu saja dapat digunakan dengan OpenRouter juga.

```
from typing import Iterator
from backend.object_model import (
    LLMProvider, ToolCall, Usage, Tool, AssistantMessage
)
from openai import OpenAI


class OpenAIProvider(LLMProvider):
    def __init__(
        self, api_key: str, base_url = None,
        default_model: str = "gpt-4o"
    ):
        self.default_model = default_model
        self.client = OpenAI(base_url=base_url, api_key=api_key)
```

Antarmuka LLMProvider didefinisikan dalam hal struktur data AI-6, yang tidak bergantung pada penyedia LLM tetapi berisi informasi yang sama yang dibutuhkan oleh LLM untuk memproses pesan pengguna dan meminta panggilan alat. Jadi, sebenarnya, tugas penyedia OpenAI adalah menerjemahkan dari representasi AI-6 ke representasi SDK Python OpenAI.

Berikut adalah metode send() yang melakukan hal tersebut. Misalnya, SDK Python OpenAI mengharapkan alat-alat tersebut didefinisikan sebagai kamus Python, sehingga metode `send()` menggunakan `_tool2dict()` privat untuk mengkonversi `tool_dict` menjadi daftar kamus yang dapat dipahami oleh SDK OpenAI.

```
def send(
        self, messages: list[Message], tool_dict: dict[str, Tool],
        model: str | None = None
    ) -> AssistantMessage:
        """
        Send a message to the OpenAI LLM and receive a response.
        :param tool_dict: The tools available for the LLM to use.
        :param messages: The list of messages to send.
        :param model: The model to use (optional).
        :return: The response from the LLM.
        """
        if not messages:
            raise ArgumentError(
                "messages",
                "At least one message is required to send to the LLM."
            )

        if model is None:
            model = self.default_model

        tool_data = [self._tool2dict(tool) for tool in tool_dict.values()]
```

Kemudian, metode ini menggunakan OpenAI Python SDK untuk membuat penyelesaian obrolan dengan model, pesan, dan alat yang telah ditentukan:

```
        response = self.client.chat.completions.create(
            model=model,
            messages=messages,
```

```
            tools=tool_data,
            tool_choice="auto"
        )
```

Selanjutnya, metode ini mengekstrak kolom tool_calls dari respons, dan jika nilainya None, maka akan dikonversi menjadi daftar kosong:

```
tool_calls = response.choices[0].message.tool_calls
tool_calls = [] if tool_calls is None else tool_calls
```

Selain itu, ia juga mengekstrak data penggunaan dari respons, yang mencakup jumlah token input dan output yang digunakan dalam permintaan:

```
# Extract usage data
input_tokens = response.usage.prompt_tokens
output_tokens = response.usage.completion_tokens
```

Terakhir, ia membuat objek AssistantMessage dengan konten respons, peran, panggilan alat, dan data penggunaan. Panggilan alat OpenAI dikonversi ke format AI-6 ToolCall, yang mencakup ID panggilan alat, nama, argumen, dan apakah alat tersebut diperlukan. Bidang yang diperlukan ditentukan dengan memeriksa apakah parameter alat tersebut diperlukan dalam spesifikasi alat. Misalnya, katakanlah alat tersebut adalah `get_weather_in_city` dan memiliki dua parameter: kota, yang diperlukan, dan satuan, yang tidak diperlukan. Jika satuan tidak diberikan, maka alat tersebut akan menganggap satuannya adalah Celsius.

Dalam hal ini, LLM dapat memutuskan untuk memanggil alat tersebut dengan dua parameter, kota: London dan satuan: Fahrenheit, atau hanya dengan parameter yang dibutuhkan, kota: London.

```
return AssistantMessage(
    content=response.choices[0].message.content,
    tool_calls=[
        ToolCall(
            id=tool_call.id,
            name=tool_call.function.name,
            arguments=tool_call.function.arguments,
            required=tool_dict[
                tool_call.function.name].parameters.required
        ) for tool_call in tool_calls if tool_call.function
    ] if tool_calls else None,
    usage=Usage(
        input_tokens=input_tokens,
        output_tokens=output_tokens
    )
)
```

Penyedia OpenAI juga mengimplementasikan metode stream(), yang memungkinkan untuk mengalirkan respons dari LLM. Namun, metode ini cukup panjang, jadi kita tidak akan membahas detailnya di sini.

Singkatnya, ini adalah generator yang memproses output LLM dalam potongan-potongan alih-alih menunggu seluruh respons tiba. Jika Anda penasaran, lihat kodenya di https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/blob/main/ch04/ai-six/py/backend/llm_providers/openai_provider.py

Properti `models` mengembalikan daftar model yang tersedia dari API OpenAI. Properti ini menggunakan OpenAI Python SDK untuk mengambil informasi tersebut:

```
@property
def models(self) -> list[str]:
    return [m.id for m in self.client.models.list().data]
```

Untuk melengkapi penjelasan, berikut adalah dua metode pembantu yang mengkonversi representasi alat AI-6 dan ToolCall ke representasi OpenAI SDK dari kamus Python:

```
@staticmethod
    def _tool2dict(tool: Tool) -> dict:
        """Convert the tool to a dictionary format for OpenAI API."""
        return {
            "type": "function",
            "function": {
                "name": tool.name,
                "description": tool.description,
                "parameters": {
                    "type": "object",
                    "required": tool.parameters.required,
                    "properties": {
                        param.name: {
                            "type": param.type,
                            "description": param.description
                        } for param in tool.parameters.properties
                    },
                }
            }
        }

    @staticmethod
    def _tool_call2dict(tool_call: ToolCall) -> dict:
        """Convert a ToolCall to OpenAI API format."""
        return {
            "id": tool_call.id,
            "type": "function",
            "function": {
                "name": tool_call.name,
                "arguments": tool_call.arguments
            }
        }
```

Mari kita beralih ke penyedia Ollama.

Penyedia Ollama

Ollama adalah penyedia LLM lokal yang memungkinkan Anda menjalankan model di mesin Anda sendiri. Ini cukup keren karena Anda tidak perlu membayar penyedia eksternal, dan Anda tidak perlu khawatir informasi sensitif Anda terekspos.

Implementasi penyedia Ollama ada di file `ollama_provider.py` AI-6. Ini juga mengimplementasikan antarmuka LLMProvider dan menyediakan metode yang diperlukan untuk mengirim pesan dan mendapatkan model. Ini tidak mengimplementasikan metode stream() dan bergantung pada implementasi default di kelas dasar, LLMProvider, untuk streaming respons.

Implementasinya cukup mirip dengan penyedia OpenAI, tetapi menggunakan SDK Python Ollama untuk berkomunikasi dengan API Ollama yang berjalan secara lokal. Penyedia ini membutuhkan klien Ollama untuk diinstal dan tersedia di jalur sistem, dan layanan Ollama harus berjalan. Kita akan melihatnya beraksi di *bab 6*.

Berikut adalah garis besar implementasinya. Saya melewatkan detail implementasi sebagian besar metode karena sangat mirip dengan penyedia OpenAI LLM. Dalam metode send(), saya memanggil fungsi sebenarnya ke SDK Ollama, melalui metode ollama.chat(), yang mengirimkan pesan dan alat ke model Ollama dan mengembalikan respons:

```
import json
from dataclasses import asdict
from backend.object_model import (
    LLMProvider, ToolCall, Usage, Tool, AssistantMessage, Message
)
import ollama

class OllamaProvider(LLMProvider):
    def __init__(self, model: str):
        self.model = model

    @staticmethod
    def _tool2dict(tool: Tool) -> dict:
        """Convert the tool to a dictionary format for Ollama."""
        ...

    @staticmethod
    def _fix_tool_call_arguments(messages):
        ...

    def send(
        self, messages: list[Message], tool_dict: dict[str, Tool],
        model: str | None = None
    ) -> AssistantMessage:
        """Send a message to the local Ollama model and receive a response."""
        .
        .
        .
        OllamaProvider._fix_tool_call_arguments(message_dicts)
        response: ollama.ChatResponse = ollama.chat(
            model,
            messages=message_dicts,
            tools=tool_data
        )
        .
```

```
        .
        .

    @property
    def models(self) -> list[str]:
        """Get the list of available models."""
        return [self.model]
```

Sekarang kita akan mengalihkan fokus kita ke bagaimana alat-alat dipanggil dan arsitektur yang mendasarinya.

4.3 ARSITEKTUR SISTEM ALAT

Kita telah membahas secara detail arsitektur inti dari kerangka kerja AI-6 dan komponen-komponennya: mesin inti, abstraksi penyedia LLM, dan bagaimana alat-alat dipanggil. Sekarang, mari kita lihat lebih dekat arsitektur sistem alat yang dapat diperluas. Arsitektur sistem alat dirancang untuk mencapai tujuan-tujuan berikut:

- Mendukung alat kustom
- Mendukung alat MCP
- Memungkinkan eksekusi alat yang aman
- Memungkinkan kontrol atas kumpulan alat yang digunakan untuk setiap sesi AI-6.

Mari kita periksa setiap aspeknya.

Dukungan Alat Kustom

Arsitektur alat kustom, mirip dengan abstraksi penyedia LLM, didasarkan pada kelas dasar abstrak. Ini memungkinkan pengembang untuk mengimplementasikan alat mereka sendiri yang sesuai dengan antarmuka yang bahkan lebih sederhana daripada penyedia LLM yang menyediakan kemampuan pemanggilan fungsi.

Ini dipelopori oleh OpenAI, tetapi telah diadopsi oleh semua pemain utama. Semuanya didefinisikan dalam file tool.py. Mari kita uraikan, dimulai dengan impor wajib. Ini mengimpor kelas dasar ABC dan dekorator abstractmethod dari modul abc, dekorator dataclass dari modul dataclasses, dan kelas NamedTuple dari modul typing.

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import NamedTuple
```

Kemudian, kita mendefinisikan kelas Parameter. Kelas ini diturunkan dari named tuple yang merepresentasikan satu argumen dalam panggilan fungsi dengan nama, tipe, dan deskripsi. Keuntungan menggunakan NamedTuple sebagai kelas dasar adalah kelas Parameter bersifat immutable (tidak dapat diubah) dan efisien.

```
class Parameter(NamedTuple):
    name: str
    type: str
    description: str
```

Kelas Tool adalah kelas dasar abstrak yang mewakili alat yang dapat digunakan oleh LLM melalui pemanggilan fungsi. Mari kita periksa. Cukup sederhana. Kelas ini didekorasi dengan dekorator @dataclass(slots=True). Slot membuat instance kelas tidak dapat diubah (tidak ada atribut baru yang dapat ditambahkan), menggunakan lebih sedikit memori, dan menyediakan akses yang lebih cepat ke atribut.

Kelas Tool memiliki atribut berikut: nama, deskripsi, parameter, dan permintaan (parameter yang harus diberikan saat menjalankan alat). Informasi ini diberikan kepada LLM, seperti yang kita lihat sebelumnya dalam daftar alat dari setiap permintaan. LLM dapat memutuskan alat mana yang akan dipanggil dan bagaimana cara melakukannya. Ingat diskusi kita dari *bab 3*, bahwa LLM dapat membuat keputusan ini berdasarkan informasi yang diberikan kepada mereka menggunakan deskripsi prompt dan/atau alat.

```
@dataclass(slots=True)
class Tool(ABC):
    name: str
    description: str
    parameters: list[Parameter]
    required: set[str]
```

Metode `configure` adalah metode opsional yang dapat diimplementasikan oleh subkelas untuk mengkonfigurasi alat dengan informasi apa pun. Metode ini hanya menerima sebuah kamus (dict), yang dapat berisi apa saja. Misalnya, alat yang berinteraksi dengan basis data mungkin memerlukan string koneksi sebagai konfigurasi.

```
def configure(self, config: dict) -> None:
    """Optional: configure the tool with given parameters."""
    pass
```

Terakhir, metode run() adalah metode abstrak yang harus diimplementasikan oleh subkelas. Metode ini mengeksekusi alat dengan argumen yang diberikan dari LLM. Argumen diteruskan sebagai argumen kata kunci, dan mengembalikan string. Seperti yang Anda ingat, mesin AI-6 mengeksekusi metode run() ketika sebuah alat dipanggil oleh LLM selama loop agenik.

```
@abstractmethod
def run(self, **kwargs) -> str:
    pass
```

Alat kustom memang bagus, tetapi penting juga untuk mendukung alat bawaan.

Bagaimana Dengan Alat Bawaan?

Beberapa model mendukung alat bawaan. Berikut adalah daftar alat bawaan yang didukung oleh beberapa model OpenAI (daftar ini tidak lengkap): pencarian web, pencarian file, pembuatan gambar, penerjemah kode (Python), dan penggunaan komputer. AI-6 saat ini tidak mendukung alat bawaan. Alasannya adalah penggunaan alat bawaan bergantung pada penyedia LLM tertentu, model spesifik untuk setiap penyedia LLM, dan bahkan API spesifik.

Misalnya, API penyelesaian obrolan OpenAI yang digunakan oleh AI-6 OpenAIProvider hanya mendukung alat bawaan pencarian web. Untuk menggunakan alat bawaan lain yang disebutkan sebelumnya, Anda harus menggunakan API respons OpenAI.

Di masa mendatang, AI-6 mungkin akan menambahkan dukungan alat bawaan dengan memperbarui sistem konfigurasinya agar memiliki daftar alat bawaan yang diizinkan per penyedia LLM dan per model. Untuk OpenAI, ini juga akan memerlukan penambahan dukungan untuk API respons ke OpenAIProvider. Kerangka kerja AI-6 juga mendukung alat MCP. Alat MCP sesuai dengan spesifikasi **Model Context Protocol** (MCP). MCP adalah protokol standar yang awalnya dikembangkan oleh Anthropic yang memungkinkan LLM untuk mengakses data dan alat eksternal dengan cara yang aman dan terkontrol. Kami akan mendedikasikan seluruh bagian bab 7 untuk mengeksplorasi topik ini. Sekarang mari kita bahas tentang keamanan dan izin yang perlu Anda pertimbangkan untuk sistem alat Anda.

Keamanan Dan Izin Alat

Salah satu aspek terpenting dari sistem AI berbasis agen yang menggunakan alat adalah mengontrol akses alat ke data dan sumber daya. Alasannya adalah bahwa alat-alat tersebut pada praktiknya dikendalikan oleh LLM, yang merupakan kotak hitam ajaib yang tidak dapat Anda lihat isinya. Pertimbangkan, misalnya, skenario di mana LLM memiliki akses ke alat basis data dengan akses penuh ke basis data produksi, dan ia mendapat tugas untuk meningkatkan kinerja akses data dan mengurangi biaya. LLM, tanpa panduan tambahan, mungkin memutuskan bahwa cara terbaik untuk mencapai hal itu adalah dengan menghapus basis data dan semua data di dalamnya dan hanya menyimpan data hari terakhir di memori.

Untungnya, ada banyak cara untuk mengatasi masalah ini. AI-6 mendukung semuanya hanya karena tidak memihak dan memungkinkan sistem AI berbasis agen untuk menggunakan mekanisme apa pun atau bahkan beberapa mekanisme. Mari kita lihat berbagai cara Anda dapat mengelola eksekusi alat dengan berbagai tingkat ketelitian, kepercayaan, dan kontrol:

- **Kepercayaan penuh, akses penuh**: Mode ini hanya cocok untuk tujuan pendidikan dan ketika alat yang dikonfigurasi tidak dapat mengakses data sensitif atau sumber daya penting apa pun. LLM dapat mengeksekusi alat apa pun tanpa pemeriksaan atau batasan. Waspadalah bahwa bahkan dalam skenario yang tampaknya tidak berbahaya, seperti alat dengan akses baca saja ke LLM, ini dapat menyebabkan kerusakan. Pertimbangkan, misalnya, alat dengan akses baca saja ke basis data. LLM tidak akan dapat merusak data atau menghapusnya, tetapi masih dapat membaca data dan mengeksfiltrasinya, dan dapat menyebabkan serangan DOS (penolakan layanan) sendiri dengan menjalankan kueri yang sangat besar.
- **Akses alat yang dimediasi secara terperinci**: Akses terperinci yang dimediasi berarti bahwa alat-alat tersebut tidak mengeksekusi perintah langsung pada sistem atau mengakses sumber daya secara langsung. Sebagai contoh negatif, pertimbangkan k8s-ai, yang memiliki akses langsung ke kluster Kubernetes melalui kubectl. Jika LLM memutuskan untuk menghapus semua namespace, k8s-ai akan melakukannya tanpa pertanyaan. Untuk mengatasi kekhawatiran ini, kita dapat menghapus alat kubectl dan menggantinya dengan beberapa alat seperti get_pods, create_Deployment, dan

sebagainya. Pendekatan ini memberikan akses terperinci ke kluster Kubernetes dan menghindari operasi berbahaya seperti menghapus namespace. Manfaat dari pendekatan ini adalah memberikan Anda kendali penuh, tetapi untuk sumber daya dengan antarmuka API yang kompleks, seperti Kubernetes, akan menjadi beban besar untuk mengelola akses dengan banyak alat.

- **Pengamanan**: Pengamanan adalah mekanisme lain untuk mengelola risiko yang ditimbulkan oleh LLM. Perlu dicatat bahwa pembatas (guardrails) adalah konsep yang lebih luas di luar penggunaan alat LLM. Pembatas dapat melindungi parameter yang diberikan dalam panggilan alat, respons panggilan alat, tetapi juga perintah yang dikirim ke LLM dan respons yang dihasilkannya sebelum diteruskan ke pengguna. Mari kita lihat berbagai jenis pembatas:
    - **Pembatas halusinasi** dirancang untuk mencegah AI menghasilkan konten yang tidak akurat secara faktual atau menyesatkan. Implementasi pembatas ini melibatkan proses evaluasi yang ketat yang membandingkan output yang dihasilkan dengan sumber tepercaya.
    - **Pembatas kepatuhan peraturan** memastikan bahwa konten yang dihasilkan AI mematuhi hukum dan peraturan yang berlaku, baik yang bersifat umum maupun yang disesuaikan dengan industri atau kasus penggunaan tertentu.
    - **Pembatas keselarasan** membantu menjaga konsistensi output dengan maksud pengguna dan tujuan awal, meminimalkan risiko konten menyimpang dari jalur yang seharusnya. Ini sangat berguna untuk menjaga nada, pesan, dan keselarasan merek.
    - **Pembatas validasi** berfungsi untuk menilai apakah konten yang dihasilkan memenuhi aturan yang telah ditentukan. Misalnya, memastikan bahwa detail tertentu ada atau tidak. Jika konten gagal dalam validasi, konten tersebut harus dialihkan ke alur kerja koreksi. Ini biasanya merupakan langkah otomatis terakhir, setelah itu setiap kasus yang ditandai atau tidak pasti harus melalui peninjauan manusia untuk penyelesaiannya.
- **Sandboxing berbasis kontainer**: Salah satu risiko utama penggunaan alat adalah LLM dapat mengeksekusi kode sembarangan pada sistem. Meskipun LLM itu sendiri mungkin belum memiliki niat jahat (belum), ia mungkin secara tidak sengaja, melalui penggunaan alat, menyebabkan masalah kinerja, membuat sistem rentan terhadap serangan lain, dan, secara umum, menyimpang dari perilaku yang dimaksudkan. Untuk mengurangi risiko ini, dimungkinkan dan sering direkomendasikan untuk menjalankan sistem AI agen Anda dalam kontainer, yang mengisolasinya dari mesin yang mendasarinya sampai batas tertentu. Perhatikan bahwa ini bukan jaminan mutlak, karena dimungkinkan untuk keluar dari kontainer. Saat menjalankan sistem AI secara lokal, seringkali diinginkan untuk menyediakan akses ke direktori lokal, file, dan sumber daya lainnya. Itu berarti bahwa kontainer harus dikonfigurasi untuk memungkinkan akses ke sumber daya ini. Ini adalah pertimbangan antara keamanan dan akses

terkontrol. Hal ini juga memungkinkan penyediaan kredensial terbatas ke layanan eksternal ke sistem AI agen dengan mengontrol file yang tersedia di dalam kontainer.

- **Izin pengguna**: Izin pengguna menawarkan lapisan kontrol efektif lainnya atas penggunaan alat dengan menjalankan agen di bawah pengguna sistem operasi khusus dengan hak akses rendah. Ini membatasi akses ke file, direktori, dan kemampuan sistem yang sensitif, membatasi apa yang dapat dibaca, ditulis, atau dieksekusi oleh alat yang dikontrol LLM. Bahkan tanpa kontainerisasi, pendekatan ini mencegah agen memodifikasi file sistem penting atau mengakses data yang tidak sah, menegakkan prinsip hak akses minimal. Misalnya, agen mungkin hanya diizinkan untuk menulis ke direktori tertentu, seperti /var/agent-output/, sementara diblokir dari mengakses direktori home atau konfigurasi sistem. Metode ini dapat dikombinasikan dengan kontrol akses berbasis grup, ACL sistem file, atau fitur Linux seperti ulimit dan seccomp untuk memperketat pembatasan lebih lanjut. Ini sangat berguna di lingkungan lokal atau ringan di mana sandboxing penuh tidak praktis. Meskipun izin tingkat pengguna tidak menghentikan LLM dari penyalahgunaan kemampuan yang diizinkan (misalnya, mengeksfiltrasi data yang dapat dibaca), izin tersebut memberikan batasan yang bersih dan mudah dikelola yang membatasi kerugian yang tidak disengaja dan mempermudah audit keamanan.
- **Keterlibatan manusia**: Terakhir, pendekatan keterlibatan manusia adalah cara ampuh untuk mengontrol penggunaan alat oleh LLM. Pendekatan ini melibatkan peninjauan dan persetujuan oleh manusia terhadap panggilan alat tertentu sebelum dieksekusi. Hal ini memberikan perlindungan yang kuat terhadap tindakan yang tidak disengaja atau berbahaya, karena memungkinkan penilaian manusia untuk menyaring keputusan yang mungkin dibuat LLM berdasarkan konteks yang tidak lengkap atau perintah yang ambigu. Misalnya, sebelum mengeluarkan perintah untuk menghapus sumber daya, sistem dapat menampilkan tindakan tersebut kepada operator manusia untuk konfirmasi, memastikan operasi penting hanya dilakukan ketika disetujui secara eksplisit. Pendekatan ini dapat diimplementasikan pada berbagai tingkat granularitas, mulai dari menyetujui setiap pemanggilan alat hingga intervensi selektif pada tindakan berisiko tinggi. Pendekatan ini sangat efektif di bidang di mana kesalahan sangat mahal atau keselamatan sangat penting, seperti manajemen infrastruktur, operasi keuangan, atau sistem medis. Meskipun menimbulkan latensi dan membutuhkan ketersediaan manusia, hal ini secara signifikan meningkatkan kepercayaan dan keandalan, menjadikannya ideal untuk alur kerja hibrida yang menyeimbangkan otomatisasi dengan pengawasan.

Memori adalah bagian penting lain dari sistem agen. Mari kita bahas selanjutnya.

4.4 MANAJEMEN MEMORI

Manajemen memori bisa dibilang merupakan tugas terpenting dari kerangka kerja AI agen. Hal ini dapat dilakukan oleh sistem AI host, tetapi cukup penting dan cukup rumit sehingga lebih baik membiarkan kerangka kerja AI menyelesaikan masalah ini sekali dan

membiarkan sistem AI yang dibangun di atasnya fokus pada masalah tingkat aplikasi. Seperti yang telah kita bahas di bab 2, beberapa jenis memori penting dalam sistem AI. AI-6 mengelola semua jenis memori yang berbeda melalui satu konsep: sesi.

Apa itu sesi?

Sesi adalah kumpulan semua pesan dari satu interaksi tingkat atas dengan LLM. Sesi juga menghitung token input dan output dari semua pesan. Satu sesi dapat melibatkan beberapa pesan input, melibatkan beberapa agen AI, dan dapat bertahan di berbagai waktu dan batas mesin. Hal ini didefinisikan dalam file engine/session.py.

Mari kita lihat cara kerjanya. Berikut adalah impornya. Paket json diperlukan untuk serialisasi/deserialisasi sesi. Paket uuid digunakan untuk menghasilkan ID sesi unik, kemudian banyak impor dari model objek AI-6:

```
import json
import uuid
from dataclasses import asdict

from backend.object_model import (
    Usage, Message, UserMessage, SystemMessage, AssistantMessage,
    ToolMessage, ToolCall)
```

Sesi diinisialisasi dengan direktori memori, tempat sesi persisten disimpan. Pertama, ID sesi acak dihasilkan, dan kemudian judul sesi dibuat berdasarkan ID sesi tersebut. Kita akan melihat nanti bagaimana cara mengatur judul sesi yang tersimpan melalui kelas yang berbeda, SessionManager. Kemudian, kita menginisialisasi objek penggunaan:

```
class Session:
    def __init__(self, memory_dir: str):
        self.session_id = str(uuid.uuid4())
        self.title = 'Untitled session ~' + self.session_id
        self.messages: list[Message] = []
        self.usage = Usage(0, 0)
        self.memory_dir = memory_dir
```

Metode `add_message()` cukup sederhana. Metode ini mengambil sebuah pesan dan menambahkannya ke daftar pesan sesi. Jika pesan tersebut memiliki objek penggunaan, maka metode ini memperbarui penggunaan sesi dengan menambahkan token input dan output, sehingga sesi terus mencatat total penggunaan. Kita akan melihat bagaimana objek penggunaan digunakan untuk menentukan apakah jendela konteks perlu dipadatkan pada bagian menangani konteks panjang.

```
def add_message(self, message: Message):
    """Add a Message object to the session."""
    self.messages.append(message)

    # Extract usage from AssistantMessage if present
    if hasattr(message, 'usage') and message.usage:
        self.usage = Usage(
            self.usage.input_tokens + message.usage.input_tokens,
```

```
                self.usage.output_tokens + message.usage.output_tokens
            )
```

Metode `save()` mengkonversi setiap pesan menjadi kamus menggunakan fungsi `asdict()` dari modul dataclasses, kemudian menyiapkan kamus lain yang berisi semua kamus pesan, menserialisasikan kamus tersebut ke JSON, dan menyimpannya ke file bernama `<session id>.json` di direktori memory. Sekarang, kita memiliki sesi persisten yang dapat kita pulihkan nanti bahkan jika prosesnya mengalami crash.

```
def save(self):
    # Convert Message objects to dictionaries for JSON serialization
    message_dicts = [asdict(msg) for msg in self.messages]

    d = dict(session_id=self.session_id,
             title=self.title,
             messages=message_dicts,
             usage=dict(
                 input_tokens=self.usage.input_tokens,
                 output_tokens=self.usage.output_tokens))

    filename = f"{self.memory_dir}/{self.session_id}.json"
    with open(filename, 'w') as f:
        json.dump(d, f, indent=4)
```

Mari kita lihat sesi yang tersimpan. Pengguna menanyakan direktori saat ini. LLM merespons dengan pesan asisten yang berisi permintaan untuk menjalankan alat pwd, yang mengembalikan direktori saat ini. Kerangka kerja AI-6 berhasil menjalankan alat tersebut dan mengirimkan pesan alat ke LLM dengan jawaban "/Users/gigi/git/AI-6". Kemudian, LLM merespons dengan pesan asisten terakhir, \n\nDirektori saat ini adalah `/Users/gigi/git/AI-6`. Sesi berlanjut saat pengguna mengirimkan kueri baru, tetapi saya memotongnya. Perhatikan bahwa pada akhirnya, ia menghitung token input dan output dari semua pesan:

```
{
    "session_id": "045d8889-d51b-4a48-bb3b-c97f9700af23",
    "title": "Untiled session ~045d8889-d51b-4a48-bb3b-c97f9700af23",
    "messages": [
        {
            "role": "user",
            "content": "what's curr dir"
        },
        {
            "role": "assistant",
            "content": null,
            "tool_calls": [
                {
                    "id": "tool_185f7ac72fb2404e95660d90964dc98f",
                    "type": "function",
                    "function": {
                        "name": "pwd",
                        "arguments": "{}"
                    }
                }
```

```
            ],
            "input_tokens": 551,
            "output_tokens": 11
        },
        {
            "role": "tool",
            "name": "pwd",
            "content": "/Users/gigi/git/AI-6",
            "tool_call_id": "tool_185f7ac72fb2404e95660d90964dc98f"
        },
        {
            "role": "assistant",
            "content": "\n\nThe current directory is `/Users/gigi/git/AI-6`."
        }
    ],
    "usage": {
        "input_tokens": 3068,
        "output_tokens": 57
    }
}
```

Metode `load()` adalah kebalikan dari `save()`. Metode ini menerima ID sesi, menemukan file sesi yang sesuai berdasarkan konvensi penamaan, dan memuat JSON dari file ke dalam kamus Python. Kemudian, metode ini membuat objek Session dari bidang JSON satu per satu. Untuk mengkonversi setiap pesan dari kamus ke objek Message (atau salah satu subkelasnya), metode ini menggunakan fungsi `dict_to_message()`:

```
def load(self, session_id: str):
    """Load session from disk, properly deserializing nested objects"""
    filename = f"{self.memory_dir}/{session_id}.json"
    with open(filename, 'r') as f:
        d = json.load(f)

    self.session_id = d['session_id']
    self.title = d['title']

    # Convert message dictionaries back to Message objects
    self.messages = [dict_to_message(msg) for msg in d['messages']]

    # Deserialize usage directly to a Usage object
    self.usage = Usage(d['usage']['input_tokens'],
                       d['usage']['output_tokens'])
```

Sekarang, mari kita lihat bagaimana sesi digunakan dengan berbagai jenis memori.

Memori Jangka Pendek Dalam Loop Agenik

Selama loop agenik, mesin AI-6 berpotensi membuat beberapa permintaan ke LLM, dan ia mengakumulasi semua pesan sebelumnya. Setiap permintaan alat dan responsnya setelah dieksekusi ditambahkan ke sesi aktif dan tersedia untuk LLM pada iterasi berikutnya dari loop agenik karena sesi menjadi jendela konteks. Misalnya, pertimbangkan interaksi berikut:

```
User: what is the current directory?
LLM (tool call): pwd (print working directory)
User: How many Python files in the current directory?
```

```
LLM (tool call): ls *.py
AI-6 (tool call response): foo.py bar.py
LLM (final response): There are two Python files in the current directory
User: What are they?
LLM (final response): The files are foo.py and bar.py
```

Pada setiap titik, LLM menerima seluruh percakapan dan bukan hanya pesan terakhir. Ketika pengguna bertanya "Apa itu?", LLM dapat menyimpulkan dari riwayat percakapan bahwa pengguna merujuk pada file Python di direktori saat ini, dan dapat mengembalikannya segera tanpa panggilan alat lain.

Sesi Dan Percakapan

Ketika loop agen berakhir, respons akhir dikembalikan kepada pengguna (atau aplikasi AI agen). Pengguna dapat melanjutkan percakapan dengan mengirimkan pesan pengguna baru yang akan memulai loop agen baru, tetapi dengan konteks yang mencakup semua pesan sebelumnya dari sesi saat ini.

Memori Jangka Panjang Dan Pemuatan Sesi

Kami telah menunjukkan bagaimana sesi dapat menyimpan dan memuat dirinya sendiri ke/dari file. Tetapi apa mekanisme yang memungkinkannya? Di sinilah pengelola sesi berperan. Pengelola sesi diimplementasikan dalam file engine/session_manager. Ini mengekspos tiga metode untuk mengelola sesi persisten:

- `list_sessions()`
- `delete_session()`
- `set_title()`

Saat aplikasi AI-6 dimulai, aplikasi dapat memanggil list_sessions() untuk meninjau semua sesi yang ada dan memutuskan untuk memuat sesi yang ada atau menghapus sesi lama. Aplikasi juga dapat mengatur judul untuk sesi yang ada. Mari kita lihat bagaimana pengelola sesi bekerja. Sesi diinisialisasi dengan direktori memori, tempat file sesi disimpan:

```
import os
import json

class SessionManager:
    def __init__(self, memory_dir: str):
        self.memory_dir = memory_dir
```

Metode `set_title()` menerima ID sesi dan judul. Metode ini menemukan file sesi (menimbulkan pengecualian jika tidak ditemukan), memuat sesi dari file JSON sebagai kamus Python, mengganti judul, dan menyimpannya kembali. Perhatikan urutan yang tidak biasa yaitu `seek`, `dump`, dan `truncate` untuk memperbarui file. Ini adalah opsi yang aman untuk memperbarui file menggunakan satu handle file dan dengan risiko minimal memotong file lalu menyebabkan crash dan meninggalkan file kosong. Jika Anda hanya menimpa file dengan perintah seperti `open(filename, 'w').write()` dan proses crash segera setelah perintah `open()`, maka Anda akan mendapatkan file kosong:

```
def set_title(self, session_id: str, title: str):
    """Set the title of a session."""
    sessions = self.list_sessions()
    if session_id not in sessions:
        raise RuntimeError(f"Session {session_id} not found.")

    filename = sessions[session_id]['filename']
    with open(filename, 'r+') as f:
        data = json.load(f)
        data['title'] = title
        f.seek(0)
        json.dump(data, f, indent=4)
        f.truncate()
```

Metode `list_sessions()` mencantumkan semua file .json di direktori memori, memverifikasi bahwa file tersebut berisi JSON yang valid, dan mengembalikan daftar kamus di mana setiap kamus sesi berisi judul dan nama filenya. Jika ada yang salah, metode ini hanya mencetak pesan kesalahan dan melanjutkan ke sesi berikutnya:

```
def list_sessions(self) -> list[dict]:
    """List all sessions in the memory directory.

    Returns:
        A dictionary mapping session IDs to tuples of (name, filename)
    """
    sessions = {}
    files = os.listdir(self.memory_dir)
    for f in files:
        if f.endswith('.json'):
            try:
    # Get session ID from the filename (dropping the .json extension)
                session_id = os.path.basename(f).rsplit('.json', 1)[0]

    # Try to open and parse the file to verify it's valid JSON
                full_path = os.path.join(self.memory_dir, f)
                with open(full_path, 'r') as file:
                    try:
                        # Attempt to parse the JSON
                        session = json.loads(file.read())
                        # Only add if we could parse the JSON
                        sessions[session_id] = dict(
                            title=session['title'],
                            filename=full_path)
                    except json.JSONDecodeError:
                        # Skip files with invalid JSON
                        print(f"Skipping file with invalid JSON: {f}")
                        continue
            except Exception as e:
                # Skip any files that cause other errors
                print(f"Error parsing session file {f}: {e}")
                continue

    return sessions
```

Menghapus sesi juga sangat mudah. Daftar sesi, dapatkan nama filenya, dan hapus file tersebut:

```
def delete_session(self, session_id: str):
    """Delete a session by its ID."""
    sessions = self.list_sessions()
    if session_id not in sessions:
        raise RuntimeError(f"Session {session_id} not found.")

    filename = sessions[session_id]['filename']
    os.remove(filename)
```

Manajer sesi dapat mengelola sesi, tetapi seiring bertambahnya jumlah sesi, konteks dapat menjadi jenuh. Mari kita pahami masalahnya dan pertimbangkan cara mengatasinya.

Menangani konteks yang panjang

Salah satu masalah terbesar yang perlu ditangani oleh kerangka kerja AI berbasis agen adalah mengelola ukuran konteks, sehingga sesuai dengan ukuran jendela konteks model. Alur kerja berbasis agen dapat mencakup banyak permintaan ke LLM dalam jangka waktu yang lama dan mengakumulasi semakin banyak konteks.

Solusinya adalah mengompres konteks ketika mendekati batas. Tetapi, bagaimana Anda memutuskan konteks mana yang harus disimpan? Nah, LLM sangat bagus dalam meringkas blok teks yang panjang. Mari kita minta LLM untuk mengompres konteks. Inilah tepatnya yang dilakukan oleh peringkas. Ini diimplementasikan dalam file engine/summarizer. Mari kita lihat cara kerjanya. Ini diinisialisasi dengan penyedia LLM:

```
from backend.object_model import (
    Message, LLMProvider, SystemMessage, UserMessage)

class Summarizer:
    """Utility class for summarizing sessions using an LLM."""
    def __init__(self, llm_provider: LLMProvider):
        """Initialize the summarizer with an LLM provider.

        Args:
            llm_provider: An LLM provider instance
        """
        self.llm_provider = llm_provider
```

Metode summarize() menerima daftar pesan dan ID model. Metode ini menyiapkan prompt sistem dengan instruksi tentang cara meringkas percakapan dan kemudian memformat pesan sesi menjadi satu string yang berisi peran dan konten setiap pesan (menggunakan metode `_format_session()`). Kemudian, metode ini mengirimkan pesan sistem dan pesan pengguna yang berisi string sesi gabungan ke penyedia, yang mengembalikan ringkasan sesi. Perhatikan bahwa ini adalah panggilan send() tunggal langsung ke penyedia LLM. Tidak ada sesi yang terlibat dalam interaksi ini:

```
def summarize(self, messages: list[Message], model_id: str) -> str:
    """Summarize a list of messages using the LLM.
```

```
    Args:
        messages: List of message dictionaries to summarize
        model_id: ID of the model to use for summarization

    Returns:
        A string summary of the session
    """

    # Format messages for the LLM
    formatted_messages = [
        SystemMessage(
            content=(
                "You are a helpful assistant tasked with summarizing a
conversation. "
                "Create a concise summary that captures the key points,
questions, decisions, and context "
                "from the session. The summary should be informative enough
that someone "
                "reading it would understand what was discussed, what
conclusions were reached, "
                "and what important context should be carried forward. "
                "Focus on preserving information that will be useful for
continuing the conversation, "
                "including names, technical terms, important numbers, and
specific details that might "
                "be referenced later. Avoid unnecessary details, repetitive
information, or tangential discussions."
            )
        ),
        UserMessage(
            content=(
                "Please summarize the following session:\n\n" +
                self._format_session(messages)
            )
        )
    ]

    # Get summary from LLM
    response = self.llm_provider.send(
        formatted_messages, {}, model_id)

    return response.content.strip()
```

Berikut adalah metode `statis _format_session()` yang mengubah daftar pesan menjadi satu string besar yang berisi peran dan isi setiap pesan:

```
@staticmethod
@staticmethod
def _format_session(messages: list[Message]) -> str:
    """
    Format a list of messages into a readable session.

    Args:
        messages: List of message dictionaries

    Returns:
```

```
        Formatted session string
    """
    formatted = []

    for msg in messages:
        role = msg.role
        content = msg.content

        if role == "tool":
            tool_name = getattr(msg, 'name', 'unknown tool')
            formatted.append(f"Tool ({tool_name}): {content}")
        else:
            formatted.append(f"{role.capitalize()}: {content}")

    return "\n\n".join(formatted)
```

Keputusan tentang apakah ringkasan diperlukan atau tidak terjadi di dalam mesin pada metode `_checkpoint_if_needed()`, yang menyimpan sesi secara berkala dan juga memeriksa jumlah total token dalam sesi (diperoleh dari objek penggunaan) terhadap ukuran jendela konteks. Jika jumlah token melebihi ambang batas tersebut, maka ringkasan akan dilakukan, dan sesi akan menggunakan konteks ringkasan yang lebih pendek mulai saat ini. Berikut adalah cara metode ini didefinisikan:

```
def _checkpoint_if_needed(self):
    """Check if we need to save a checkpoint and do so if needed."""
    self.message_count_since_checkpoint += 1

    # Only save if we've reached the checkpoint interval exactly
    if self.message_count_since_checkpoint == self.checkpoint_interval:
        self.session.save()
        self.message_count_since_checkpoint = 0

    # Check and summarize if above token threshold (80% of context window)
    total_tokens = (
        self.session.usage.input_tokens + \
        self.session.usage.output_tokens
    )

    if total_tokens >= self.token_threshold:
        context_window_size = get_context_window_size(
            self.default_model_id)
        print(
            f"Session tokens ({total_tokens}) have reached "
{self.summary_threshold_ratio * 100}% of context window ({context_window_size}).
Summarizing:..."
            )
            self._summarize_and_reset_session()
```

Perlu dicatat bahwa peringkasan otomatis oleh LLM bukanlah satu-satunya cara untuk mengompres jendela konteks ketika ukurannya menjadi terlalu besar. Pendekatan lain meliputi **pencarian semantik dengan penyematan** untuk mengambil hanya giliran masa lalu yang paling relevan, **pemangkasan berbasis aturan** untuk menghilangkan pertukaran yang sepele atau berulang, **penyimpanan terstruktur** fakta dan entitas untuk injeksi sesuai permintaan,

dan teknik **pengkodean ulang yang efisien token** seperti buku kode, format singkatan khusus, atau skema kompresi khusus domain.

Di masa depan, AI-6 mungkin akan berevolusi untuk membuat manajemen memori dapat diperluas. Mesin tidak akan langsung memanggil summarizer ketika jendela konteks menjadi terlalu besar, tetapi akan diinisialisasi dengan pengelola konteks generik yang akan mengimplementasikan beberapa metode pemadatan. Pendekatan ini akan mengikuti cetak biru alat dan LLMProvider, di mana mesin berinteraksi dengan komponen lain melalui antarmuka abstrak. Secara khusus, antarmuka ContextManager generik dengan metode summarize() dapat diimplementasikan oleh beberapa pengelola konteks konkret yang dapat dikonfigurasi oleh pengguna untuk memilih metode terbaik untuk meringkas sesi.

4.5 RINGKASAN

Dalam bab ini, kita telah mempelajari bahwa kerangka kerja AI-6 adalah platform yang tangguh dan dapat diperluas yang dirancang untuk mendukung sistem AI agen yang canggih. Dibangun di atas agen k8s-ai minimal, AI-6 memperkenalkan arsitektur backend modular yang mengkoordinasikan eksekusi alat, manajemen memori, dan komunikasi dengan beberapa penyedia LLM. Intinya adalah kelas Engine, yang mengatur loop agen, menerima input pengguna, memanggil alat sesuai kebutuhan, mengelola riwayat sesi, dan berinteraksi dengan LLM melalui antarmuka penyedia terpadu. Konfigurasi dipusatkan melalui objek Config sederhana, memungkinkan pengguna untuk mendefinisikan alat, direktori memori, default model, dan parameter perilaku. Sistem ini dirancang agar tidak memihak, sehingga mudah untuk membangun sistem khusus atau sistem yang lebih terarah di atasnya.

Kekuatan utama AI-6 terletak pada abstraksi penyedia dan alat LLM. Ia mendukung penyedia jarak jauh seperti OpenAI dan penyedia lokal seperti Ollama, menggunakan antarmuka umum yang menangani pemformatan pesan, integrasi alat, dan penguraian respons. Alat itu sendiri didefinisikan melalui antarmuka Python ringan yang mendukung deskripsi, parameter bertipe, dan logika eksekusi. Kerangka kerja ini dirancang untuk mengelola penggunaan alat dengan aman dengan menawarkan dukungan untuk definisi alat yang terperinci, pengaman runtime, sandboxing melalui izin OS atau kontainer, dan kontrol manusia dalam loop bila diperlukan. Ini memberikan fondasi yang kuat untuk membangun sistem AI yang aman dan andal yang mampu menggunakan kemampuan eksternal secara efektif. Memori dikelola melalui konsep yang disebut sesi, yang merangkum semua riwayat interaksi dan penggunaan token. Sesi dapat disimpan, dilanjutkan, diringkas, dan diperiksa melalui objek SessionManager khusus. Ketika sebuah sesi menjadi terlalu besar, modul peringkasan bawaan menggunakan LLM untuk mengompres pesan-pesan sebelumnya sambil mempertahankan konteks penting. Ini memastikan bahwa AI-6 dapat beroperasi pada percakapan yang panjang tanpa melampaui batas model. Bersama-sama, fitur-fitur ini membentuk kerangka kerja yang kohesif dan siap produksi untuk mengelola perilaku agen, penggunaan alat, dan memori jangka panjang dalam sistem AI. Pada bab selanjutnya, kita akan melihat bagaimana mengimplementasikan alat khusus untuk AI-6 dan bereksperimen dengan berbagai kombinasi alat untuk melakukan tugas-tugas yang bermanfaat.

BAB 5
MENGIMPLEMENTASIKAN ALAT KUSTOM

5.1 PENDAHULUAN

Pada bab sebelumnya, kita telah mengeksplorasi arsitektur backend dari kerangka kerja AI-6 secara detail, dengan fokus pada bagaimana mesin mengatur loop agen, mengelola memori dan status sesi, serta berintegrasi dengan berbagai penyedia dan alat LLM. Kita telah melihat bagaimana alat dan model ditemukan secara dinamis dan bagaimana mesin menangani koordinasinya melalui abstraksi terpadu.

Pada bab ini, kita akan mengalihkan fokus dari arsitektur mesin secara keseluruhan ke investigasi yang lebih mendalam tentang sistem alat itu sendiri. Sementara bab 4 memperlakukan alat sebagian besar sebagai kotak hitam yang dapat dipasang, di sini kita akan melihat ke dalam dan memeriksa bagaimana alat generik dirancang agar dapat digunakan kembali, konsisten, dan portabel di seluruh penyedia LLM. Kita akan mengeksplorasi bagaimana alat dapat didefinisikan dengan skema terstruktur, bagaimana argumen divalidasi, dan bagaimana alat tersebut didaftarkan dan diterjemahkan ke dalam format khusus penyedia.

Kita akan mulai dengan membahas bagaimana AI-6 memungkinkan penggunanya untuk mengimplementasikan sebuah alat dalam format alat AI-6 generik. Implementasi alat generik tunggal ini akan bekerja dengan penyedia LLM apa pun, seperti OpenAI, Ollama, atau beberapa penyedia LLM di masa mendatang yang bahkan belum ada. Kita juga akan melihat API spesifik OpenAI dan Ollama untuk memahami bagaimana mereka menyediakan fungsionalitas serupa melalui spesifikasi yang sedikit berbeda, dan bagaimana AI-6 dapat memanfaatkan kedua penyedia ini tanpa mengekspos internalnya kepada penggunanya, diikuti dengan tinjauan definisi skema dan strategi validasi input.

Dari sana, kita akan melihat bagaimana AI-6 memetakan representasi alat internalnya ke format spesifik penyedia (seperti pemanggilan fungsi OpenAI). Terakhir, kita akan memeriksa bagaimana eksekusi alat dikelola, termasuk bagaimana hasil ditangkap dan dialihkan kembali ke LLM melalui sesi. Bab ini akan meletakkan dasar untuk membangun alat yang ampuh, aman, dan interoperabel yang memperluas kemampuan sistem AI agen. Dalam bab ini, kita akan membahas topik-topik utama berikut:

- Mendefinisikan alat generik
- Mendefinisikan alat kustom AI-6
- Mengimplementasikan alat baru secara manual

5.2 PERSYARATAN TEKNIS

Tidak ada persyaratan teknis khusus untuk bab ini. Jika Anda ingin membuat alat kustom Anda sendiri, Anda dapat menggunakan pustaka Python apa pun. Anda hanya perlu mengimplementasikan antarmuka Tool, yang merupakan bagian dari kerangka kerja AI-6 itu sendiri, untuk menyesuaikan alat Anda. Kode untuk bab ini tersedia di sini:

https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and A2A/tree/main/ch05/ai-six.

5.3 MENDEFINISIKAN ALAT YANG TIDAK BERGANTUNG PADA PENYEDIA

Setiap API penyedia LLM, seperti OpenAI, Anthropic, Gemini, dan Ollama, memiliki cara tersendiri untuk menentukan alat. Karena AI-6 perlu bekerja dengan semua penyedia LLM, ia mendefinisikan formatnya sendiri yang tidak bergantung pada penyedia. Penyedia LLM kerangka kerja AI-6 yang berinteraksi langsung dengan API atau SDK dari penyedia LLM pihak ketiga mengetahui cara menerjemahkan definisi alat generik ke definisi alat khusus penyedia.

Definisi alat AI-6 harus merupakan superset dari informasi yang dibutuhkan oleh semua API. Fokus definisi alat adalah pada pemanggilan fungsi, meskipun alat bawaan juga didukung. Mari kita lihat definisi alat dari SDK Python penyedia LLM, seperti OpenAI dan Ollama. Alasan kita melihat SDK Python dan bukan API REST mentah adalah karena penyedia LLM AI-6 diimplementasikan dalam Python dan menggunakan SDK Python dari penyedia LLM pihak ketiga yang sesuai. Mari kita lihat terlebih dahulu SDK Python OpenAI dan lihat bagaimana ia menentukan alat dan panggilan alat.

Spesifikasi Alat Python OpenAI

SDK Python OpenAI tersedia di https://github.com/openai/openai-python. Ini dihasilkan secara otomatis dari spesifikasi OpenAPI dari OpenAI REST API, sehingga memiliki fidelitas 100% terhadap API OpenAI yang sebenarnya.

Definisi alat yang kita minati ada di endpoint penyelesaian obrolan, yang dapat diakses melalui metode create() dari kelas Completions dari API Penyelesaian Obrolan. Jangan bingung dengan API Penyelesaian lama.

Metode ini melakukan panggilan ke LLM (opsional dengan banyak argumen) dan mendapatkan respons. Respons ini dapat berupa respons akhir atau panggilan alat, yang akan diproses oleh AI-6 (seperti yang telah kita bahas di bab sebelumnya).

Metode create() memiliki delapan versi yang di-overload. Ini berarti ada delapan metode berbeda, dan semuanya diberi nama create() dengan argumen yang sedikit berbeda. Alasan mengapa metode yang di-overload berguna, terutama untuk metode dengan banyak argumen, adalah karena versi overload yang berbeda dapat memberikan tingkat kontrol yang berbeda dibandingkan dengan kemudahan penggunaan:

```
@overload
    def create(
        self,
        *,
        messages: Iterable[ChatCompletionMessageParam],
        model: Union[str, ChatModel],
        audio: Optional[ChatCompletionAudioParam] | NotGiven = NOT_GIVEN,
        frequency_penalty: Optional[float] | NotGiven = NOT_GIVEN,
        function_call: completion_create_params.FunctionCall | NotGiven =
NOT_GIVEN,
        functions: Iterable[completion_create_params.Function] | NotGiven
NOT_GIVEN,
        logit_bias: Optional[Dict[str, int]] | NotGiven = NOT_GIVEN,
```

```
        logprobs: Optional[bool] | NotGiven = NOT_GIVEN,
        max_completion_tokens: Optional[int] | NotGiven = NOT_GIVEN,
        max_tokens: Optional[int] | NotGiven = NOT_GIVEN,
        metadata: Optional[Metadata] | NotGiven = NOT_GIVEN,
        modalities: Optional[List[Literal["text", "audio"]]] | NotGiven =
NOT_GIVEN,
        n: Optional[int] | NotGiven = NOT_GIVEN,
        parallel_tool_calls: bool | NotGiven = NOT_GIVEN,
        prediction: Optional[ChatCompletionPredictionContentParam] | NotGiven =
NOT_GIVEN,
        presence_penalty: Optional[float] | NotGiven = NOT_GIVEN,
        reasoning_effort: Optional[ReasoningEffort] | NotGiven = NOT_GIVEN,
        response_format: completion_create_params.ResponseFormat | NotGiven =
NOT_GIVEN,
        seed: Optional[int] | NotGiven = NOT_GIVEN,
        service_tier: Optional[Literal["auto", "default", "flex", "scale"]]
NotGiven = NOT_GIVEN,
    stop: Union[Optional[str], List[str], None] | NotGiven = NOT_GIVEN,
    store: Optional[bool] | NotGiven = NOT_GIVEN,
    stream: Optional[Literal[False]] | NotGiven = NOT_GIVEN,
    stream_options: Optional[ChatCompletionStreamOptionsParam] | NotGiven =
NOT_GIVEN,
    temperature: Optional[float] | NotGiven = NOT_GIVEN,
    tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN,
    tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN, # 
    top_logprobs: Optional[int] | NotGiven = NOT_GIVEN,
    top_p: Optional[float] | NotGiven = NOT_GIVEN,
    user: str | NotGiven = NOT_GIVEN,
    web_search_options: completion_create_params.WebSearchOptions |
NotGiven = NOT_GIVEN,
# Use the following arguments if you need to pass additional parameters to the API
that aren't available via kwargs.
        # The extra values given here take precedence over values defined on the
client or passed to this method.
        extra_headers: Headers | None = None,
        extra_query: Query | None = None,
        extra_body: Body | None = None,
        timeout: float | httpx.Timeout | None | NotGiven = NOT_GIVEN,
    ) -> ChatCompletion:
        """
        ...
        """
```

Beberapa metode yang kelebihan beban bersifat sinkron, dan beberapa bersifat asinkron, tetapi semuanya mendefinisikan alat-alat tersebut dengan cara yang sama:

```
tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN,
```

Apa arti baris ini?

1. `Iterable[ChatCompletionToolParam]` berarti bahwa alat yang disediakan harus berupa urutan iterable di mana setiap elemennya bertipe ChatCompletionToolParam.
2. Bagian kedua, `| NotGiven = NOT_GIVEN`, berarti bahwa argumen alat dapat berupa tipe NotGiven dan nilai default jika tidak ada alat yang disediakan. Nilai NOT_GIVEN adalah konstanta bertipe NotGiven.

Dengan kata lain, argumen tools dapat berupa salah satu dari dua tipe (Python mengizinkan hal itu). Tipe pertama adalah `Iterable[ChatCompletionToolParam]`, yang dapat Anda bayangkan sebagai daftar item dari tipe ChatCompletionToolParam. Tipe kedua adalah NotGiven. Nilai default dari argumen tools adalah `NOT_GIVEN`. Jika pemanggil tidak menyediakan tools apa pun, maka nilai argumen tools akan menjadi nilai default, `NOT_GIVEN`.

Jika Anda bertanya-tanya apa itu NotGiven, itu adalah kelas sentinel sederhana yang digunakan untuk menunjukkan parameter yang tidak ditentukan dan memberikannya nilai default `NOT_GIVEN`. Baiklah, Mari kita pahami apa itu ChatCompletionToolParam. Ini adalah kelas dengan dua field: function dan type. Kedua field ini wajib diisi. Field function bertipe FunctionDefinition, dan field type harus berupa string literal "function". Berikut cara Anda dapat mendefinisikan kelas ini:

```
class ChatCompletionToolParam(TypedDict, total=False):
    function: Required[FunctionDefinition]

    type: Required[Literal["function"]]
    """The type of the tool. Currently, only `function` is supported."""
```

Ini bisa sedikit membingungkan karena ada dua jenis tipe yang berbeda di sini. Tipe fungsi, Required[FunctionDefinition], adalah petunjuk tipe untuk tipe Python yang diperlukan, FunctionDefinition. API Penyelesaian Obrolan OpenAI menarik karena hanya mendukung satu jenis alat, yaitu alat fungsi, tetapi API didefinisikan secara lebih umum seolah-olah dapat mendukung jenis alat tambahan.

API OpenAI lainnya memang mendukung beberapa jenis alat. Assistants%20API%20(beta) yang sudah usang mendukung dua alat bawaan: file_search dan code_interpreter. Berikut adalah definisi dari alat CodeInterpreter dan FileSearch bawaan:

```
class ToolResourcesCodeInterpreter(BaseModel):
    file_ids: Optional[List[str]] = None
    """
    A list of [file](https://platform.openai.com/docs/api-reference/files) IDs
made
    available to the `code_interpreter` tool. There can be a maximum of 20 files
    associated with the tool.
    """


class ToolResourcesFileSearch(BaseModel):
    vector_store_ids: Optional[List[str]] = None
    """
    The ID of the
    [vector store](https://platform.openai.com/docs/api-reference/vector-stores/
object)
    attached to this assistant. There can be a maximum of 1 vector store attached
to
    the assistant.
    """

class ToolResources(BaseModel):
```

```
    code_interpreter: Optional[ToolResourcesCodeInterpreter] = None
    file_search: Optional[ToolResourcesFileSearch] = None
```

API Respons yang baru mendukung alat-alat berikut yang didefinisikan dalam types/responses/tool_param.py:

```
ToolParam: TypeAlias = Union[
    FunctionToolParam,
    FileSearchToolParam,
    WebSearchToolParam,
    ComputerToolParam,
    Mcp,
    CodeInterpreter,
    ImageGeneration,

    LocalShell,
]
```

Saat ini, penyedia OpenAI LLM dari AI-6 menggunakan API Penyelesaian Obrolan, jadi mari kita kembali ke kelas FunctionDefinition. Kelas ini mewakili alat fungsi tunggal dan memiliki nama, deskripsi, parameter opsional, dan flag ketat opsional (False kecuali secara eksplisit diatur ke True):

```
class FunctionDefinition(BaseModel):
    name: str
    """
    The name of the function to be called.

    Must be a-z, A-Z, 0-9, or contain underscores and dashes, with a maximum
length
    of 64.
    """

    description: Optional[str] = None
    """
    A description of what the function does, used by the model to choose when and
    how to call the function.
    """

    parameters: Optional[FunctionParameters] = None
    """
    The parameters the functions accepts, described as a JSON Schema object.

    See the [guide](https://platform.openai.com/docs/guides/function-calling) for
    examples, and the
    [JSON Schema reference](https://json-schema.org/understanding-json-schema/)
for
    documentation about the format.

    Omitting `parameters` defines a function with an empty parameter list.
    """

    strict: Optional[bool] = None
    """
```

```
    Whether to enable strict schema adherence when generating the function call.

    If set to true, the model will follow the exact schema defined in the
    `parameters` field. Only a subset of JSON Schema is supported when `strict`
is
    `true`. Learn more about Structured Outputs in the
    [function calling guide](docs/guides/function-calling).
    """
```

Mari kita telusuri lebih dalam dan lihat apa sebenarnya FunctionParameters di kelas FunctionDefinition. Ternyata itu hanyalah kamus sederhana yang mengubah string menjadi objek apa pun.

```
# File generated from our OpenAPI spec by Stainless.

from typing import Dict
from typing_extensions import TypeAlias

__all__ = ["FunctionParameters"]

FunctionParameters: TypeAlias = Dict[str, object]
```

Mari kita rangkum persyaratan definisi alat API Penyelesaian Obrolan seperti yang ditentukan oleh OpenAI:

- Setiap alat harus bertipe "fungsi"
- Setiap alat memiliki nama dan deskripsi
- Sebuah alat dapat memiliki nol atau lebih parameter dengan nama dan tipe nilai arbitrer
- Sebuah alat dapat menentukan apakah keluarannya ketat

Ada satu elemen penting lagi untuk spesifikasi alat OpenAI. Selain argumen tools dari metode create(), ada juga argumen tool_choice:

```
tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN,
```

Pilihan alat dapat mengambil salah satu nilai berikut: "none" (jangan panggil alat apa pun), "auto" (Anda memutuskan apakah ingin memanggil satu atau lebih alat dan alat mana saja), "required" (Anda harus memanggil setidaknya satu alat; Anda memutuskan alat mana yang akan dipanggil), atau nama alat tertentu (Anda harus memanggil alat ini):

```
ChatCompletionToolChoiceOptionParam: TypeAlias = Union[
    Literal["none", "auto", "required"],
    ChatCompletionNamedToolChoiceParam
]
```

Jika ini terdengar membingungkan, itu karena memang membingungkan. Saya menulis seluruh postingan blog tentang masalah pemilihan alat dan bagaimana hal itu dapat disederhanakan secara signifikan: Memperbaiki Pemanggilan Alat OpenAI. Anda dapat

menemukannya di sini: https://medium.com/@the.gigi/fixing-the-openai-tool-calling-api-ede67419f9f5. Baiklah. Mari kita ingat ini dan lihat bagaimana Ollama Python SDK menentukan panggilan alat.

Spesifikasi Alat Ollama Python

Ollama Python SDK sudah merupakan abstraksi atas banyak API karena mendukung banyak penyedia melalui antarmuka terpadu. Mari kita lihat bagaimana alat ditentukan dalam Ollama Python SDK. Ollama menggunakan metode chat() (beberapa overload serupa dengan OpenAI) dari kelas Client untuk berinteraksi dengan LLM dan menentukan alat:

```
@overload
def chat(
    self,
    model: str = '',
    messages: Optional[
        Sequence[Union[Mapping[str, Any], Message]]] = None,
    *,
    tools: Optional[
        Sequence[Union[Mapping[str, Any], Tool, Callable]]] = None, # 
    stream: Literal[False] = False,
    think: Optional[bool] = None,
    format: Optional[Union[Literal['', 'json'], JsonSchemaValue]] = None,
    options: Optional[Union[Mapping[str, Any], Options]] = None,
    keep_alive: Optional[Union[float, str]] = None,
) -> ChatResponse: ...
```

Definisi alat ini cukup panjang. Mari kita uraikan. Parameter bernama tools bersifat opsional dan defaultnya adalah None. Jika diberikan, parameter ini harus berupa urutan (seperti daftar atau tuple) di mana setiap item adalah salah satu dari tiga tipe yang diizinkan:

- Skema JSON Mapping[str, Any] sebagai kamus dengan kunci string dan nilai dari tipe apa pun
- Instance Ollama Tool (kelas yang disediakan oleh SDK Ollama yang membungkus metadata dan perilaku alat)
- Objek Callable yang mengikuti docstring gaya Google untuk dikonversi menjadi instance Ollama Tool:

```
tools: Optional[
    Sequence[Union[Mapping[str, Any], Tool, Callable]]
] = None
```

Terlepas dari representasinya, semuanya akan berakhir sebagai urutan instance Ollama Tool:

```
class Tool(SubscriptableBaseModel):
    type: Optional[Literal['function']] = 'function'

    class Function(SubscriptableBaseModel):
        name: Optional[str] = None
        description: Optional[str] = None

        class Parameters(SubscriptableBaseModel):
            model_config = ConfigDict(populate_by_name=True)
```

```
            type: Optional[Literal['object']] = 'object'
            defs: Optional[Any] = Field(None, alias='$defs')
            items: Optional[Any] = None
            required: Optional[Sequence[str]] = None

class Property(SubscriptableBaseModel):
    model_config = ConfigDict(arbitrary_types_allowed=True)

    type: Optional[Union[str, Sequence[str]]] = None
    items: Optional[Any] = None
    description: Optional[str] = None
    enum: Optional[Sequence[Any]] = None
    properties: Optional[Mapping[str, Property]] = None

parameters: Optional[Parameters] = None
function: Optional[Function] = None
```

Kita tidak akan menguraikan setiap baris dalam kode Python ini dan hanya mencatat bahwa Ollama melampirkan beberapa informasi model ke definisi alat, yang mungkin digunakan saat menerjemahkan Alat ke definisi alat khusus model. Hal penting yang perlu diperhatikan adalah, mirip dengan API Penyelesaian OpenAI, satu-satunya tipe alat yang didukung adalah "fungsi".

Singkatnya, OpenAI dan Ollama mendefinisikan alat dalam format mereka sendiri, tetapi secara konseptual, mereka berisi informasi yang setara untuk setiap alat, yang sangat bagus untuk AI-6. Kita sekarang siap untuk melihat skema spesifikasi alat generik AI-6.

Mendefinisikan Skema Spesifikasi Alat AI-6

Kita telah membahas kelas Parameter dan Alat AI-6 di bab sebelumnya. Berikut adalah penyegaran singkat. Tujuannya adalah kesederhanaan tanpa terlalu banyak hal yang rumit. Satu fitur khusus yang ada di AI-6 dan tidak ada dalam spesifikasi alat OpenAI dan Ollama adalah bidang yang diperlukan dari kelas Alat. Bidang ini adalah sekumpulan nama parameter yang harus diberikan saat LLM memanggil alat tersebut. Ini adalah bagian dari spesifikasi alat OpenAI dalam REST API, tetapi entah mengapa tidak masuk ke dalam Python SDK:

```
class Parameter(NamedTuple):
    name: str
    type: str
    description: str

@dataclass(slots=True)
class Tool(ABC):
    name: str
    description: str
    parameters: list[Parameter]
    required: set[str]
    def configure(self, config: dict) -> None:
        """Optional: configure the tool with given parameters."""

    @abstractmethod
    def run(self, **kwargs) -> str:
        """Execute the tool using the given arguments and return the result as
 a string."""
```

Kode sederhana ini adalah keseluruhan skema spesifikasi alat AI-6. Sangat sederhana dan mudah dipahami.

Nah, Anda telah melihat bagaimana AI-6 dapat mengabstraksi penyedia dan implementasi masing-masing untuk mendefinisikan alat dengan antarmuka LLMProvider. Sekarang kita akan melihat bagaimana mendefinisikan alat kustom yang dapat diterjemahkan ke format alat penyedia LLM aktif di bagian selanjutnya.

5.4 MENDEFINISIKAN ALAT KUSTOM AI-6

AI-6 hadir dengan banyak contoh alat siap pakai, dan mudah untuk mengimplementasikan alat baru. Di bagian ini, kita akan meninjau beberapa alat yang ada dan kemudian mengimplementasikan alat baru dari awal. Proses mendefinisikan alat kustom AI-6 terdiri dari langkah-langkah berikut:

1. Buat kelas Python baru.
2. Turunkan dari kelas dasar Tool secara langsung atau tidak langsung.
3. Implementasikan metode run().
4. Secara opsional, implementasikan metode configure().

Mari kita mulai dengan alat baris perintah umum, yang sangat ampuh.

Alat Baris Perintah Umum

Banyak alat berguna yang disediakan AI-6 secara bawaan adalah alat baris perintah. Alat-alat ini berjalan di mana pun AI-6 berjalan (di mesin Anda, di dalam kontainer, di cloud, dan lain-lain.). AI-6 menyediakan kelas dasar untuk alat baris perintah yang menangani semua kode boilerplate yang diperlukan untuk menjalankan program baris perintah dengan argumen sembarang dan menawarkan opsi untuk menjalankan program baris perintah sebagai pengguna khusus (berbeda dari pengguna saat ini). Mari kita lihat bagaimana kelas CommandTool bekerja sebelum kita memeriksa alat-alat yang diturunkan darinya.

Ini mengimpor paket sh untuk menjalankan perintah shell, paket bawaan shlex untuk mengurai argumen baris perintah POSIX, dan Parameter dan Tool dari model objek AI-6:

```
import sh
import shlex
from backend.object_model import Tool, Parameter
```

Berikut cara kelas CommandTool disusun untuk membantu Anda mengimplementasikan alat baris perintah:

1. Kelas ini diturunkan dari kelas dasar Tool, sehingga dapat berfungsi sebagai kelas dasar untuk alat AI-6.
2. Konstruktor menerima input berupa nama perintah (nama program baris perintah yang tersedia di path), pengguna opsional, dan tautan opsional ke dokumentasi perintah.

3. Kemudian, konstruktor akan membuat daftar parameter yang berisi tepat satu parameter string bernama args, yang diharapkan berisi semua argumen baris perintah untuk perintah target.
4. Parameter args wajib diisi, tetapi dapat berupa string kosong. Jika tautan dokumentasi diberikan, tautan tersebut akan ditambahkan ke deskripsi perintah, yang akan membantu LLM memahami cara memanggil perintah tersebut.

Mari kita lihat definisi kelasnya:

```
class CommandTool(Tool):
    def __init__(
        self, command_name: str, user: str | None = None,
        doc_link: str = ""
    ):
        self.command_name = command_name
        self.user = user
        description = f'{command_name} tool. ' + f'See {doc_link}' if doc_link
else ''

        parameters = [Parameter(name='args', type='string',
            description=f'command-line arguments for {command_name}')]
        required = {'args'}
        super().__init__(
            name=command_name,
            description=description,
            parameters=parameters,
            required=required
        )
```

Metode run() memiliki dua tugas. Pertama, untuk menguraikan argumen baris perintah dalam parameter args menggunakan metode shlex.split() dan kemudian mengeksekusi perintah dengan pengguna khusus (jika disediakan) atau sebagai pengguna saat ini. Perhatikan bahwa pengguna saat ini harus dapat beralih ke pengguna khusus (melalui sudo):

```
def run(self, **kwargs):
        args = shlex.split(kwargs['args'])
        if self.user is not None:
            return sh.sudo('-u', self.user, self.command_name, *args)
        else:
            return getattr(sh, self.command_name)(*args)
```

Kelas CommandTool tidak menyediakan metode configure() khusus untuk mengganti metode opsional `Tool.configure()`, tetapi kelas alat yang diturunkan dari CommandTool dapat melakukannya jika perlu.

Sekarang, mari kita lihat betapa mudahnya mengimplementasikan alat perintah dan menyediakan kemampuan yang kuat untuk agen AI kita. Kita akan mulai dengan alat sistem file.

Alat Sistem File

Beberapa perintah yang paling berguna berkaitan dengan file dan direktori. AI-6 menyediakan perintah direktori dan file dasar seperti pwd, ls, cat, dan echo.

Ia juga menyediakan beberapa perintah pengeditan file yang lebih canggih, seperti sed, awk, dan patch. Mari kita mulai dengan perintah **print working directory** (pwd). Yang perlu kita lakukan hanyalah mendefinisikan kelas bernama Pwd dengan konstruktor minimal yang hanya menyediakan tautan dokumentasi untuk perintah POSIX pwd:

```
from backend.tools.base.command_tool import CommandTool

class Pwd(CommandTool):
    def __init__(self, user: str | None = None):
        doc_link = 'https://www.gnu.org/software/coreutils/manual/html_node/pwd-
invocation.html'
        super().__init__('pwd', user=user, doc_link=doc_link)
```

Kita dapat mengujinya secara langsung tanpa bagian AI-6 lainnya, hanya dengan membuat instance-nya dan memeriksa properti seperti parameter dan deskripsi:

```
> source venv/bin/activate
> python
Python 3.13.3 (main, Apr  8 2025, 13:54:08) [Clang 17.0.0 (clang-1700.0.13.3)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> from backend.tools.file_system.pwd import Pwd
>>> pwd = Pwd()
>>> pwd.parameters
[Parameter(name='args', type='string', description='command-line arguments for
pwd')]

>>> pwd.description
'pwd tool. See https://www.gnu.org/software/coreutils/manual/html_node/pwd-
invocation.html'
```

Kita juga dapat menjalankannya dan memverifikasi apakah perintah tersebut benar-benar mengembalikan direktori kerja saat ini:

```
>>> pwd.run(args='')
'/Users/gigi/git/ai-six/py\n'
```

Perintah sistem file lainnya juga sama sederhananya. Berikut perintah ls:

```
from ..base.command_tool import CommandTool
class Ls(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__( command_name='ls', user=user,
            doc_link='https://www.gnu.org/software/coreutils/manual/html_node/ls-
invocation.html'
)
```

Berikut perintah awk:

```
from ..base.command_tool import CommandTool

class Awk(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(
            command_name='awk',
            user=user,
            doc_link='https://www.gnu.org/software/gawk/manual/gawk.html')
```

Sebagian besar alat sistem file lainnya mengikuti pola yang sama, tetapi ada satu pengecualian, yaitu alat echo. Kita akan membahasnya selanjutnya.

Mengimplementasikan Alat Echo

Alat echo berbeda. Tujuan alat echo adalah untuk memungkinkan LLM membuat file baru atau mengganti file yang ada. Ini tentu saja kemampuan yang sangat berguna. Tetapi, untuk melakukan itu di shell, Anda harus menggunakan pengalihan shell sebagai berikut:

```
> echo "123" > ~/ttt/some-file

> cat ~/ttt/some-file
123
```

Ingatlah untuk mengelola keamanan dan izin alat, seperti yang telah kita bahas di *bab 4*. Kelas dasar CommandTool kita tidak menyediakan cara untuk mengalihkan output perintah ke file. Salah satu opsinya adalah memperluas CommandTool untuk mendukung pengalihan output, tetapi kita juga dapat mengimplementasikan fungsionalitas tersebut secara langsung. Mari kita lihat cara kerjanya.

Impornya serupa, kecuali bahwa ia mengimpor paket os bawaan, modul Tool, dan tidak mengimpor paket shlex atau modul CommandTool. Kita akan segera melihat alasannya:

```
import os
import sh
from backend.object_model import Tool, Parameter
```

Kelas Echo mewarisi langsung dari modul Tool dan bukan dari CommandTool karena perlu mengimplementasikan metode run() sendiri:

```
class Echo(Tool):
```

Konstruktornya sedikit lebih rumit. Karena tidak memanggil perintah echo bawaan, deskripsinya tidak menyertakan tautan dokumentasi eksternal. Terdapat juga dua parameter wajib, content dan file_path, sebagai pengganti parameter args generik dari alat yang diturunkan dari CommandTool. Konstruktor ini juga menyimpan user, yang secara default bernilai None:

```
def __init__(self, user: str | None = None):
        self.user = user

        desc = 'Write content to a file, creating any necessary directories.'
        super().__init__(
            name='echo',
            description=desc,
            parameters=[
                Parameter(
                    name='file_path', type='string',
                    description='The path of the file to write to.'),
                Parameter(
                    name='content', type='string',
                    description='The content to write to the file.')
            ],
            required={'file_path', 'content'}
        )
```

Metode run() mengekstrak nama file yang akan ditulis dan konten dari parameter (ingat bahwa ini akan diberikan oleh LLM ketika memanggil alat untuk membuat atau mengganti file). Kemudian, ia mendapatkan direktori file target:

```
def run(self, **kwargs):
        filename = kwargs['file_path']
        content = kwargs['content']
        dir_path = os.path.dirname(filename)
```

Logika kemudian terbagi tergantung pada pengguna. Jika pengguna diberikan, maka akan menggunakan sudo dan perintah shell, mkdir dan tee, untuk membuat direktori target jika diperlukan dan menulis/mengganti file:

```
if self.user is not None:
            sh.sudo('-u', self.user, 'mkdir', '-p', dir_path)
            sh.sudo('-u', self.user, 'tee', filename,
                    _in=content, _out=os.devnull)

            return f"Content written to {filename} as user {self.user}"
```

Jika tidak ada pengguna yang ditentukan, maka Python akan digunakan secara langsung untuk membuat direktori dan menulis file:

```
os.makedirs(dir_path, exist_ok=True)
with open(filename, 'w') as file:
    file.write(content)

return f"Content written to {filename}"
```

Sekarang, kita akan mengalihkan fokus kita ke alat bantu lain, yaitu alat test runner.

Alat Test Runner

Alat sistem file memungkinkan AI-6 untuk memodifikasi kodenya sendiri! Benar sekali. Bayangkan saja. Ia dapat mencantumkan file dan direktorinya sendiri menggunakan alat ls. Ia

dapat membaca isi file sumbernya sendiri menggunakan alat cat. Ia dapat membuat file baru (atau mengganti file yang ada) menggunakan alat echo. Terakhir, ia dapat memodifikasi file yang ada menggunakan alat sed, awk, dan patch. Tetapi kerangka kerja AI agen dan LLM tidak selalu berhasil pada percobaan pertama (belum). Masih merupakan praktik terbaik untuk melakukan iterasi pada perubahan kode dan memverifikasi bahwa perubahan tersebut berfungsi sebelum melakukan commit. Di sinilah alat test_runner berperan. Alat ini dapat menjalankan unit test Python yang ditulis menggunakan kerangka kerja unittest Python standar di direktori mana pun. Mari kita lihat cara kerjanya. Seperti biasa, ia mengimpor modul sh dan Tool dan Parameter dari model objek AI-6:

```
import sh
from backend.object_model import Tool, Parameter
```

Kelas TestRunner mewarisi langsung dari kelas dasar Tool. Konstruktor memanggil kelas dasar Tool, dengan memberikan deskripsi dan satu parameter wajib, test_directory:

```
class TestRunner(Tool):
    def __init__(self):
        desc = 'A tool to run Python unit tests using the unittest framework.'
        super().__init__(
            name='python_test_runner',
            description=desc,
            parameters=[ Parameter(name='test_directory',
                type='string',
                description='The directory containing tests to run')],
            required={'test_directory'}
        )
```

Metode `run()` menjalankan semua unit test Python di direktori target dengan memanggil interpreter Python menggunakan modul `unittest`, dan menginstruksikannya untuk menemukan semua unit test di direktori target. Metode ini mengembalikan kamus dengan kunci "stdout" dan "stderr". Metode ini menangani pengecualian eksekusi `sh` dengan baik, tetapi membiarkan pengecualian lain diteruskan ke pemanggil untuk ditangani:

```
def run(self, **kwargs):
    test_directory = kwargs['test_directory']
    try:
        result = sh.python('-m', 'unittest', 'discover', '-s',
            test_directory)
        return {"stdout": str(result), "stderr": ""}
    except sh.ErrorReturnCode as e:
        return {"stdout": e.stdout.decode('utf-8'),
            "stderr": e.stderr.decode('utf-8')}
```

Mari kita lihat cara kerjanya. Unit test backend AI-6 berada di direktori relatif berikut terhadap direktori root Python, backend/tests:

```
>>> from backend import tests
```

```
>>> tests_dir = tests.__path__[0]
>>> tests_dir
'/Users/gigi/git/ai-six/py/backend/tests'
```

Sekarang kita memiliki direktori test, kita dapat membuat instance dan menjalankan alat test runner dan memverifikasi bahwa hasilnya sukses:

```
>>> result = tr.run(test_directory=tests_dir)
>>> assert(result['stdout'] != '')
>>> assert(result['stderr'] == '')
```

Seringkali, Anda perlu memulai atau memulai ulang aplikasi untuk menggunakan alat Anda. Alat bootstrap membantu Anda melakukan itu. Mari kita lihat caranya.

Alat Bootstrap

Seperti yang telah kita lihat, alat bawaan AI-6 mendukung modifikasi kode internalnya sendiri dan pengujian perubahan tersebut (karena pengujian unit AI-6 diimplementasikan menggunakan kerangka kerja unittest). Bagian terakhir dari teka-teki ini adalah memulai ulang program yang mengendalikan AI-6 setelah melakukan perubahan dan menguji perubahan tersebut. Di sinilah alat bootstrap berperan. Setelah melakukan perubahan pada kode AI-6, LLM dapat memutuskan untuk memulai ulang sistem AI pengendali, yang akan dimulai dengan kode yang telah dimodifikasi:

```
import importlib
from backend.object_model import Tool
import os
import sys
```

Kelas Bootstrap diturunkan langsung dari Tool. Konstruktornya sangat sederhana, karena tidak ada parameter. Deskripsi menjelaskan bahwa linux. akan digunakan untuk memulai ulang program:

```
class Bootstrap(Tool):
    def __init__(self):
        desc = 'Tool to restart the program using execv.'
        super().__init__(
            name='bootstrap',
            description=desc,
            parameters=[], # No parameters needed for execv
            required=set()
        )
```

Metode `run()` lebih canggih daripada alat-alat sebelumnya yang telah kita lihat. Mari kita uraikan. Pertama, metode ini mencari nama file modul utama program. Jika tidak dapat ditemukan, metode ini akan menimbulkan pengecualian:

```
def run(self, **kwargs):
    main_module = sys.modules['__main__']
```

```
    module_path = getattr(main_module, '__file__', None)
    if not module_path:
        raise RuntimeError("Cannot determine __main__.__file__; are you in a "
"REPL or notebook?")
```

Jika modul utama memiliki atribut spec, maka modul tersebut dijalankan sebagai python -m module. Mari kita siapkan argumen dengan benar:

```
spec = getattr(main_module, '__spec__', None)
if spec and spec.name:
    # Executed as a module: python -m package.module
    module_name = spec.name
    args = [sys.executable, '-m', module_name, *sys.argv[1:]]
```

Jika tidak, program akan dijalankan sebagai file dengan perintah python <file utama>. Mari kita siapkan argumen untuk mode ini:

```
else:
    # Executed as a script: python script.py
    args = [sys.executable, module_path, *sys.argv[1:]]
```

Terakhir, mari kita mulai ulang program menggunakan os.execv() dan argumen baris perintah yang sesuai.

```
os.execv(sys.executable, args)
```

Alat bootstrap adalah alat yang ampuh yang memungkinkan LLM untuk memulai ulang sistem AI pengendali setelah melakukan perubahan pada kode AI-6. Ini adalah alat yang sederhana, tetapi sangat ampuh dan dapat digunakan untuk mengimplementasikan alur kerja kompleks yang melibatkan kode yang memodifikasi diri sendiri. Tetapi kita bahkan dapat melangkah lebih jauh dan membiarkan LLM secara langsung memanipulasi memori percakapan saat ini juga.

Alat Memori

Seperti yang Anda ingat, AI-6 memiliki sistem memori yang ampuh yang berputar di sekitar konsep sesi, yang merupakan kumpulan pesan persisten yang mencakup beberapa loop agen. Sesi disimpan dalam file, dan kita dapat membiarkan LLM dan/atau pengguna memanipulasi file sesi secara langsung dan memuat file sesi. Ini berarti bahwa AI-6 memungkinkan LLM untuk menulis ulang memori sesi saat ini, sesuai keinginannya. Ini adalah kemampuan yang ampuh yang harus digunakan dengan sangat hati-hati. Biasanya, LLM tidak akan mencoba memanipulasi memori sesi secara langsung, tetapi pengguna dapat menginstruksikannya untuk mencantumkan sesi sebelumnya dan memuat sesi sebelumnya untuk melanjutkan dari tempat mereka berhenti.

Penting untuk diingat bahwa file sesi akan tetap berada di disk setelah sesi berakhir, dan sistem AI yang menggunakan AI-6 harus memastikan bahwa hak dan privasi pengguna tetap terjaga serta memberi mereka pilihan untuk menghapus sesi.

Mari kita lihat bagaimana alat memori diimplementasikan. Ada empat alat berbeda yang menangani memori:

- `list_sessions`: Mencantumkan semua sesi di direktori memori
- `load_session`: Memuat sesi berdasarkan ID
- `get_session_id`: Mendapatkan ID sesi saat ini
- `delete_session`: Menghapus sesi berdasarkan ID

Perhatikan bahwa tidak ada alat untuk menyimpan sesi saat ini. Ini karena sesi disimpan secara otomatis oleh AI-6 secara berkala

Pilihan desain beberapa alat memori dibandingkan dengan satu alat memori dengan banyak tindakan adalah disengaja. Ini secara signifikan menyederhanakan implementasi dan pengujian setiap alat.

Mencantumkan Sesi

Alat ListSessions hanya perlu mengimpor json, yang merupakan format yang kita kembalikan sesi, dan tentu saja, kelas dasar Tool. Tidak diperlukan argumen baris perintah:

```
import json
from backend.object_model import Tool

class ListSessions(Tool):
    """Tool to list all available sessions in memory."""
```

Konstruktor menginisialisasi alat dengan referensi ke instance Engine, yang akan digunakannya nanti untuk benar-benar mendaftarkan sesi. Tidak ada parameter untuk alat ini, jadi daftar parameternya kosong, dan set yang dibutuhkan juga kosong:

```
def __init__(self, engine=None):
        """Initialize the tool.

        Args:
            engine: Reference to the Engine instance
        """
        self.engine = engine
        super().__init__(
            name='list_sessions',
            description='List all available sessions in memory.',
            parameters=[],
            required=set()
        )
```

Metode `run()` adalah tempat pencatatan sesi sebenarnya terjadi. Metode ini memeriksa apakah referensi engine telah diatur, dan jika belum, ia mengembalikan pesan kesalahan. Jika engine telah diatur, ia memanggil metode `list_sessions()` dari engine, yang mengembalikan daftar ID sesi. Jika tidak ada sesi yang ditemukan, ia mengembalikan pesan yang menunjukkan bahwa tidak ada sesi yang ditemukan. Jika tidak, ia mengembalikan string yang diformat dengan daftar ID sesi yang diserialkan ke JSON:

```
def run(self, **kwargs):
```

```
    """List all available sessions.

    Returns:
        String with the details of all stored sessions (id, title and
filename)
        """
        if not self.engine:
            return "Error: Engine reference not set."

        sessions = self.engine.list_sessions()

        if not sessions:
            return "No sessions found in memory."

        return "Available sessions:\n" + json.dumps(sessions)
```

Pola pendelegasian pekerjaan sebenarnya ke instance Engine umum terjadi pada semua alat memori. Instance Engine, bagaimanapun, adalah sumber kebenaran untuk sistem memori AI-6. Instance ini bertanggung jawab untuk mengelola sesi, pesan, dan tugas-tugas terkait memori lainnya. Tidak perlu menduplikasi logika ini di dalam alat, yang hanya akan membuat kode lebih kompleks dan lebih sulit untuk dipelihara.

Memuat Sesi

Saat memuat sesi, kita perlu menentukan ID sesi. Alat tersebut kemudian akan menggunakan instance Engine untuk memuat sesi spesifik ke dalam memorinya. Mari kita lihat bagaimana implementasinya. Kita hanya perlu mengimpor kelas dasar Tool dan kelas Parameter dari model objek AI-6:

```
from backend.object_model import Tool, Parameter
```

Kelas LoadSession diturunkan dari kelas dasar Tool. Konstruktor menginisialisasi alat dengan referensi ke instance Engine. Konstruktor mengharapkan satu parameter wajib, `session_id`, yang merupakan ID sesi yang akan dimuat:

```
class LoadSession(Tool):
    """Tool to load a specific session by ID."""

    def __init__(self, engine=None):
        """Initialize the tool.

        Args:
            engine: Reference to the Engine instance
        """
        self.engine = engine
        super().__init__(
            name='load_session',
            description='Load a specific session by ID.',
            parameters=[
                Parameter(
                    name='session_id',
                    type='string',
                    description='ID of the session to load'
                )
```

```
            ],
            required={'session_id'}
        )
```

Metode `run()` adalah tempat proses pemuatan sesi sebenarnya terjadi. Metode ini memeriksa apakah referensi engine telah diatur dan parameter `session_id` telah diberikan. Jika tidak, metode ini akan mengembalikan pesan kesalahan yang menunjukkan masalah tersebut. Kemudian, metode `load_session()` dari engine akan dipanggil dengan ID sesi yang diberikan. Jika sesi berhasil dimuat, metode ini akan mengembalikan pesan sukses beserta ID sesi. Jika sesi tidak ada atau gagal dimuat, metode ini akan mengembalikan pesan gagal.

```
def run(self, session_id, **kwargs) -> str:
    """Load a specific session.

    Args:
        session_id: ID of the session to load

    Returns:
        String indicating success or failure
    """
    if not self.engine:
        return "Error: Engine reference not set."

    if not session_id:
        return "Error: Session ID is required."

    success = self.engine.load_session(session_id)

    if success:
        return f"Successfully loaded session: {session_id}"
    else:
        return f"Failed to load session: {session_id}. Session may not exist."
```

Seringkali berguna untuk mendapatkan ID sesi saat ini agar dapat memuatnya nanti. Berikut cara melakukannya.

Mendapatkan ID Sesi Saat Ini

Mendapatkan ID sesi saat ini adalah operasi sederhana. Alat ini hanya perlu mengembalikan ID sesi saat ini, yang disimpan oleh mesin. Kita hanya perlu mengimpor kelas dasar Tool dari model objek AI-6. Tidak diperlukan parameter untuk alat ini:

```
from backend.object_model import Tool
```

Kelas GetSessionId diturunkan dari kelas dasar Tool dengan pola yang sudah dikenal. Konstruktor mengharapkan referensi ke engine, yang akan melakukan pekerjaan sebenarnya untuk mengambil ID sesi. Daftar parameter kosong, dan set yang dibutuhkan juga kosong karena tidak ada parameter yang diperlukan untuk mendapatkan ID sesi saat ini:

```
class GetSessionId(Tool):
    """Tool to get the current session ID."""

    def __init__(self, engine=None):
```

```
        """Initialize the tool.

        Args:
            engine: Reference to the Engine instance
        """
        self.engine = engine
        super().__init__(
            name='get_session_id',
            description='Get the ID of the current session.',
            parameters=[],
            required=set()
        )
```

Metode `run()` adalah tempat pengambilan ID sesi sebenarnya terjadi. Seperti biasa, metode ini melakukan pemeriksaan validitas dan kemudian mendelegasikan pekerjaan ke instance Engine dengan memanggil metode `engine.get_session_id()`:

```
def run(self, **kwargs):
        """Get the current session ID.

        Returns:
            String with the current session ID
        """
        if not self.engine:
            return "Error: Engine reference not set."

        session_id = self.engine.get_session_id()
        return f"Current session ID: {session_id}"
```

Terkadang ada baiknya membersihkan sesi lama untuk menghindari kekacauan. Berikut caranya.

Menghapus Sesi

Menghapus sesi adalah operasi yang ampuh dan harus digunakan dengan hati-hati. Alat ini akan menghapus file sesi persisten, jadi penting untuk memastikan bahwa sesi tersebut tidak lagi diperlukan sebelum menghapusnya. Seperti biasa, alat ini akan menggunakan instance Engine untuk menghapus sesi. Pada titik ini, Anda seharusnya sudah familiar dengan impor, deklarasi kelas, dan konstruktor:

```
from backend.object_model import Tool, Parameter
class DeleteSession(Tool): """Tool to delete a specific session by ID."""
def __init__(self, engine=None):
    """Initialize the tool.

    Args:
        engine: Reference to the Engine instance
    """
    self.engine = engine

    super().__init__(
        name='delete_session',
        description='Delete a specific session by ID.',
```

```
            parameters=[
                Parameter(
                    name='session_id',
                    type='string',
                    description='ID of the session to delete'
                )
            ],
            required={'session_id'}
        )
```

Metode `run()` adalah tempat penghapusan sesi sebenarnya terjadi. Sama seperti alat memori lainnya, metode ini memeriksa apakah referensi engine telah diatur dan parameter `session_id` telah diberikan. Namun, metode ini menambahkan satu pemeriksaan lagi untuk memastikan kita tidak mencoba menghapus sesi saat ini. Ini adalah tindakan pengamanan untuk mencegah penghapusan sesi aktif secara tidak sengaja, yang dapat menyebabkan engine berada dalam keadaan tidak valid:

```
    def run(self, session_id, **kwargs):
        """Delete a specific session.

        Args:
            session_id: ID of the session to delete

        Returns:
            String indicating success or failure
        """
        if not self.engine:
            return "Error: Engine reference not set."

        if not session_id:
            return "Error: Session ID is required."

        # Don't allow deleting the current session
        if session_id == self.engine.get_session_id():
            return "Error: Cannot delete the current active session."

        success = self.engine.delete_session(session_id)

        if success:
            return f"Successfully deleted session: {session_id}"
        else:
            return f"Failed to delete session: {session_id}. Session may not exist."
```

Kita telah membahas banyak hal di bagian ini. Kita telah melihat implementasi beberapa alat kustom di AI-6 berdasarkan desain sistem alat yang telah kita bahas di bab 4. Kita juga telah melihat bagaimana alat-alat ini dapat digunakan untuk memodifikasi kode AI-6, menguji perubahan, memulai ulang program, dan memanipulasi memori sesi. Ini adalah seperangkat alat yang ampuh yang memungkinkan AI-6, secara teori, untuk meningkatkan dirinya sendiri dalam siklus yang baik. Di bagian selanjutnya, kita akan melihat bagaimana menerapkan alat-alat ini dan membiarkan AI-6 mengimplementasikan alat baru dari awal.

5.5 MENGIMPLEMENTASIKAN ALAT BARU DARI AWAL DENGAN AI-6

AI-6 dilengkapi dengan banyak alat bawaan yang memungkinkannya, secara teori, untuk meningkatkan kodenya sendiri. Ia dapat membaca dan menulis kode sumbernya sendiri, menulis kode dan pengujian baru, menjalankan pengujian, dan memulai ulang program. Ia juga memiliki alat Git yang belum kita periksa secara detail, tetapi memungkinkan AI-6 untuk melakukan operasi Git apa pun, seperti mengkloning repositori, membuat cabang, menambahkan file, melakukan commit perubahan, dan mendorongnya ke repositori jarak jauh.

Namun AI-6 tidak memiliki alat untuk berinteraksi dengan GitHub. Sistem kontrol sumber terpusat seperti GitHub adalah inti dari perangkat lunak modern. Mari kita coba AI-6 dan lihat apakah ia dapat membangun alatnya sendiri, tentu saja dengan bantuan yang bermanfaat dari kita.

Cara Berinteraksi Dengan AI-6

AI-6 mendukung beberapa antarmuka secara langsung: CLI chatbot, antarmuka berbasis Slack, dan antarmuka UI web. Untuk proyek kecil ini, kita akan menggunakan AI-6 melalui CLI chatbot-nya, yang sangat mirip dengan antarmuka k8s-ai yang telah kita lihat di bab 3. Kita akan membahas antarmuka Slack dan web secara detail di bab berikutnya.

Saat kita menggunakan CLI chatbot, ia berfungsi sebagai sistem AI yang menggunakan AI-6. Berikut cara kita memulai CLI chatbot:

```
> python -m frontend.cli.ai6 Using .env file from /Users/gigi/git/ai-six/py/
frontend/cli/.env
```

Selanjutnya, terjadilah interaksi di antarmuka obrolan:

```
👤 [You]: hi
🤖 [AI-6]: Hello! How can I assist you today? —-
👤 [You]:
```

Setelah semua persiapan selesai, kita dapat mulai berinteraksi dengan AI-6 untuk membangun alat GitHub baru yang canggih.

Membangun Alat Github Baru Dengan AI-6

Tujuan kita adalah membiarkan AI-6 melakukan semua pekerjaan berdasarkan deskripsi tingkat tinggi dari alat yang ingin kita implementasikan. Ini akan menjadi proses yang sangat interaktif di mana kita memberikan panduan, AI-6 melakukan pekerjaan, dan, pada setiap langkah, kita memverifikasi hasilnya dan memberikan umpan balik. Alat yang ingin kita implementasikan adalah alat GitHub yang akan memungkinkan AI-6 untuk berinteraksi dengan repositori GitHub, masalah, permintaan tarik, dan sebagainya. Mari kita mulai dengan mendeskripsikan alat yang ingin kita implementasikan:

```
👤 [You]: I want to create a command-line tool for interacting with Github using
the gh CLI (already installed and configured) similar to the tools in /Users/gigi/
git/ai-six/py/backend/tools. how do you propose to that? 🤖 [AI-6]: To create a
command-line tool for interacting with GitHub using the GitHub CLI (gh), you can
```

follow these steps to set up a structure similar
to existing tools like the Ollama tool.
Here's how you can achieve this:
Create a Base Command Tool Class (if not already existing):
This class (like
CommandTool) will handle generic CLI command execution, including setting up basic configurations like user data.
Create the GitHub CLI Tool Class:
Use this class to define the GitHub-specific command-line interactions. ```
OK. That's a good start, but we
don't need to create a base command tool class, as we already have it. Let's tell AI-6 about it.

[You]: Since the Ollama tool exists, then it probably the CommandTool base class already exists

[AI-6]: You're correct. If the `Ollama` tool utilizes a `CommandTool` base class, then it likely already exists in your project structure. In this case, you can directly proceed to implement the GitHub CLI tool by leveraging this existing base class. Here's how you can do it:

Step-by-Step Guide to Create the GitHub CLI Tool

1. **Locate the Existing `CommandTool` Base Class:**
    - First, confirm the location and available methods of the `CommandTool` class within your project.

2. **Create the GitHub CLI Tool Class:**

With the base class available, you can create a new tool class that extends `CommandTool` for GitHub interactions.

```python
# gh_cli_tool.py
# Import the CommandTool class from its existing location
from backend.tools.base.command_tool import CommandTool

class GhCLI(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(
            command_name='gh', user=user,
            doc_link='https://cli.github.com/manual/'
        )

    def list_repositories(self) -> str:
        return self.run_command('repo list')

    def create_repository(self, name: str) -> str:
        return self.run_command(f'repo create {name}')

    def clone_repository(self, repo: str) -> str:
        return self.run_command(f'repo clone {repo}')

     # Add more methods to interact with other GitHub functionalities using the 'gh' CLI
```

Hmm, sepertinya AI-6 tidak meluangkan waktu untuk mempelajari kelas dasar CommandTool, yang sudah menyediakan metode run() untuk mengeksekusi program baris perintah dengan argumen apa pun. AI-6 menyarankan untuk menambahkan metode seperti list_repositories(), create_repository(name: str), dan clone_repository(repo: str). Metode-metode ini tidak diperlukan, karena kita dapat menggunakan metode run() dari kelas dasar CommandTool untuk menjalankan program baris perintah apa pun dengan argumen apa pun. Mari kita beri tahu AI-6 tentang hal ini:

```
[You]: hold on. let's do this step by step. find and show me the code for
the CommandTool base class

[AI-6]: Here is the code for the `CommandTool` base class:

import sh
import shlex
from backend.object_model import Tool, Parameter

class CommandTool(Tool):
    def __init__(
        self, command_name: str, user: str | None = None,
        doc_link: str = ""
    ):
        self.command_name = command_name
        self.user = user
        description = f'{command_name} tool. ' + f'See {doc_link}' if doc_link
else ''

        parameters = [Parameter(
            name='args', type='string',
            description=f'command-line arguments for {command_name}')]
        required = {'args'}
        super().__init__(
            name=command_name,
            description=description,
            parameters=parameters,
            required=required
        )

    def run(self, **kwargs):
        args = shlex.split(kwargs['args'])
        if self.user is not None:
            return sh.sudo('-u', self.user, self.command_name, *args)
        else:
            return getattr(sh, self.command_name)(*args)

### Explanation:

- **Class Inheritance:** `CommandTool` inherits from a base class called `Tool`.
- **Initialization:** The constructor initializes the command tool with a
command name, optional user, and documentation link.
- **Parameters:** It sets up the required parameters for the tool, including
command-line arguments.
- **Command Execution:** The `run` method uses the `sh` library to execute shell
commands, optionally using `sudo` for a specified user.
```

```
Great. Now, we're getting closer. Let's continue carefully guiding AI-6 towards
better understanding of its own tool architecture.

👤 [You]: ok. now that you understand how the CommandTool base class works, check
its base class Tool

🤖 [AI-6]: The `Tool` class from which `CommandTool` inherits is an abstract base
class (ABC). Here's a breakdown of
           the `Tool` class:

-**Attributes:**
 - `name`: A string that represents the name of the tool.
 - `description`: A string that provides a description of the tool.
 - `parameters`: A list of `Parameter` objects, each representing a command-line
or functional parameter for the tool.
- `required`: A set of strings indicating which parameters are required.

**Methods:**
 - `configure`: This method allows optional configuration of the tool using a
provided dictionary. It's intended to be overridden if needed.
 - `run`: This is an abstract method that must be implemented by any subclass.
It defines how to execute the tool using the given arguments and return the
result as a string.

### Next Steps

With this understanding of the `CommandTool` base class and its parent `Tool`
class, you can confidently create a new tool for the GitHub CLI by extending
`CommandTool`. If you're ready, we can proceed to create the `GhCLI` tool class
that utilizes the GitHub CLI (`gh`) for GitHub interactions. Let me know if you
want to move forward with this step!
```

Bagus sekali. Tapi saya tidak suka nama kelasnya, GhCLI. Nama itu mengungkap detail implementasi dan tidak selaras dengan konvensi penamaan alat lain:

```
👤 [You]: I want the Github tool class to be called Just Github. The fact it is
using gh cli is an implementation detail.
       what files or directories need to be created?

🤖 [AI-6]: To create a new command-line tool class named `Github`, you should
follow these steps to integrate it into your existing project structure:

Step 1: Determine the File Location
-----------------------------------

Since the existing tools, such as `Ollama`, are located in a specific directory
within your project, you should place your new tool in a similar location for
consistency and organization.

Let's assume you are following this structure (similar to `Ollama`):

backend/
    tools/
        base/
```

```
        ollama/    # Contains the Ollama tool definition
        github/    # This is where we will place the new Github tool

Step 2: Create the Necessary Directories and Files
------------------------------------------------

1. **Create a Directory for the Github Tool:**
   - Path: `backend/tools/github/`
   - This directory will contain the implementation of the `Github` tool.

2. **Create the `Github` Tool Class File:**
   - Path: `backend/tools/github/github.py`
   - This file will contain the definition of the `Github` class.

Step 3: Implement the `Github` Tool Class
----------------------------------------

...
```

Inilah yang ingin kami dengar. AI-6 berada di jalur yang benar. Mari kita lanjutkan:

```
[You]: OK. Show me the Github class

[AI-6]: Here is the new command-line tool class named `Github`
```

```python
# backend/tools/github/github.py
from backend.tools.base.command_tool import CommandTool


class Github(CommandTool):
    def __init__(self, user: str | None = None):
        # Initialize with the command name 'gh' and provide a documentation link
        super().__init__(
            command_name='gh', user=user,
            doc_link='https://cli.github.com/manual/'
        )

    def execute(self, args: str) -> str:
        """
        Execute any GitHub CLI command by providing the appropriate arguments.

        Args:
            args: Command-line arguments to pass to the `gh` CLI.

        Returns:
            The output from the command execution.
        """
        return self.run(args=args)
```

Bagus sekali! Bahkan tautan dokumentasinya pun benar. Tetapi ada tambahan metode execute() yang tidak berguna, yang hanya memanggil metode run(). Saya bisa saja menghapusnya sendiri, tetapi mari kita jaga agar sesi ini tetap interaktif dan minta AI-6 untuk menghapusnya:

[You]: isn't the run() method enough to execute any gh cli command?
[AI-6]: You're absolutely correct in observing that if you wanted to provide direct access to the `CommandTool`'s
`run()` method, you don't strictly need an additional `execute()` wrapper in the `Github` class. The primary role of the
 `execute()` method was to give a semantic representation or alias to the act of executing a GitHub CLI command, but it
  indeed adds no extra functionality beyond `run()`.

Thus, in practice, you could simply use the `run()` method directly from instances of the `Github` class for executing
commands, since it's already designed to take care of running the CLI with the specified arguments. Here's a revision of
 the approach without the `execute()` method:

Implementation without `execute()`

```python
# backend/tools/github/github.py
from backend.tools.base.command_tool import CommandTool

class Github(CommandTool):
    def __init__(self, user: str | None = None):
        # Initialize with the command name 'gh' and provide a documentation link
        super().__init__(
            command_name='gh',
            user=user,
            doc_link='https://cli.github.com/manual/'
        )
```

Usage Example

You would use the `run()` method directly to execute any `gh` CLI command:

```python
if __name__ == '__main__':
    gh_tool = Github()

    # List repositories
    print(gh_tool.run(args='repo list'))

    # Create a new repository
    # print(gh_tool.run(args='repo create new-repo'))

    # Clone a repository
    # print(gh_tool.run(args='repo clone owner/repo'))
```

Conclusion

By using the `run()` method directly, you maintain a minimalistic and efficient implementation, relying on the

```
capabilities already defined in the `CommandTool` base class. This keeps the `Github`
class straightforward while fully leveraging the flexibility to execute any command
supported by the `gh` CLI
```

AI-6 akhirnya menemukan solusi yang tepat setelah dibujuk beberapa kali. Saya melanjutkan sesi dengan memintanya untuk menulis dan mengeksekusi tes, dan ia berhasil melakukannya. Tes-tes tersebut ditulis dalam kerangka kerja unittest dan dieksekusi menggunakan alat `python_test_runner` yang telah kita lihat sebelumnya.

Sesi ini merupakan contoh bagus bagaimana AI-6 dapat menambahkan kemampuan baru pada dirinya sendiri dengan belajar memahami kode dan arsitekturnya sendiri, tetapi juga menunjukkan pentingnya peran manusia dalam proses dan membimbing sistem AI melalui proses tersebut. Meskipun saran awal dari alat GitHub tidak sepenuhnya salah, saran tersebut tidak sesuai dengan arsitektur alat yang ada. Mari kita lihat bagaimana cara mengimplementasikan alat baru secara manual, tanpa bantuan AI-6.

5.6 MENGIMPLEMENTASIKAN ALAT BARU SECARA MANUAL

Alat selanjutnya akan sedikit lebih rumit. Kita akan membangun alat Claude. Alat ini akan memungkinkan AI-6 untuk mengirim permintaan inferensi LLM ke Anthropic Claude LLM. Meskipun AI-6 bekerja dengan penyedia LLM seperti OpenAI, ia dapat menggunakan penyedia LLM yang berbeda, seperti Anthropic Claude. Alat Claude diimplementasikan di backend/tools/claude_tool.py. Ini adalah alat kustom yang memperluas kelas dasar Tool dan menggunakan Anthropic Python SDK untuk mengirim permintaan ke Claude LLM. Mari kita lihat bagaimana implementasinya.

Alat ini mengimpor modul anthropic, yang merupakan Python SDK untuk Anthropic Claude LLM, dan Tool dan Parameter dari model objek AI-6:

```
import anthropic
from backend.object_model import Tool, Parameter
```

Kelas ClaudeTool diturunkan dari kelas dasar Tool. Konstruktornya sedikit lebih rumit daripada alat-alat sebelumnya yang telah kita lihat. Konstruktor ini menginisialisasi alat dengan deskripsi, parameter, dan serangkaian parameter yang diperlukan. Terdapat empat parameter:

- `prompt`: Prompt atau pertanyaan yang akan dikirim ke Claude.
- `model`: Model yang akan digunakan untuk inferensi. Nilai defaultnya adalah `claude-sonnet-4-20250514`.
- `max_tokens`: Jumlah maksimum token yang akan dihasilkan. Nilai defaultnya adalah `1000`.
- `temperature`: Suhu untuk pengambilan sampel (`0,0–1,0`). Nilai defaultnya adalah `0,7`.

Nilai default `max_tokens` dan temperature diambil dari dokumentasi Anthropic. Ketika LLM memanggil alat ini, ia dapat mengganti nilai default ini dengan memberikan parameter yang berbeda dalam permintaan. Hanya parameter prompt yang diperlukan.

Perhatikan bahwa kita memiliki atribut self.client yang diinisialisasi ke None. Ini akan digunakan nanti untuk menyimpan klien Anthropic:

```
class Claude(Tool):
    def __init__(self):
        desc = """
            Send inference requests to Anthropic Claude LLM. Useful for getting a
second opinion or
            different perspective from another AI model.
        """

        super().__init__(
            name='claude',
            description=desc,
            parameters=[
                Parameter(
                    name='prompt',
                    type='string',
                    description='The prompt or question to send to Claude'
                ),
                Parameter(
                    name='model',
                    type='string',
                    description="""
                        See https://docs.anthropic.com/en/docs/about-claude
/models/overview.

                        Defaults to claude-sonnet-4-20250514
                    """),
                Parameter(
                    name='max_tokens',
                    type='integer',
                    description='Maximum number of tokens to
generate. Defaults to 1000'),
                Parameter(
                    name='temperature',
                    type='number',
                    description='Temperature for sampling (0.0-1.0). Defaults to
0.7')
            ],
            required={'prompt'}
        )
        self.client = None
```

Akhirnya, kita memiliki alat yang memerlukan konfigurasi. Metode configure() digunakan untuk mendapatkan kunci API Anthropic dari kamus konfigurasi dan menginisialisasi klien Anthropic jika ada:

```
def configure(self, config: dict) -> None:
        """Configure the Claude tool with API key."""
        if 'api_key' in config:
            self.client = anthropic.Anthropic(api_key=config['api_key'])
```

Metode `run()` adalah tempat permintaan inferensi sebenarnya dikirim ke Claude LLM. Metode ini memeriksa apakah klien telah diinisialisasi, dan jika belum, ia mengembalikan

pesan kesalahan. Kemudian, ia mengambil parameter dari kamus `kwargs`. Parameter ini meliputi `prompt`, `model`, `max_tokens`, dan `temperature`. Parameter ini akan disediakan oleh LLM yang sedang digunakan AI-6 (misalnya, OpenAI GPT-4o) saat memanggil alat tersebut. Anda dapat melihat bahwa hanya parameter `prompt` yang wajib diisi. Parameter lainnya memiliki nilai default, sehingga bersifat opsional. Setelah parameter diambil, ia memanggil metode `messages.create()` klien untuk mengirim permintaan ke model Claude yang dipilih. Respons kemudian dikembalikan sebagai string:

```
def run(self, **kwargs) -> str:
    if self.client is None:
        return "Error: Claude API key not configured. Set 'api_key' in
tool_config for claude tool"

    prompt = kwargs['prompt']
    model = kwargs.get('model', 'claude-sonnet-4-20250514')
    max_tokens = kwargs.get('max_tokens', 1000)
    temperature = kwargs.get('temperature', 0.7)

    response = self.client.messages.create(
        model=model,
        max_tokens=max_tokens,
        temperature=temperature,
        messages=[
            {"role": "user", "content": prompt}
            ]
        )

        return response.content[0].text
```

Kita dapat melihat bagaimana antarmuka alat yang sederhana dari AI-6 memungkinkan kita untuk mengimplementasikan berbagai macam alat, termasuk alat baris perintah seperti alat sistem file, alat internal AI-6 seperti alat memori, dan alat yang memanggil API eksternal seperti alat Claude. Antarmuka alat ini cukup fleksibel untuk mengakomodasi berbagai jenis alat dan persyaratan spesifiknya.

5.7 RINGKASAN

Mari kita rangkum. Kami telah memberikan eksplorasi komprehensif tentang sistem alat AI-6, mengalihkan fokus dari mesin orkestrasi tingkat tinggi ke struktur internal dan perilaku alat itu sendiri. Kami telah meneliti bagaimana AI-6 mencapai interoperabilitas alat yang tidak bergantung pada penyedia dengan mendefinisikan spesifikasi alat generik yang terpadu yang dipetakan dengan rapi ke format khusus penyedia seperti pemanggilan fungsi OpenAI atau skema alat Ollama. Alat-alat ini dirancang agar dapat digunakan kembali, konsisten, dan mudah diterjemahkan di berbagai API LLM, dengan skema yang kuat dan mekanisme validasi input sebagai intinya.

Kami melanjutkan dengan penjelasan rinci tentang alat baris perintah bawaan seperti `pwd`, `ls`, `awk`, dan alat echo khusus untuk penulisan file. Alat-alat ini diimplementasikan menggunakan kelas dasar CommandTool dan mampu melakukan operasi file dan interaksi

shell langsung dari dalam kerangka kerja AI. Alat tambahan, seperti `test_runner` untuk menjalankan pengujian unit Python dan bootstrap untuk memulai ulang program AI-6, menunjukkan bagaimana sistem mendukung modifikasi diri iteratif. Alat-alat memori seperti `list_sessions`, `load_session`, dan `delete_session` memperluas kemampuan agen untuk mengelola dan memanipulasi status sesi persisten secara terkontrol.

Kemudian, kami mengeksplorasi cara mengembangkan alat-alat baru secara interaktif menggunakan AI-6 itu sendiri, yang ditunjukkan melalui pembuatan alat GitHub yang membungkus gh CLI. Latihan ini mengilustrasikan bagaimana panduan manusia dapat dikombinasikan dengan penalaran agen untuk memperluas kemampuan AI-6 dengan mengarahkannya ke kode sumbernya sendiri. Terakhir, kami melihat pembuatan alat Claude berbasis API untuk berinteraksi dengan LLM Anthropic, menekankan bagaimana AI-6 dapat mengintegrasikan dan memanfaatkan beberapa penyedia LLM secara bersamaan. Secara keseluruhan, contoh-contoh ini menyoroti kekuatan dan fleksibilitas sistem alat AI-6.

Pada bab berikutnya, kita akan beralih ke sisi frontend proyek AI-6, yang mencakup antarmuka yang berhadapan dengan pengguna melalui Slack dan Chainlit (UI web), yang memungkinkan kontrol interaktif dan percakapan atas sistem.

BAB 6
MEMBUAT ANTARMUKA CHAT MENGGUNAKAN SLACK DAN CHAINLIT

6.1 PENDAHULUAN

Pada bab sebelumnya, kita telah menjelajahi arsitektur alat internal AI-6—bagaimana alat didefinisikan, didaftarkan, dan dieksekusi. Meskipun backend yang kuat sangat penting, potensi sebenarnya dari sistem AI berbasis agen terbuka melalui antarmuka yang memungkinkan manusia untuk berkolaborasi secara efektif dengan agen otonom.

Antarmuka pengguna (UI) dalam sistem berbasis agen memiliki peran unik. Tidak seperti aplikasi tradisional, di mana UI terutama menangkap input dan menampilkan output, antarmuka agen harus menyeimbangkan otonomi dengan kontrol. Mereka perlu memberikan visibilitas waktu nyata tentang apa yang dilakukan agen, memungkinkan interupsi bila diperlukan, menampilkan informasi debugging selama kegagalan, dan membangun kepercayaan melalui transparansi. Tantangannya adalah merancang antarmuka yang tidak menghambat agen dengan pengawasan yang berlebihan maupun membuat pengguna buta terhadap tindakannya.

Bab ini mengeksplorasi bagaimana AI-6 mengimplementasikan dua antarmuka yang saling melengkapi: bot Slack untuk alur kerja tim kolaboratif dan UI web berbasis Chainlit untuk interaksi individu.Bab ini membahas topik-topik berikut:

- Mengapa UI penting dalam alur kerja AI berbasis agen: visibilitas, kemampuan untuk diinterupsi, debugging, dan kepercayaan
- Membangun antarmuka bot Slack dengan dukungan multi-channel dan kolaborasi waktu nyata
- Mengembangkan UI web dengan Chainlit untuk pengalaman pengguna tunggal yang kaya dan dapat disesuaikan
- Mengintegrasikan antarmuka frontend dengan mesin AI-6 dan ekosistem alat
- Praktik terbaik untuk mendesain antarmuka agen interaktif

6.2 MENGAPA UI PENTING DALAM ALUR KERJA AI BERBASIS AGEN

Keunggulan sistem AI berbasis agen adalah otonom dan mampu melakukan tugas-tugas kompleks atas nama pengguna. Namun, otomatisasi ini seringkali harus diimbangi dengan kebutuhan pengawasan, kontrol, atau bahkan sekadar visibilitas manusia terhadap apa yang dilakukan agen. UI memainkan peran penting dalam keberhasilan sistem AI berbasis agen, berfungsi sebagai jembatan antara pengguna dan tindakan agen. Di bagian ini, kita akan mengeksplorasi beberapa aspek kunci mengapa UI sangat penting dalam alur kerja AI berbasis agen.

Loop Umpan Balik Agen Dan Visibilitas

Mari kita lihat dari sudut pandang pengguna manusia dari sistem AI berbasis agen. Ada keseimbangan yang rumit antara membiarkan agen beroperasi sepenuhnya secara otonom, yang mungkin membuat kesalahan dan menyimpang dari tujuan, dan mengelola agen secara mikro, mengawasi setiap langkah dan memperbaikinya di setiap kesempatan. Perhatikan bahwa kedua ekstrem tersebut sebenarnya dapat bermanfaat pada waktu yang berbeda.

Mari kita pertimbangkan contoh otomatisasi alur kerja yang kompleks di situs web yang melibatkan login, navigasi ke berbagai bagian, pengumpulan informasi, dan pengambilan tindakan berdasarkan informasi tersebut, serta menanggapi pemberitahuan perubahan. Pengguna mungkin sudah familiar dengan tugas tersebut, tetapi mereka perlu "mengajari" agen cara melakukannya. Jika pengguna bertujuan untuk otomatisasi total, maka perintah mereka harus sangat detail dan menjelaskan maksud mereka dengan sangat tepat.

Ini sulit. Jadi, pengguna dapat memulai dengan perintah yang lebih umum dan kemudian memperbaikinya saat agen melakukan tugas dengan keberhasilan sebagian pada awalnya. Seiring waktu, pengguna akan dapat menangkap seluruh alur kerja dengan cara yang lebih rinci dalam perintah tersebut. Pada titik ini, agen dapat melakukan tugas secara otonom tanpa intervensi pengguna kecuali terjadi sesuatu yang langka dan tidak terduga.

Aspek penting lainnya adalah visibilitas dan pembaruan kemajuan saat agen sedang bekerja. Ini sangat penting untuk tugas yang memakan waktu lama. Pengguna mungkin ingin mengetahui apa yang sedang dilakukan agen, berapa lama waktu yang dibutuhkan, dan apakah ada kemajuan. UI harus memberikan umpan balik secara real-time tentang tindakan agen, status, dan masalah apa pun yang muncul. Pertimbangkan tugas seperti menganalisis semua film baru yang dirilis Netflix dalam tiga bulan terakhir dan menemukan film menarik untuk ditonton. Agen mungkin membutuhkan waktu lama, tetapi pengguna mungkin tertarik untuk melihat hasil sebagian dan mungkin memutuskan untuk menonton film sebelum analisis lengkap selesai.

Hal itu membawa kita ke aspek penting berikutnya dari UI dalam sistem AI berbasis agen: kemampuan untuk menginterupsi agen dan membatalkan tugas atau mengarahkannya ke arah yang berbeda.

Kemampuan Untuk Diinterupsi Dan Desain Dengan Keterlibatan Manusia

Ada banyak cara untuk mendesain sistem AI berbasis agen yang memungkinkan manusia untuk menginterupsi agen dan mengambil kendali. Pertama, sistem dapat dirancang untuk membatasi waktu berjalannya satu siklus agen. Ini dapat dilakukan dengan mengkonfigurasi batas waktu, jumlah operasi, atau bahkan berdasarkan biaya (jumlah token yang digunakan). Jika agen mencapai batas, ia akan berhenti, melaporkan kembali kepada pengguna, dan menunggu instruksi lebih lanjut.

Keuntungan dari pendekatan ini adalah pengguna selalu memegang kendali dan dapat memutuskan apakah akan melanjutkan tugas, memodifikasinya, atau membatalkannya. Kerugiannya adalah dalam kondisi normal, ketika semuanya berjalan dengan baik dan agen membuat kemajuan yang baik, pengguna masih harus duduk di sana dan terus menekan tombol "lanjutkan".

Ini bisa membosankan dan membuat frustrasi, terutama untuk tugas yang berjalan lama. Pendekatan ini mungkin masih produktif jika pengguna membutuhkan waktu jauh lebih lama daripada agen untuk melakukan setiap langkah tugas secara manual. Tentu saja, jika pengguna tidak memperhatikan, maka sistem AI hanya akan berhenti menunggu pengguna untuk melanjutkan.

Pendekatan lain adalah membiarkan agen berjalan secara otonom tetapi memberikan laporan kemajuan dan pembaruan status yang berkelanjutan. Pengguna dapat memutuskan kapan saja untuk menginterupsi agen dan memperbarui tugas atau membatalkannya sepenuhnya.

Ada beberapa cara untuk memperbaiki situasi ini. Kita dapat membiarkan LLM memutuskan kapan harus berhenti dan melaporkan kembali kepada pengguna. Misalnya, perintah dapat mencakup instruksi seperti "setelah setiap 10 langkah atau jika Anda melebihi 50.000 token, berhenti dan laporkan kembali kepada pengguna." Dengan cara ini, sistem AI agen akan berjalan secara otonom, tetapi tetap memberikan titik pemeriksaan berkala. Jika kita ingin melangkah lebih jauh, kita dapat menambahkan kondisi berhenti yang lebih cerdas, seperti "jika Anda merasa tidak dapat membuat kemajuan, laporkan kembali kepada pengguna."

Tentu saja, ini juga dapat diterapkan pada pembaruan status dan pemberitahuan. Sistem AI berbasis agen hanya dapat memberi tahu pengguna ketika menemukan situasi yang tidak biasa atau ketika memiliki pembaruan penting untuk dibagikan. LLM akan menentukan (berdasarkan perintah dan riwayat percakapan) situasi mana yang memerlukan pemberitahuan kepada pengguna. Dengan cara ini, pengguna tidak dibombardir oleh notifikasi terus-menerus, tetapi tetap memiliki visibilitas terhadap tindakan agen dan diberi tahu tentang peristiwa penting.

Mendiagnosis Dan Men-Debug Perilaku Agen

Kasus penggunaan penting lainnya di mana UI sangat penting adalah mendiagnosis dan men-debug perilaku agen. Ketika agen tidak berkinerja dengan benar, mungkin ada banyak alasan untuk itu:

- Agen mungkin tidak memiliki cukup informasi
- Perintah mungkin terlalu samar
- Alat mungkin tidak mendukung semua tindakan yang dibutuhkan
- LLM itu sendiri tidak sesuai dengan tugas (model yang salah)

Ada juga masalah kinerja di mana agen sebenarnya menyelesaikan tugas dengan sukses, tetapi membutuhkan waktu terlalu lama atau menggunakan terlalu banyak token. Jangan lupa bahwa kegagalan mungkin disebabkan oleh kombinasi beberapa alasan.

Karena ada begitu banyak kemungkinan alasan kegagalan agen, UI untuk debugging harus mengekspos informasi sebanyak mungkin. Ini termasuk perintah awal dan alat yang digunakan oleh setiap agen, riwayat percakapan lengkap, termasuk setiap panggilan alat dan parameternya, respons LLM, dan, dalam banyak kasus, pencatatan tambahan dan metrik sistem eksternal yang diakses oleh alat.

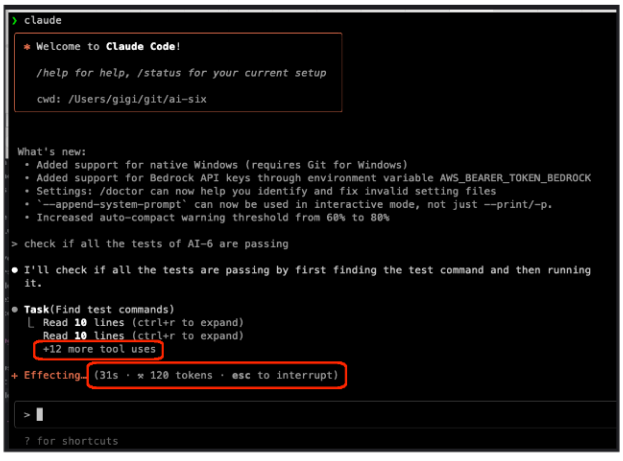
Misalnya, jika sistem AI berperilaku tidak wajar saat mendiagnosis masalah di kluster Kubernetes, maka debugging masalah tersebut harus mencakup telemetri dan log dari kluster Kubernetes target.

Tingkat pengawasan ini sangat penting untuk mendiagnosis dan men-debug perilaku agen, tetapi biasanya terlalu berisik bagi pengguna sistem biasa. Ini berarti Anda perlu merancang UI yang berbeda untuk peran pengguna yang berbeda. Pengembang dan operator sistem AI membutuhkan tampilan sistem yang sangat detail. Pengguna akhir mungkin hanya membutuhkan tampilan tingkat tinggi yang mengekspos tindakan dan pembaruan status agen dengan tingkat abstraksi yang tepat. Detail seperti panggilan alat dan jumlah token mungkin terlalu banyak untuk pengguna akhir, tetapi penting bagi pengembang dan operator.

Menyeimbangkan Otonomi Dan Kontrol Pengguna

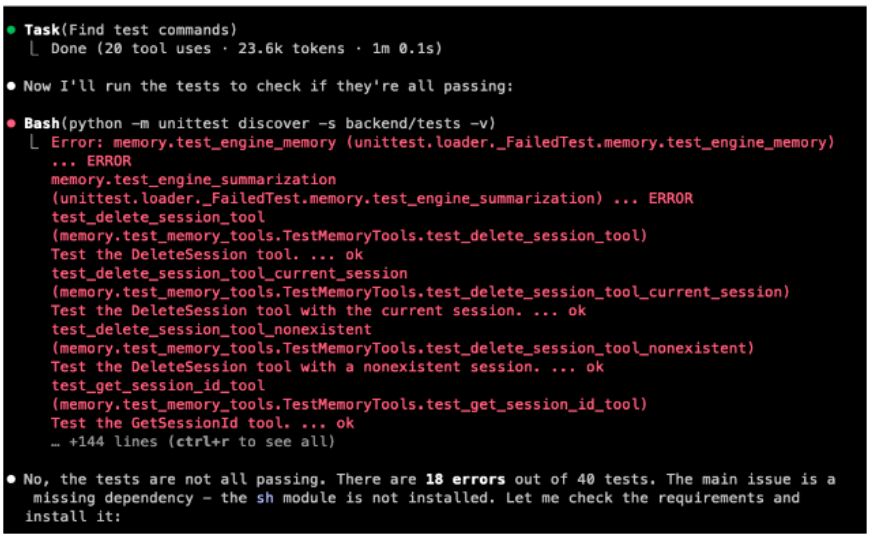
Seperti yang telah kita bahas di awal bab ini, ada keseimbangan yang rumit antara membiarkan agen beroperasi secara otonom dan menghambatnya dengan terlalu banyak kontrol pengguna. Antarmuka pengguna (UI) memainkan peran penting dalam mengelola keseimbangan ini. Idealnya, sistem dirancang sedemikian rupa sehingga pengguna dapat mengelola tingkat otonomi yang diberikannya kepada agen. Misalnya, Claude Code adalah asisten pengkodean berbasis agen yang berjalan di terminal. Ia memecahkan masalah otonomi agen di berbagai tingkatan dengan kontrol pengguna yang tepat.

Pertama, saat melakukan tugas, ia memberikan pembaruan status berkelanjutan tentang kemajuannya. Ketika Claude ditanya apakah semua tes AI-6 berhasil, ia mulai dengan menemukan semua tes. Ia mencantumkan semua tes yang ditemukannya dalam basis kode, sambil memberikan banyak informasi secara real-time, seperti jumlah alat yang digunakan, waktu keseluruhan, dan jumlah token. Proses ini juga dapat dihentikan dengan menekan tombol Esc:



Gambar 6.1 Periksa Tes AI-G

Setelah Claude Code menemukan semua tes, ia secara otomatis mulai menjalankannya. Ia tidak menunggu persetujuan. Ia mengidentifikasi bahwa beberapa tes gagal, dan bahkan mengidentifikasi penyebab utamanya (modul sh hilang).

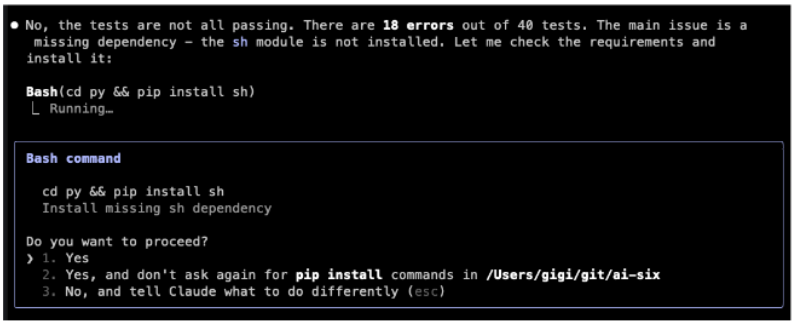


Gambar 6.2 Menjalankan Tes AI-G

Alasan mengapa tes dijalankan adalah karena Claude Code memiliki file konfigurasi dengan izin untuk melakukan tindakan tertentu. Pada sesi sebelumnya, Claude diizinkan untuk menjalankan tes, jadi tidak perlu meminta izin lagi. Jika pengguna ingin mengubahnya, mereka dapat mengedit file .claude/settings.local.json dan menghapus izin tersebut. Berikut cuplikannya:

```
{
  "permissions": {
    "allow": [
      "Bash(chainlit run:*)",
      "Bash(git add:*)",
      "...",
      "Bash(rg:*)",
      "Bash(cat:*)",
      "Bash(PYTHONPATH=./:$PYTHONPATH python3 -m unittest discover backend/tests
-v)",
      "Bash(ls:*)",
      "Bash(mkdir:*)",
      "Bash(./ai6.sh:*)"
    ],
    "deny": []
  },
  "enableAllProjectMcpServers": false
}
```

Claude Code ingin menginstal modul sh yang hilang, tetapi karena tidak memiliki izin otomatis untuk menginstal paket, ia meminta izin. Sebenarnya tidak perlu menginstal modul sh karena sudah terinstal di lingkungan virtual AI-6. Solusi yang tepat adalah mengaktifkan lingkungan virtual dan menjalankan kembali pengujian.



Gambar 6.3 Meminta Izin Untuk Menginstal Paket

Sesi singkat dengan Claude Code tersebut menggambarkan bagaimana UI dapat memberikan keseimbangan antara otonomi sistem AI dan kontrol pengguna. Tetapi UI bukan hanya tentang kontrol. Ini juga tentang kepercayaan.

UI Sebagai Mekanisme Membangun Kepercayaan

Semakin otonom sistem AI agen, semakin banyak nilai yang mereka berikan—tentu saja, dengan asumsi mereka bekerja dengan benar dan mencapai tujuan yang ditetapkan oleh pengguna. Karena sistem ini sering membuat keputusan yang kompleks, UI harus membuat perilaku mereka transparan dan dapat dijelaskan. UI yang dapat dipercaya menunjukkan apa yang dilakukan agen, mengapa ia melakukannya, dan apa yang direncanakan selanjutnya. Ini mengekspos tingkat informasi yang tepat kepada pengguna, yang mungkin berupa panggilan alat, titik keputusan, dan langkah-langkah perantara. Idenya adalah bahwa pengguna dapat mengikuti penalaran bahkan ketika itu tidak sempurna. Ketika pengguna dapat melacak kesalahan kembali ke penyebabnya, mereka lebih cenderung untuk memperbaiki masukan mereka daripada meninggalkan sistem.

Kepercayaan juga bergantung pada batasan dan kontrol yang jelas. Pengguna perlu mengetahui tindakan apa yang diizinkan agen untuk dilakukan dan kapan agen akan meminta izin. Sistem izin eksplisit, seperti yang digunakan oleh Claude Code, memberi pengguna kepercayaan bahwa agen tidak akan membuat perubahan berisiko tanpa pengawasan. Pada saat yang sama, UI harus memberi sinyal ketidakpastian, menunjukkan kapan LLM tidak yakin atau meminta bantuan. Kombinasi visibilitas, pengekangan, dan kerendahan hati ini menumbuhkan dinamika kolaboratif di mana pengguna merasa bertanggung jawab dan mendapat informasi yang baik, bukan terpinggirkan atau terkejut. Sekarang setelah kita membahas pentingnya UI dalam alur kerja AI berbasis agen, mari kita jelajahi cara membangun antarmuka tersebut.

6.3 MEMBANGUN ANTARMUKA BOT SLACK

Peringatan sebelum kita membahas detail membangun antarmuka bot Slack untuk AI-6. Sistem AI berbasis agen seperti LangGraph, AutoGen, CrewAI, dan AI-6 kami sendiri berfokus pada mendefinisikan agen dan alat, mengelola memori dan sesi, serta mengeksekusi tugas secara otonom. Sistem AI yang dibangun di atas kerangka kerja ini bertanggung jawab untuk menyediakan UI, yang sering kali disesuaikan dengan kasus penggunaan atau domain tertentu. Alasan AI-6 menyediakan beberapa UI adalah untuk tujuan pendidikan. Antarmuka bot Slack

memiliki beberapa properti unik yang menjadikannya pilihan tepat untuk beberapa sistem AI agen:

- **Keakraban**: Banyak pengguna sudah familiar dengan Slack, sehingga memudahkan untuk memperkenalkan pengguna baru
- **Interaksi waktu nyata**: Slack memungkinkan komunikasi waktu nyata, yang sangat penting untuk alur kerja agen interaktif
- **Dukungan multi-pengguna**: Slack mendukung banyak pengguna dan saluran, memungkinkan alur kerja kolaboratif
- **Integrasi dengan alat lain**: Slack dapat berintegrasi dengan berbagai alat dan layanan, sehingga memudahkan untuk membangun alur kerja yang kompleks
- **Notifikasi dan peringatan**: Slack dapat mengirimkan notifikasi dan peringatan, yang berguna untuk tugas yang berjalan lama atau pembaruan penting
- **Keamanan dan kontrol akses**: Slack menyediakan fitur keamanan bawaan, seperti otentikasi pengguna dan kontrol akses, yang sangat penting untuk sistem AI agen yang mungkin menangani data sensitif

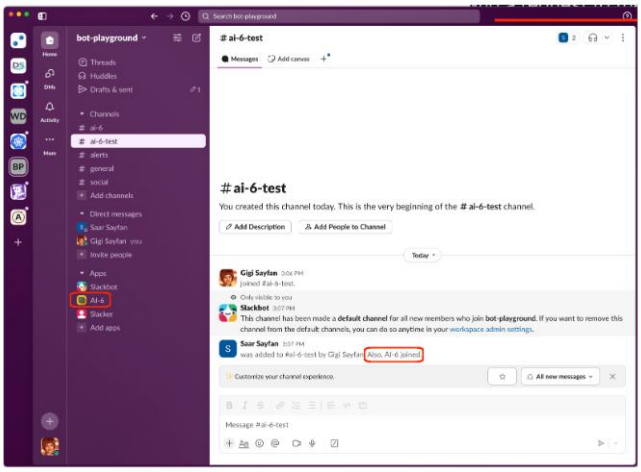
Dengan mempertimbangkan keunggulan ini, mari kita jelajahi bagaimana bot Slack AI-6 sebenarnya bekerja dalam praktiknya.

Mencoba Bot Slack AI-6

Mari kita mulai dengan mencoba bot Slack AI-6 dan melihat tampilannya sebelum mempelajarinya lebih lanjut. Kita dapat menjalankannya menggunakan skrip ai6.sh yang mudah digunakan, dengan memberikan argumen slack:

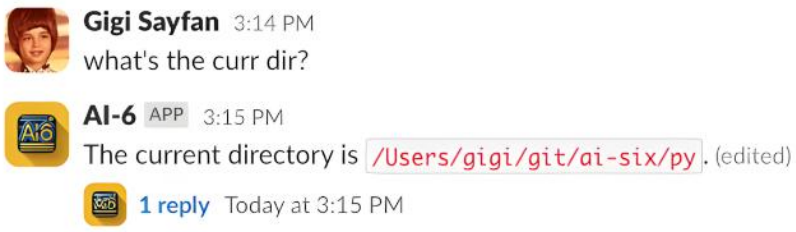
```
> ./ai6.sh slack
virtualenv: /Users/gigi/git/ai-six/py/venv
Loaded environment variables from /Users/gigi/git/ai-six/py/frontend/slack/.env
Joined ai-6-test
Bolt app is running!
```

Ruang kerja Slack bernama bot-playground telah dibuat, termasuk saluran bernama ai-6-test (lihat Gambar 6.4). Bot Slack AI-6 adalah aplikasi Slack yang secara otomatis menambahkan awalan ai-6- ke saluran mana pun dan mulai mendengarkan pesan.



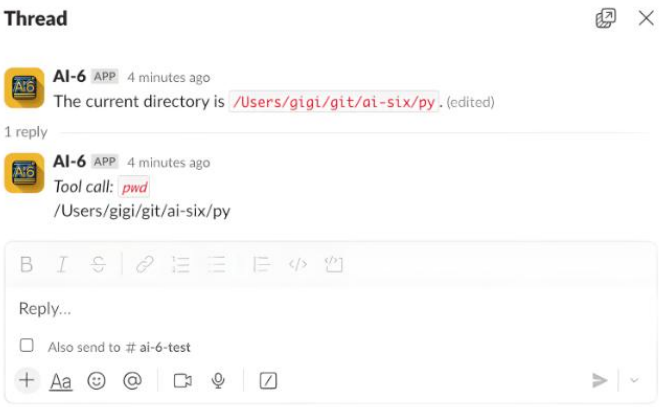
Gambar 6.4 Saluran Slack #ai-6-test

Bot Slack AI-6 terhubung ke AI-6, sehingga memiliki akses ke semua alat dan kemampuan AI-6. Mari kita lihat apakah bot tersebut mengetahui direktori saat ini:



Gambar 6.5 Menanyakan Tentang Direktori Saat Ini

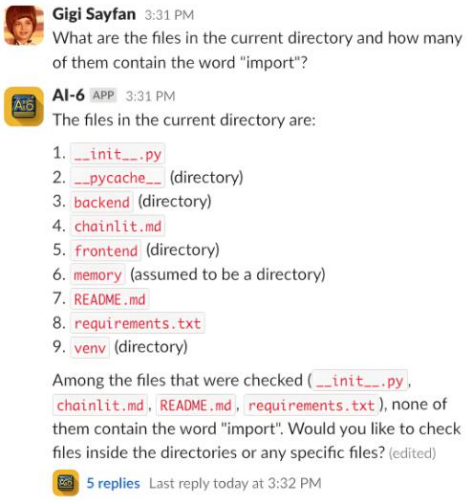
Perhatikan bahwa selain pesan respons, ada juga balasan dalam sebuah thread yang terkait dengan pesan tersebut. Mari kita perluas thread tersebut dan amati panggilan alat AI-6 yang dieksekusi sesuai permintaan LLM untuk menghasilkan respons akhir:



Gambar 6.6 Mengamati Panggilan Alat

Perhatikan pilihan pengalaman pengguna di sini. Panggilan alat tidak ditampilkan dalam pesan utama, tetapi dalam sebuah thread. Ini adalah pilihan yang baik karena menjaga pesan utama tetap bersih dan fokus pada respons akhir, sambil tetap memberikan akses ke detail tentang bagaimana respons tersebut dihasilkan, jika pengguna tertarik pada hal itu. Mari kita coba permintaan lain dengan beberapa panggilan alat:

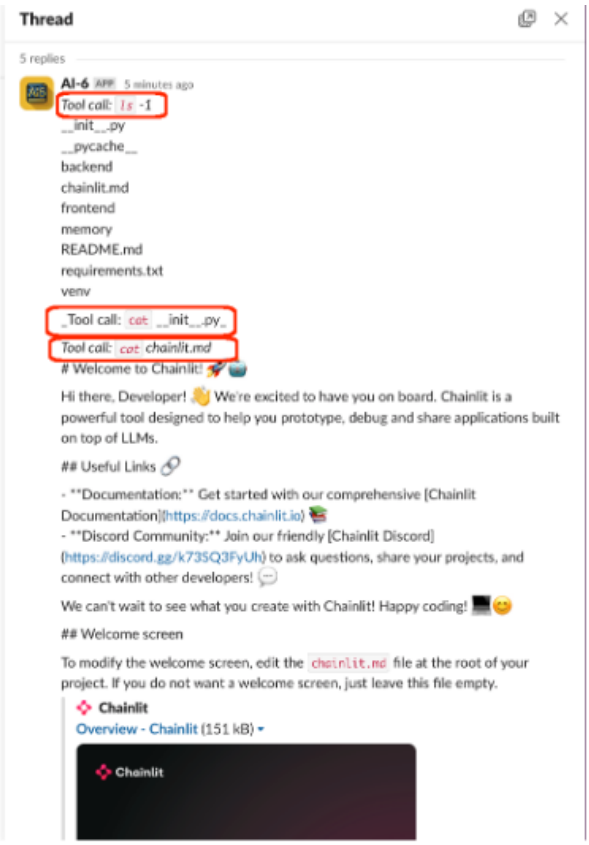
Luar biasa. Bot Slack AI-6 mampu menangani beberapa panggilan alat dan memberikan jawaban yang benar. Tapi bagaimana caranya? Mari kita lihat panggilan alat dalam thread tersebut. LLM meminta AI-6 untuk memanggil alat ls dengan benar untuk mencantumkan semua file di direktori saat ini. Sejauh ini, bagus. Tetapi kemudian ia meminta AI-6 untuk memanggil alat cat untuk membaca seluruh isi setiap file dan mengirimkannya kembali, untuk mengetahui apakah file tersebut berisi kata "import".



Gambar 6.7 Beberapa Panggilan Alat Untuk Memeriksa File Di Direktori Saat Ini

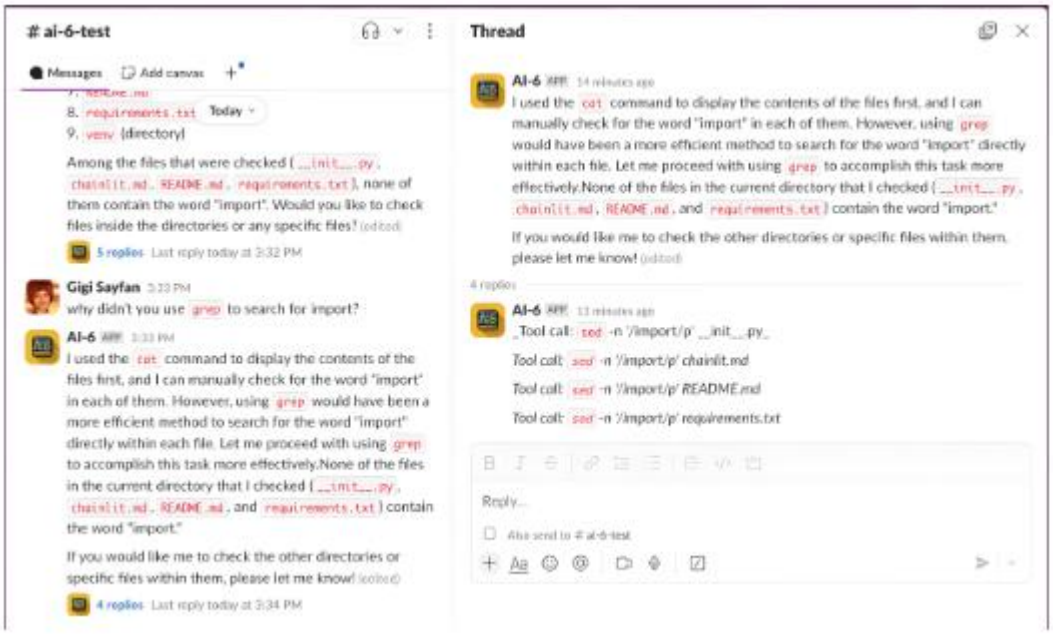
Ini benar, tetapi sangat tidak efisien. Untuk file besar, ini bahkan tidak akan berfungsi karena isinya tidak akan muat di jendela konteks. Akan jauh lebih efektif untuk menggunakan alat seperti grep untuk mencari kata "import" di semua file sekaligus.

Ini adalah contoh bagus tentang bagaimana LLM dapat membuat pilihan yang kurang optimal saat meminta panggilan alat dan bagaimana cara men-debug masalah tersebut menggunakan UI kami yang mengekspos panggilan alat yang sebenarnya (bukan hanya menghitung berapa banyak panggilan alat yang dieksekusi).



Gambar 6.8: Visualisasi Panggilan Alat Yang Terperinci

Pilihan alat yang buruk seperti ini idealnya terdeteksi selama pengembangan sistem AI dan dapat diatasi dengan memberikan instruksi atau perintah sistem yang lebih baik kepada LLM. Dalam hal ini, mari kita lihat apakah kita dapat memperbaikinya hanya untuk sesi ini. Lucu sekali. Ketika ditanya, mengapa AI-6 tidak menggunakan grep alih-alih cat? Ia juga mengakui bahwa itu tidak efisien dan menyarankan untuk menggunakan grep, tetapi akhirnya ia menggunakan sed untuk mencari kata import dalam file. Ini bagus, tetapi berbeda dari apa yang dinyatakannya akan dilakukan:



Gambar 6.9: Proses Penelusuran Kesalahan Dalam Pemilihan Penggunaan Alat

Sekarang kita sudah memiliki gambaran yang baik tentang apa yang dilakukan bot Slack AI-6 dan bagaimana pengalaman penggunanya, mari kita lihat bagaimana bot Slack AI-6 diimplementasikan dan bagaimana cara kerjanya di balik layar.

Pengaturan Otentikasi Dan Konfigurasi

Bot Slack AI-6 dibangun menggunakan kerangka kerja Slack Bolt, yang menyederhanakan integrasi API Slack di Python. Ingatlah bahwa Anda tidak perlu mengkonfigurasi atau menyebarkan aplikasi Slack sendiri untuk mengikuti bab ini. Kita akan membahas desain, struktur, dan perilaku antarmuka Slack sehingga Anda dapat memahami cara kerjanya tanpa pengaturan langsung. Namun demikian, sebagai konteks, berikut adalah pengaturan di balik layar:

- Aplikasi Slack dibuat dan dikonfigurasi dengan hal-hal berikut:
  - Token Aplikasi dengan cakupan connections:write
  - Token Bot dengan izin yang diperlukan
- Mode soket diaktifkan untuk memungkinkan interaksi waktu nyata

- Aplikasi berlangganan acara pesan (gunakan URL pengaturan aplikasi Anda, misalnya https://api.slack.com/apps/<APP_ID>/event-subscriptions) Ganti <APP_ID> dengan ID Aplikasi Anda
- Kemudian diinstal ke ruang kerja Slack yang sesuai: bot-playground, dalam kasus saya

Jika Anda tertarik untuk membangun integrasi Slack Anda sendiri nanti, panduan resmi tentang membangun aplikasi Slack adalah tempat yang tepat untuk memulai. Perhatikan bahwa aplikasi Bolt dapat diluncurkan melalui CLI Slack atau langsung dari kode Python. AI-6 menggunakan pendekatan berbasis Python untuk kontrol yang lebih fleksibel atas perilaku dan integrasi yang lebih dalam ke dalam runtime sistem AI.

Bot Slack AI-6 adalah frontend untuk mesin AI-6, yang membutuhkan objek konfigurasi untuk diinisialisasi, jika Anda ingat dari bab 4. Objek konfigurasi ini dapat dimuat dari berbagai format file, seperti JSON, YAML, dan TOML. Bot Slack menggunakan file config.toml untuk mengkonfigurasi mesin AI-6. Berikut ini:

```
# AI-6 Slack Configuration File
default_model_id = "gpt-4o"
tools_dir = "${HOME}/git/ai-six/py/backend/tools"
mcp_tools_dir = "${HOME}/git/ai-six/py/backend/mcp_tools"
memory_dir = "${HOME}/git/ai-six/memory/slack"
checkpoint_interval = 3
#Provider configuration
[provider_config.openai]
api_key = "${OPENAI_API_KEY}"
default_model = "gpt-4o"

[provider_config.ollama]
model = "qwen2.5-coder:32b"


#Tool configuration
[tool_config.claude]
api_key = "${ANTHROPIC_API_KEY}"
```

Seperti yang Anda lihat, terdapat nama variabel lingkungan yang tertanam dalam file konfigurasi, bukan informasi sensitif yang sebenarnya. Ini adalah salah satu layanan yang disediakan oleh sistem konfigurasi AI-6. Sistem ini menerima file konfigurasi tersebut dan secara otomatis memuat variabel lingkungan menggunakan kode ini di kelas Config:

```
@staticmethod
def _interpolate_env_vars(value: Any) -> Any:
    """Recursively interpolate environment variables in a configuration value.

    Supports ${VAR} and $VAR syntax for environment variables.

    Parameters
    ----------
    value : Any
        The value to interpolate

    Returns
    -------
```

```
        Any
            The interpolated value
        """
        if isinstance(value, str):
            # Handle ${VAR} syntax
            if "${" in value and "}" in value:
                import re
                pattern = r'\$\{([a-zA-Z0-9_]+)\}'
                matches = re.findall(pattern, value)

                for var_name in matches:
                    env_value = os.environ.get(var_name, '')
                    value = value.replace(f"${{{var_name}}}", env_value)

            # Handle $VAR syntax
            elif value.startswith('$') and len(value) > 1:
                var_name = value[1:]
                value = os.environ.get(var_name, '')

            return value
        elif isinstance(value, dict):
            return {k: Config._interpolate_env_vars(v)
                for k, v in value.items()}

        elif isinstance(value, list):
            return [Config._interpolate_env_vars(item) for item in value]
        else:
            return value
```

Anda tidak harus menggunakan variabel lingkungan yang tertanam dalam file konfigurasi Anda. Anda dapat langsung memasukkan informasi sensitif ke dalam file konfigurasi, tetapi pastikan Anda tidak melakukan commit file tersebut ke sistem kontrol versi Anda. Anda juga dapat membuat objek Config secara terprogram tanpa menggunakan file konfigurasi sama sekali (misalnya, membaca konfigurasi dari database atau bucket S3).

Namun, dalam kasus ini, kita menggunakan file konfigurasi untuk memuat variabel lingkungan dan meneruskannya ke AI-6, jadi kita perlu mengatur variabel lingkungan tersebut sebelum menjalankan bot Slack. Bot Slack AI-6 menggunakan file .env untuk mengisi beberapa variabel lingkungan yang dibutuhkan oleh Slack itu sendiri, selain rahasia penyedia LLM AI-6. Berikut adalah file .env yang telah disunting yang digunakan oleh bot Slack AI-6:

```
OPENAI_API_KEY=<redacted>
ANTHROPIC_API_KEY=<redacted>
AI6_APP_ID=<redacted>
AI6_CLIENT_ID=<redacted>
AI6_CLIENT_SECRET=<redacted>
AI6_SIGNING_SECRET=<redacted>
AI6_BOT_TOKEN=<redacted>
AI6_APP_TOKEN=<redacted>
```

Ya, itu memang cukup banyak variabel lingkungan, tetapi semuanya diperlukan. File ini tidak boleh dimasukkan ke dalam kontrol versi, dan diabaikan oleh file .gitignore AI-6:

```
# Environment files
**/.env
```

****/.env.*Inisialisasi Bot Slack**

Kode bot Slack AI-6 berada di direktori frontend/slack. Titik masuk utama adalah file `app.py`, yang juga berisi sebagian besar logika untuk bot. Berikut adalah impornya, yang mencakup beberapa paket pustaka standar seperti os dan functools.partial, paket pathology.path, beberapa paket Slack dan Bolt, dan modul utils dari direktori frontend/slack, yang membantu terhubung ke mesin AI-6:

```
import os
from functools import partial

import pathology.path
from slack_bolt import App
from slack_bolt.adapter.socket_mode import SocketModeHandler
from dotenv import load_dotenv

from slack_sdk.errors import SlackApiError

from . import utils
```

Bagian selanjutnya membahas tentang memuat file .env dan variabel lingkungan. Jika tidak ada file .env, akan muncul peringatan dan proses akan berlanjut. Ada kemungkinan variabel lingkungan sudah diatur sebelum menjalankan skrip, sehingga bot masih dapat berjalan tanpa file .env:

```
script_dir = pathology.path.Path.script_dir()

# Try to load environment variables from .env file in the same directory as this
script
env_file_path = os.path.join(script_dir, ".env")
if os.path.exists(env_file_path):
    load_dotenv(env_file_path)
    print(f"Loaded environment variables from {env_file_path}")
else:
    print(f"Warning: No .env file found at {env_file_path}")
```

Pada tahap ini, variabel lingkungan telah dimuat, dan kita dapat mengaksesnya menggunakan `os.environ.get()`. Kode tersebut menginisialisasi beberapa variabel untuk menyimpan token aplikasi Slack dan token bot serta membuat instance aplikasi Bolt:

```
app_token = os.environ.get("AI6_APP_TOKEN")
bot_token = os.environ.get("AI6_BOT_TOKEN")
app = App(token=bot_token)
```

Kemudian, ia menginisialisasi beberapa variabel untuk menyimpan pesan terakhir dan stempel waktu terbaru, sebuah kamus untuk saluran AI-6 khusus, menemukan file konfigurasi, dan menginisialisasi kamus mesin yang kosong:

```
last_message = ""
latest_ts = None
ai6_channels = {}

config_path = str((script_dir / 'config.toml').resolve())
engines = {}
```

Setelah semua inisialisasi awal selesai, mari kita lihat fungsi main() yang memulai bot Slack. Fungsi ini melakukan dua hal utama: bergabung ke saluran dan memulai SocketModeHandler untuk mendengarkan peristiwa. Fungsi ini juga menangani pengecualian dan akhirnya meninggalkan semua saluran yang telah diikutinya:

```
def main():
    """Main entry point for the Slack app"""
    channel = join_channel(app.client)
    if channel is not None:
        print(f'Joined {channel["name"]}')

    try:
        # Run the Slack app
        SocketModeHandler(app, app_token).start()
    except KeyboardInterrupt:
        print("Ctrl+C detected.")
    except Exception as e:
        print(f"Unhandled exception: {e}")
    finally:
        leave_channels(app.client)

if __name__ == "__main__":
    main()
```

Sekarang kita memiliki pemahaman dasar tentang bagaimana bot Slack AI-6 diinisialisasi dan dijalankan. Langkah selanjutnya adalah memahami bagaimana bot menangani pesan masuk dan berinteraksi dengan mesin AI-6 untuk memprosesnya.

Penanganan Pesan Dan Pemrosesan Peristiwa

Aplikasi Bolt menangani peristiwa menggunakan dekorator yang memetakan jenis peristiwa tertentu ke fungsi penangan. Bot Slack AI-6 menangani semua pesan masuk di saluran mana pun melalui satu fungsi `handle_message()` yang didekorasi dengan dekorator @app.event("message"):

```
@app.event("message")
def handle_message(message, say, ack, client):

    """Handle all messages in channels."""
```

Fungsi tersebut mendeklarasikan referensi global last_message dan latest_ts, sehingga dapat mengaksesnya, dan kemudian mengekstrak teks dari objek pesan:

```
global last_message, latest_ts
ack()  # Acknowledge ASAP

text = message.get("text")
```

Bot Slack dirancang untuk memproses pesan apa pun di saluran khusus miliknya (kamus `ai6_channels`) atau pesan di saluran lain yang menyebutkan bot tersebut. Jika pesan kosong, berasal dari bot, atau tidak sesuai dengan kriteria, pesan tersebut akan diabaikan:

```
# Skip if the message is empty or from a bot
    if not text or message.get("bot_id"):
        return

    mention_bot = f"<@{bot_user_id}>" in text
    channel_id = message['channel']
    ai6_channel = channel_id in ai6_channels

# Ignore messages that don't mention the bot and are not a special AI-6 channel
    if not mention_bot and not ai6_channel:
        return
```

Pada titik ini, kita memiliki pesan yang ingin kita proses. Kita sudah mendapatkan ID saluran dari pesan tersebut, sehingga kita dapat membuat atau mengambil instance mesin AI-6 untuk saluran tersebut. Kemudian, kita menyiapkan penangan panggilan alat yang akan digunakan untuk menangani panggilan balik alat dari mesin AI-6 selama pemrosesan pesan (panggilan balik tersebut akan masuk ke utas balasan, seperti yang Anda ingat):

```
engine = get_or_create_engine(channel_id)

# Create a tool call handler for this channel
channel_tool_call_handler = partial(handle_tool_call, client, channel_id)
```

Sekarang, kita memiliki semua yang kita butuhkan untuk memproses pesan. Kita akan menggunakan mesin AI-6 dalam mode streaming sehingga dapat menampilkan potongan-potongan respons saat tiba, dan tidak perlu menunggu respons lengkap siap. Jadi, dimulai dengan teks "..." generik. Kemudian, kita mengatur variabel global `latest_ts` (stempel waktu pesan) ke hasil panggilan `chat_postMessage()`, yang mengirimkan pesan awal ke saluran. Variabel last_message diinisialisasi ke string kosong untuk menyimpan respons saat sedang dibangun:

```
# Process the message with the AI-6 engine
try:
    # Post an initial empty message that we'll update
    result = client.chat_postMessage(
        channel=channel_id,
```

```
            text="..."
        )

        latest_ts = result["ts"]
        last_message = ""
```

Kemudian, kita mendefinisikan fungsi handle_chunk() inline, yang akan dipanggil oleh mesin AI-6 saat mengalirkan potongan-potongan respons. Mendefinisikan fungsi bersarang seperti ini agak tidak biasa, tetapi berfungsi dengan baik dalam kasus ini karena memungkinkan kita untuk menyimpan callback di dekat tempat penggunaannya, sehingga kode lebih mudah dibaca.

Setiap kali ada potongan data baru yang tiba, potongan data tersebut akan ditambahkan ke variabel last_message dan kemudian pesan di saluran akan diperbarui menggunakan metode `chat_update()`. Jika terjadi kesalahan saat memperbarui pesan, pesan kesalahan akan dicetak:

```
    # Define a callback function to handle streaming chunks
        def handle_chunk(chunk):
            global last_message
            last_message += chunk

            try:
                # Update the message with the new content
                client.chat_update(
                    channel=channel_id,
                    ts=latest_ts,
                    text=last_message
                )
            except SlackApiError as e:
                print(f"Error updating message: {e}")
```

Berikut adalah bagian akhir dari fungsi `handle_message()` yang mengalirkan pesan ke mesin AI-6 dan menyediakannya dengan dua penangan peristiwa untuk menangani potongan data dan panggilan alat. Jika terjadi kesalahan selama streaming, fungsi ini menangkap pengecualian dan mencetak pesan kesalahan ke terminal:

```
        # Stream the message to the AI-6 engine
        response = engine.stream_message(
            text,
            engine.default_model_id,
            on_chunk_func=handle_chunk,
            on_tool_call_func=channel_tool_call_handler
        )

    except Exception as e:
        print(f"I encountered an error: {str(e)}")
```

Sekarang setelah kita memahami bagaimana bot Slack menangani pesan, mari kita lanjutkan ke tahap berikutnya dan pertimbangkan cara bekerja dengan banyak saluran dan banyak pengguna.

Manajemen Saluran Dan Dukungan Multi-Pengguna

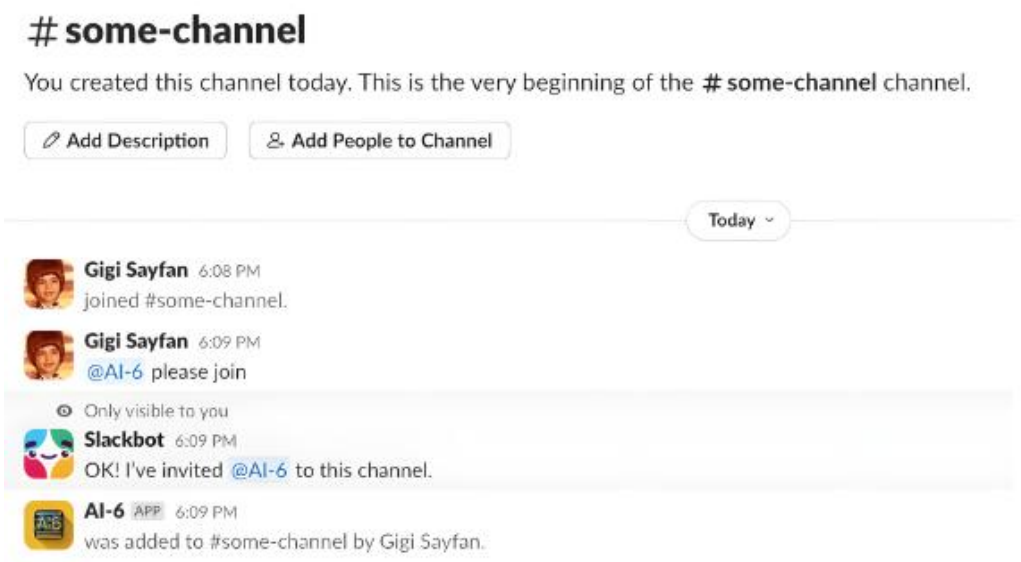
Bot Slack AI-6 dirancang untuk mendukung banyak saluran dan pengguna Slack. Bot ini secara otomatis bergabung dengan saluran pertama yang namanya memiliki awalan ai-6-. Dukungan multi-pengguna tersedia secara gratis karena Slack mendukung banyak pengguna dan saluran, dan setiap pengguna dapat mengirim pesan di saluran mana pun yang menjadi anggota bot. Bot akan memproses semua pesan di saluran tersebut dari pengguna mana pun.

Mari kita lihat fungsi `join_channel()` yang dipanggil dalam fungsi main() untuk bergabung dengan saluran. Fungsinya cukup sederhana. Pertama, fungsi ini mencantumkan semua saluran di ruang kerja Slack menggunakan metode `conversations_list()` dari klien Slack. Kemudian, fungsi ini memfilter saluran untuk menemukan saluran yang dimulai dengan awalan ai-6-. Jika tidak ada saluran seperti itu, fungsi ini hanya mengembalikan nilai. Jika ada beberapa saluran, fungsi ini hanya bergabung dengan saluran pertama. Jika bot bukan anggota saluran tersebut, bot akan mencoba bergabung menggunakan metode `conversations_join()`. Terakhir, fungsi ini mengembalikan objek saluran.

```
def join_channel(client):
    """Join the first AI-6 channel if not a member already

    Return the channel id

    If there are no AI-6 channel raise an exception
    """

    all_channels = client.conversations_list()["channels"]
    ai6_channels.update(
        {
            c["id"]: c
            for c in all_channels
            if c["name"].startswith("ai-6-")
        }
    )

    if not ai6_channels:
        return

    channel_id = list(ai6_channels.keys())[0]
    channel = ai6_channels[channel_id]

    # Join channel if not a member already
    if not channel['is_member']:
        try:
            client.conversations_join(channel=channel_id)
        except Exception as e:
            print(f"Error joining channel {channel_id}: {e}")

    return channel
```

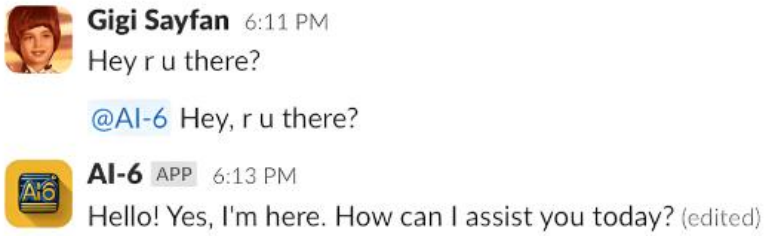
Perilaku bergabung otomatis ke saluran dengan konvensi penamaan tertentu ini, terus terang, sebagian besar berguna untuk tujuan pengembangan dan pengujian. Dalam aplikasi dunia nyata, Anda cukup mengundang bot Slack AI-6 ke saluran tertentu. Perilaku ini juga didukung, karena penangan `handle_message()` merespons pesan apa pun di saluran

tempat bot berada, seperti yang Anda ingat, selama pesan tersebut menyebutkan bot Slack AI-6. Berikut cara mengundang bot Slack AI-6 ke saluran yang bukan merupakan anggotanya:



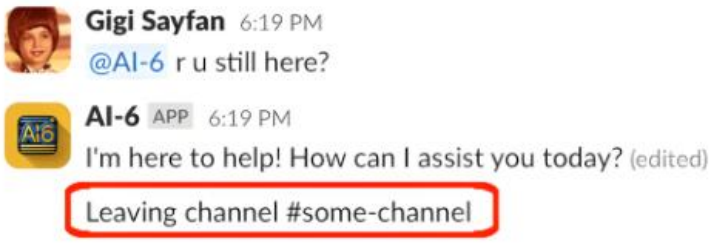
Gambar 6.10 Secara Eksplisit Mengundang Bot AI-G Ke Saluran

Saluran ini bernama **some-channel**, yang berarti bot Slack AI-6 tidak akan secara otomatis membalas pesan apa pun. Kita harus menyebutkannya menggunakan @AI-6 untuk mendapatkan respons. Mari kita coba:



Gambar 6.11 Mengontrol AI-G

Ketika aplikasi AI-6 ditutup, aplikasi tersebut akan meninggalkan semua saluran yang telah diikutinya (secara otomatis atau melalui undangan).



Gambar 6.12 Bot AI-G Meninggalkan Saluran

Ini dilakukan dalam fungsi `leave_channels()`, yang cukup sederhana:

```
def leave_channels(client):
    """Leave all the channels
    """
    channels = (c for c in client.conversations_list()['channels']
        if c['is_member'])
    for c in channels:
        client.chat_postMessage(
            channel=c['id'],
            text=f"Leaving channel #{c['name']}"
        )
        client.conversations_leave(channel=c['id'])
```

Kita telah membahas bagaimana bot Slack AI-6 menangani pesan dan mengelola saluran. Sekarang, mari kita lihat bagaimana integrasinya dengan mesin AI-6.

Integrasi Dengan Mesin AI-6

Ada dua titik kontak utama antara bot Slack AI-6 dan mesin AI-6: membuat atau mengambil instance mesin untuk setiap saluran dan menangani panggilan alat dari mesin AI-6. Fungsi `get_or_create_engine()` membantu memastikan setiap saluran memiliki instance mesin AI-6-nya sendiri, yang penting untuk dukungan multi-saluran dan multi-pengguna.

Jika kita hanya memiliki satu mesin bersama, akan sulit untuk mengelola status dan memori untuk setiap saluran. Pertama, jendela konteks akan terisi jauh lebih cepat, dan juga, LLM akan bingung dengan pesan yang tidak terkait dari saluran yang berbeda.

Berikut kodenya. Cukup sederhana. Semua mesin untuk semua saluran disimpan dalam kamus mesin. Jika ID saluran yang diminta tidak ada dalam kamus, maka akan dibuat instans mesin baru menggunakan fungsi `utils.create_channel_engine()`, dan ditambahkan ke kamus mesin. Kemudian, ia hanya mengembalikan engine tersebut:

```
def get_or_create_engine(channel_id):
    """Get an existing engine for a channel or create a new one."""
    if channel_id not in engines:
        # Create a channel-specific engine using the Slack utility function
        engines[channel_id] = utils.create_channel_engine(
            base_config_path=config_path,
            channel_id=channel_id,
            env_file_path=env_file_path
        )

    return engines[channel_id]
```

Fungsi `utils.create_channel_engine()` berisi kode serupa untuk menginstansiasi mesin AI-6 seperti yang kita lihat di bab 4, tetapi juga mengatur ID saluran dalam konfigurasi dan menginisialisasi memori untuk saluran tersebut.

Mari kita lihat kode yang menangani panggilan alat dari mesin AI-6. Fungsi `handle_tool_call()` menambahkan setiap panggilan alat dan responsnya ke utas pesan

asli di saluran Slack. Ini memungkinkan pengguna untuk melihat detail bagaimana mesin AI-6 sampai pada responsnya, termasuk panggilan alat dan hasilnya:

```
def handle_tool_call(client, channel, name, args, result):
    """Handle a tool call from the AI-6 engine."""
    # Post the tool call result as a message
    try:
        client.chat_postMessage(
            channel=channel,
            thread_ts=latest_ts,
            text=f"_Tool call: `{name}` {', '.join(args.values()) if args else
''}_\n{result}"
        )
    except SlackApiError as e:
        print(f"Error posting tool call result: {e}")
```

Sebagai rangkuman, kita telah membahas bagaimana bot Slack AI-6 dirancang dan diimplementasikan serta bagaimana integrasinya dengan mesin AI-6. Kita telah membahas beberapa fitur utama, seperti dukungan multi-channel, penanganan pesan, dan pemrosesan panggilan alat. Sekarang, mari kita alihkan perhatian kita ke UI web Chainlit, yang mirip dengan chatbot web seperti ChatGPT.

6.4 MENGEMBANGKAN UI WEB DENGAN CHAINLIT

Pada bagian sebelumnya, kita telah menjelajahi antarmuka bot Slack AI-6 dan melihat bagaimana antarmuka tersebut menyediakan lingkungan kolaboratif multi-pengguna untuk berinteraksi dengan sistem AI agen. Bot Slack unggul dalam alur kerja tim, notifikasi waktu nyata, dan integrasi dengan saluran komunikasi yang ada.

Namun, ada skenario di mana antarmuka web khusus menawarkan keuntungan: kontrol lebih besar atas pengalaman pengguna, elemen visual yang lebih kaya, dukungan yang lebih baik untuk percakapan panjang, dan penyebaran yang lebih mudah bagi pengguna yang tidak memiliki akses ke ruang kerja Slack. Di sinilah Chainlit berperan. Pada bagian ini, kita akan meneliti bagaimana AI-6 memanfaatkan Chainlit untuk menciptakan antarmuka obrolan berbasis web yang mirip dengan ChatGPT sambil mempertahankan akses penuh ke ekosistem alat dan kemampuan agen AI-6.

Gambaran Umum Kerangka Kerja Chainlit

Chainlit adalah kerangka kerja Python yang dirancang khusus untuk membangun antarmuka obrolan untuk aplikasi LLM. Kerangka kerja ini menyediakan dukungan siap pakai untuk banyak elemen UI, siklus hidup obrolan, respons streaming, unggahan file, manajemen pengaturan, persistensi sesi, dan banyak lagi—semua fitur yang dibutuhkan untuk antarmuka AI percakapan yang canggih. Kerangka kerja ini juga sangat mudah disesuaikan, memungkinkan Anda untuk menambahkan elemen UI kustom Anda sendiri.

Pengalaman pengembangan Chainlit dirancang untuk iterasi cepat. Kerangka kerja ini menjalankan server pengembangan lokal dengan hot-reloading, memungkinkan pembuatan prototipe dan debugging yang cepat. Kerangka kerja ini terintegrasi dengan mulus dengan

backend LLM populer seperti OpenAI, LangChain, dan Hugging Face, dan mendukung alur kerja sinkron dan asinkron untuk fleksibilitas maksimal.

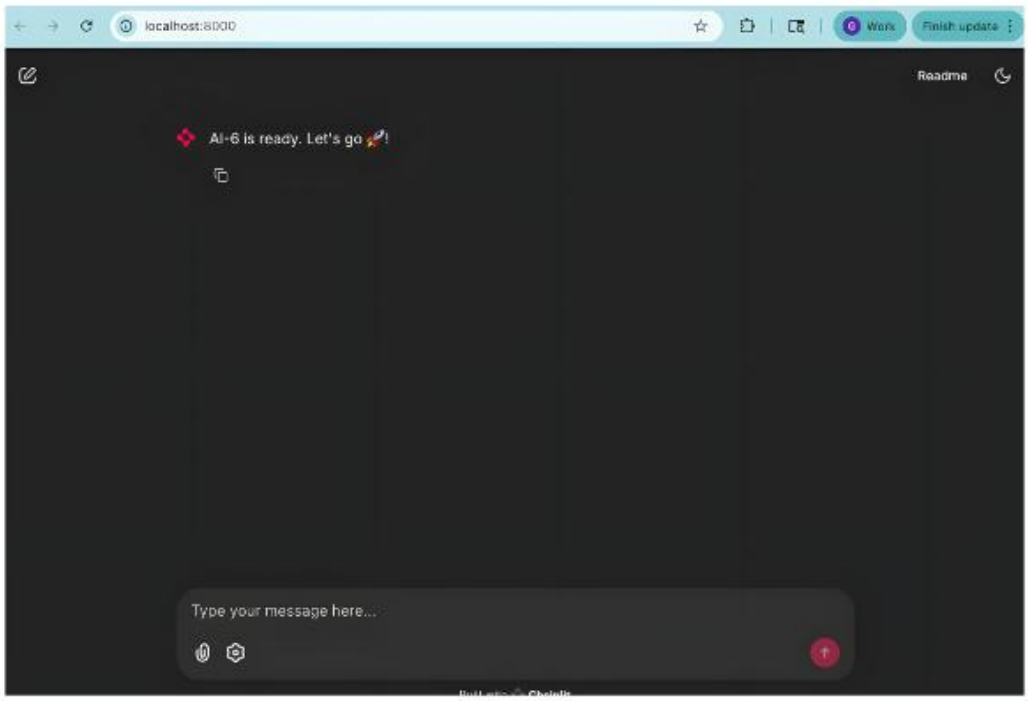
Mari kita jelajahi bagaimana AI-6 memanfaatkan Chainlit untuk membangun antarmuka web sederhana yang melengkapi bot Slack-nya. Aplikasi Chainlit AI-6 sengaja dibuat minimal dan tidak dirancang untuk sepenuhnya menampilkan kemampuan Chainlit. Tujuan utamanya adalah untuk menunjukkan bagaimana backend AI-6 dapat diintegrasikan ke dalam UI berbasis web, yang berfungsi sebagai fondasi untuk aplikasi web yang lebih canggih dan lebih baik.

Mencoba AI-6 Chainlit

Mari kita mulai dengan kasus penggunaan dunia nyata yang sangat berguna, yang melibatkan penambahan alur kerja GitHub Actions ke repositori AI-6 untuk menjalankan semua pengujian kita setiap kali kita mengirimkan perubahan baru. Pertama, kita perlu meluncurkan aplikasi Chainlit. Kita dapat melakukannya menggunakan skrip ai6.sh yang mudah digunakan dengan meneruskan argumen chainlit:

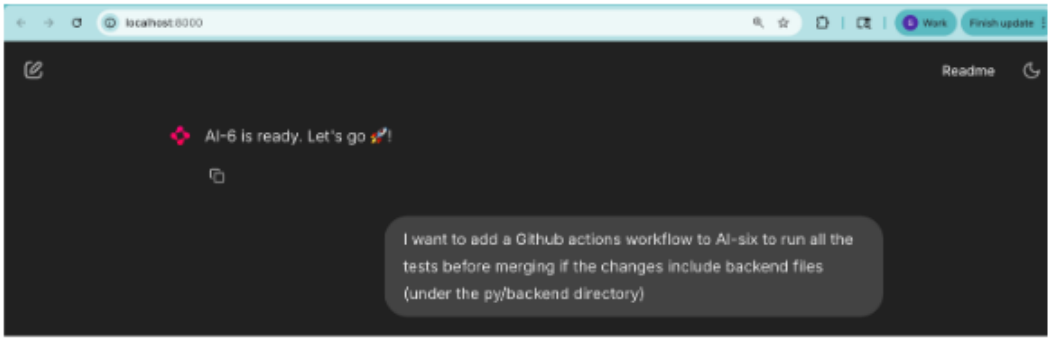
```
> ./ai6.sh chainlit
virtualenv: /Users/gigi/gigi/git/ai-six/py/venv
2025-07-19 11:28:15 - HTTP Request: GET https://api.openai.com/v1/models "HTTP/1.1
200 OK"
2025-07-19 11:28:17 - HTTP Request: GET https://api.openai.com/v1/models "HTTP/1.1
200 OK"
2025-07-19 11:28:18 - Your app is available at http://localhost:8000
```

Aplikasi AI-6 Chainlit sekarang sudah berjalan dan kita dapat mengaksesnya di http://localhost:8000. Mari kita buka di peramban web:



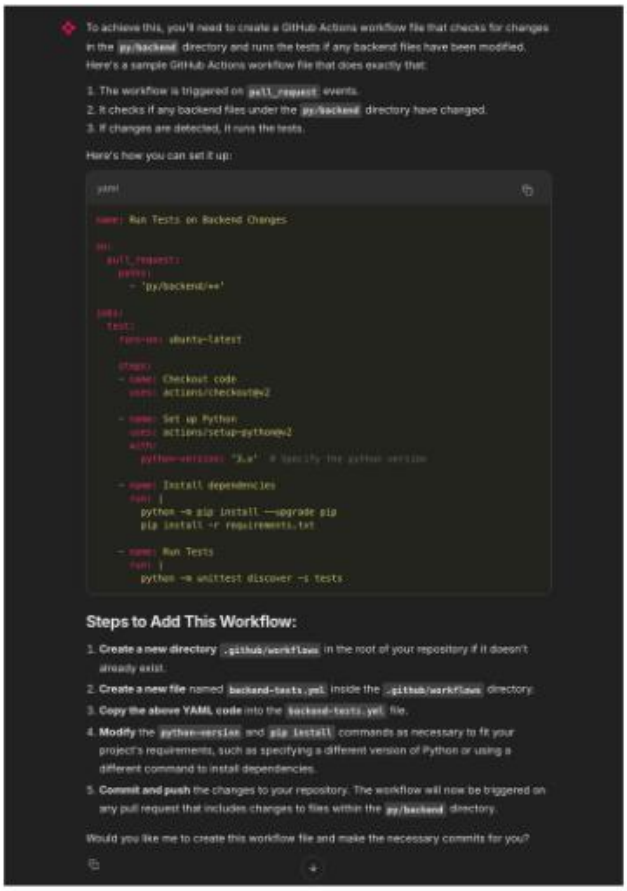
Gambar 6.13 Aplikasi Web Chainlit

Sekarang, mari kita minta AI-6 untuk menambahkan alur kerja GitHub Actions ke repositori AI-6:



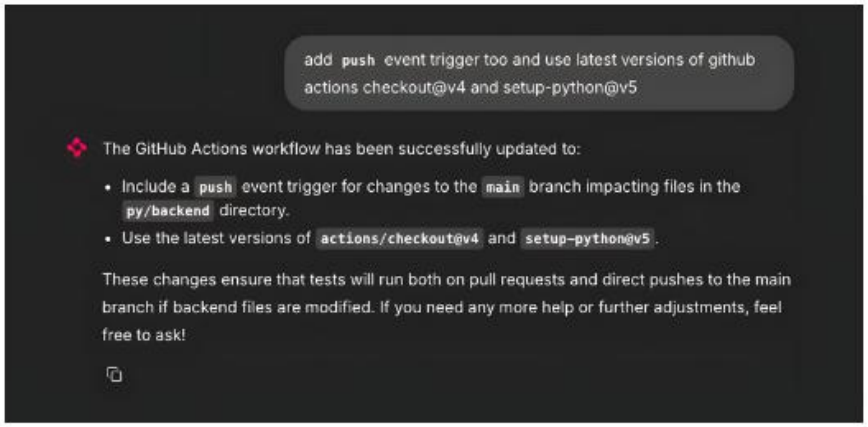
Gambar 6.14 Perintah Awal Pengguna

AI-6 memberikan penjelasan yang sangat jelas dan detail, termasuk file alur kerja sebenarnya dan semua langkah untuk menyelesaikan tugas. Perhatikan betapa menyenangkannya UI-nya. Kode diberi kode warna dengan benar dan Markdown diformat dengan rapi:



Gambar 6.15 Respons LLM Yang Elaboratif

Namun solusi ini memiliki beberapa masalah. Pertama, solusi ini menggunakan event `pull_request`, yang tidak terpicu pada push reguler ke cabang utama. Kita ingin alur kerja berjalan juga pada event push. Kedua, solusi ini menggunakan versi usang dari beberapa GitHub Actions, seperti `actions/checkout@v2` dan `actions/setup-python@v2`. Kita ingin menggunakan versi terbaru. Mari kita minta AI-6 untuk memperbaiki masalah ini:



Gambar 6.16 Interaksi Lanjutan

Mari kita lihat apa yang sebenarnya dilakukan AI-6. Berikut adalah commit asli dan commit perbaikannya:

```
> git log --oneline -n 2
f5198ca (HEAD -> main, origin/main, origin/HEAD)
Update_workflow_for_push_event_and_latest_actions_versions
65a0c12 Add_GitHub_Actions_workflow_to_run_tests_on_backend_changes
```

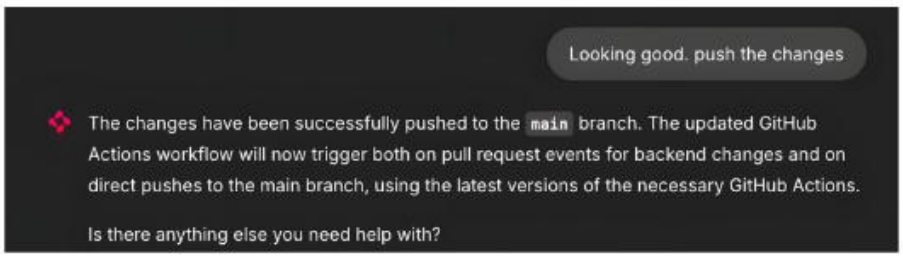
Mari kita tinjau perubahannya:

```
> git diff HEAD~2
diff  --git  a/.github/workflows/backend-tests.yml  b/.github/workflows/backend-
tests.yml
new file mode 100644
index 0000000..1339264
--- /dev/null
+++ b/.github/workflows/backend-tests.yml
@@ -0,0 +1,35 @@
+name: Run Tests on Backend Changes
+
+
+on:
+  pull_request:
+    paths:
+      - 'py/backend/**'
+  push:
+    branches:
+      - main
```

```
+    paths:
+      - 'py/backend/**'
+
+
+
+jobs:
+  test:
+    runs-on: ubuntu-latest
+
+
+
+    steps:
+    - name: Checkout code
+      uses: actions/checkout@v4
+
+
+
+
+    - name: Set up Python
+      uses: actions/setup-python@v5
+      with:
+        python-version: '3.x'  # Specify the python version
+
+
+
+    - name: Install dependencies
+      working-directory: py
+      run: |
+        python -m pip install --upgrade pip
+        pip install -r requirements.txt
+
+
+
+    - name: Run Tests
+      working-directory: py
+      run: |
+        python -m unittest discover -s backend/tests
```

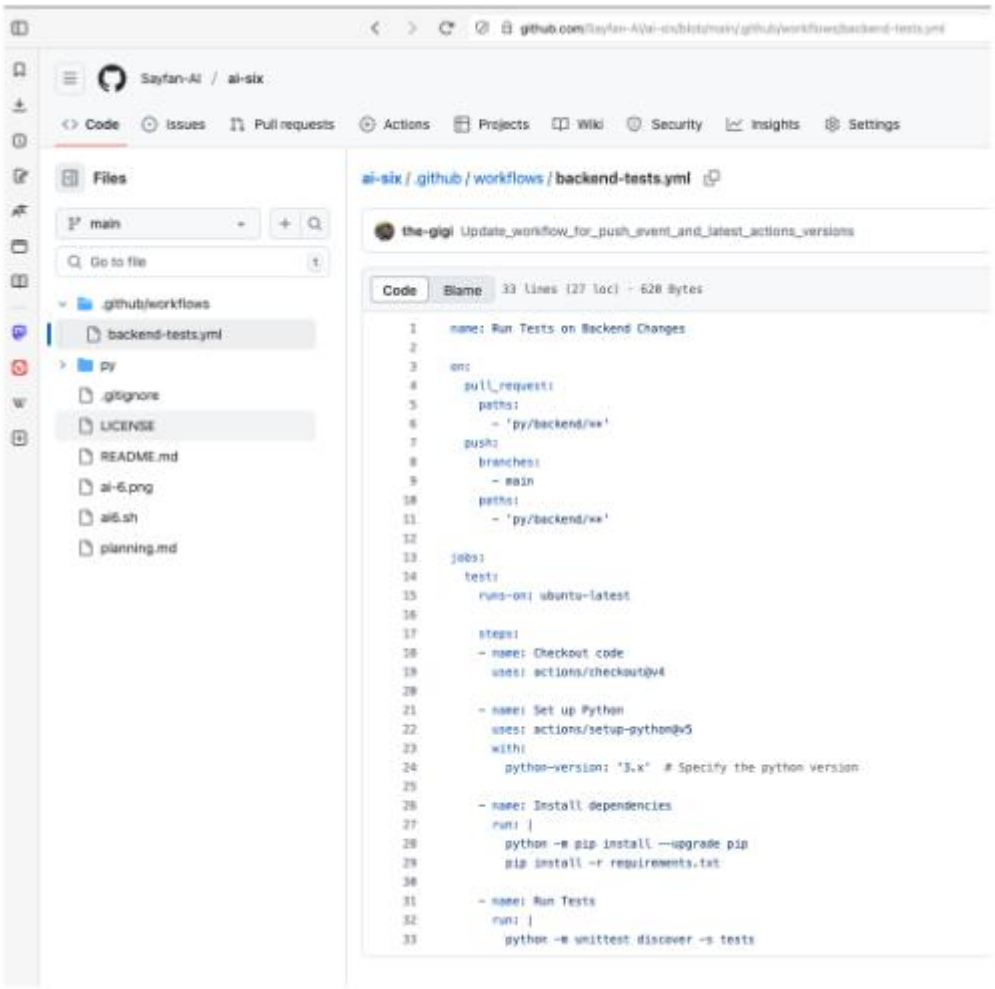
Semuanya terlihat benar sekarang. AI-6 telah memperbaiki semua masalah yang telah kami identifikasi. Ini menyoroti pentingnya konsep keterlibatan manusia.

LLM memang bagus, tetapi belum sempurna (untuk saat ini). Sekarang kita dapat memerintahkan AI-6 untuk mengirimkan perubahan ke GitHub:



Gambar 6.17 AI-G Mengirimkan Perubahan Ke GitHub

Mari kita periksa GitHub dan konfirmasikan bahwa alur kerja GitHub Actions yang baru memang ada di sana:



Gambar 6.18 Alur Kerja GitHub Actions

Sesi tadi sangat bagus dengan AI-6 menggunakan antarmuka Chainlit. Kami berhasil menambahkan alur kerja GitHub Actions baru ke repositori AI-6, memperbaiki beberapa masalah, dan mendorong perubahan ke GitHub. Antarmuka Chainlit memberikan tampilan percakapan yang bersih dan modern.

Mari kita lihat bagaimana antarmuka Chainlit diimplementasikan dan bagaimana integrasinya dengan mesin AI-6.

Konfigurasi Dan Integrasi Mesin

Sama seperti antarmuka bot Slack, aplikasi web Chainlit memerlukan konfigurasi untuk menginisialisasi dan terhubung dengan mesin AI-6. Titik masuk program utama ada di file `app.py`. Mari kita uraikan.

Mari kita mulai dengan impor. Aplikasi Chainlit mengimpor kelas SimpleNamespace dari modul typing pustaka standar. Ini adalah kelas keren yang dapat Anda gunakan seperti kamus, tetapi dengan akses notasi titik ke atribut. Ia juga mengimpor chainlit itu sendiri, yang

di-alias ke cl, dan fungsi `run_chainlit` dari modul `chainlit.cli`, yang digunakan untuk menjalankan aplikasi Chainlit.

Selain itu, ia juga mengimpor paket pathology.path untuk menemukan file konfigurasi dan modul engine_utils dari paket frontend.common bersama, yang akan digunakan untuk membuat instance mesin AI-6:

```
from types import SimpleNamespace

import chainlit as cl
from chainlit.cli import run_chainlit
import pathology.path

from frontend.common import engine_utils
```

Selanjutnya, ia menemukan file konfigurasi dan membuat instance mesin AI-6 darinya. Inilah bagian inisialisasi mesin AI-6:

```
script_dir = pathology.path.Path.script_dir()

# Load the engine from YAML configuration
config_path = str((script_dir / "config.yaml").resolve())

# Create engine from configuration file
# Environment variables will be automatically interpolated by Config.from_file
engine, engine_config = engine_utils.create_from_config(config_path)
```

Di sini, kita membuat objek `app_config`, yang merupakan instance SimpleNamespace yang menyimpan konfigurasi untuk aplikasi Chainlit:

```
app_config = SimpleNamespace(
    selected_model=engine.default_model_id,
    available_models=list(engine.model_provider_map.keys()),
    enabled_tools={tool: True for tool in engine.tool_dict},
)

TOOL_PREFIX = "tool:"
```

Objek konfigurasi ini akan digunakan nanti saat memproses pesan dari pengguna. Integrasi dengan mesin AI-6 dilakukan melalui variabel mesin yang telah dibuat sebelumnya.

Siklus Hidup Aplikasi Dan Penanganan Peristiwa

Pada titik ini, kita telah menyiapkan mesin dan konfigurasi (objek app_config), dan logika utama aplikasi berjalan dengan memanggil fungsi `run_chainlit()` (diimpor dari `chainlit.cli`) dengan file `app.py`. Ini adalah cara yang rumit untuk meluncurkan aplikasi Chainlit dari program Python. Ini diperlukan untuk debugging:

```
if __name__ == "__main__":
    target = str(script_dir / "app.py")
    run_chainlit(target)
```

Cara standar untuk menjalankan aplikasi Chainlit di terminal adalah dengan menggunakan CLI Chainlit:

```
> chainlit run app.py
```

Pokoknya, setelah aplikasi Chainlit berjalan, aplikasi akan mulai mendengarkan peristiwa dan memprosesnya. Aplikasi Chainlit kita menangani peristiwa-peristiwa berikut:

- `-on_chat_start`: Dipanggil saat sesi obrolan baru dimulai
- `-on_message`: Dipanggil saat pesan pengguna baru diterima
- `-on_settings_update`: Dipanggil saat pengaturan diubah di panel pengaturan Mari kita lihat masing-masing penangan peristiwa ini dan bagaimana cara kerjanya.

Penanganan Peristiwa ON_CHAT_START

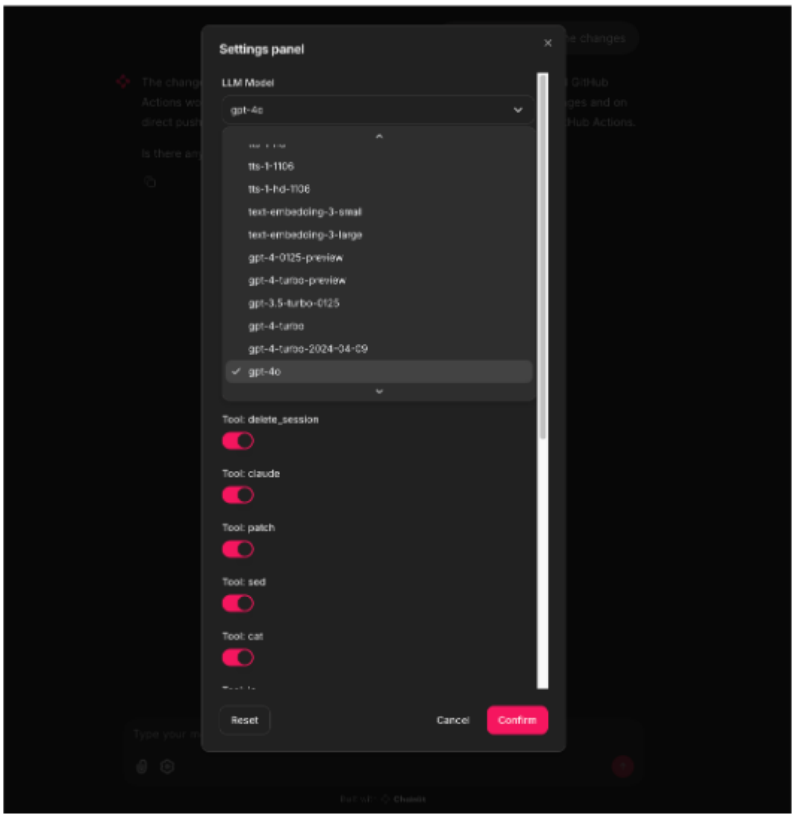
Penanganan peristiwa ini dipanggil tepat sekali saat aplikasi dimulai. Ini cukup sederhana. Ia memanggil `setup_settings()` dan menampilkan pesan selamat datang yang kita lihat saat kita mencoba aplikasi Chainlit:

```
@cl.on_chat_start
async def on_chat_start():
    await setup_settings()
    await cl.Message(content="AI-6 is ready. Let's go 🚀!").send()
```

Fungsi `setup_settings()` sangat menarik. Fungsi ini mengatur pengaturan awal untuk aplikasi Chainlit di widget UI, termasuk pemilihan LLM dan pergantian alat. Fungsi ini menggunakan `cl.input_widget.Select` untuk pemilihan model dan `cl.input_widget.Switch` untuk setiap alat. Nilai awal diambil dari objek `app_config` yang telah kita buat sebelumnya:

```
async def setup_settings():
    model_select = cl.input_widget.Select(
        id="model",
        label="LLM Model",
        values=app_config.available_models,
        initial_value=app_config.selected_model,
        initial_index=app_config.available_models.index(
            app_config.selected_model),
    )
    tool_switches = [
        cl.input_widget.Switch(
            id=f"{TOOL_PREFIX}{tool_name}",
            label=f"Tool: {tool_name}",
            initial=tool_value,
        )
        for tool_name, tool_value in app_config.enabled_tools.items()
    ]

    await cl.ChatSettings([model_select] + tool_switches).send()
```

Namun, di mana pengaturan UI tersebut ditampilkan? Nah, ada tombol roda gigi kecil di pojok kiri bawah kotak input pengguna yang akan menampilkan **panel Pengaturan**. Tampilannya seperti ini:



Gambar 6.19 Panel Pengaturan Aplikasi Chainlit

Kita akan melihat bagaimana pengaturan ini digunakan dan dapat dimodifikasi nanti. Untuk saat ini, mari kita perhatikan bahwa nilai awal LLM diatur ke nilai dalam objek `app_config`, serta alat yang diaktifkan. Seluruhnya dikirim ke Chainlit menggunakan metode `cl.ChatSettings().send()`.

Pada titik ini, kita telah menginisialisasi aplikasi Chainlit, mengkonfigurasinya dengan LLM yang tepat dan serangkaian alat yang diaktifkan, dan siap untuk menangani pesan.

Penanganan Peristiwa ON_MESSAGE

Ini adalah penanganan peristiwa utama yang memproses pesan masuk dari pengguna dan berinteraksi dengan mesin AI-6. Penanganan ini dipanggil setiap kali pengguna mengetik pesan baru.

Awalnya, penanganan ini membuat objek cl.Message baru dengan konten kosong dan langsung mengirimkannya ke UI Chainlit. Pesan ini akan diperbarui nanti dengan respons dari mesin AI-6:

```
@cl.on_message
async def on_message(message: cl.Message):
    """Process user messages and generate responses."""
    # Create a new message for the response
```

```
    msg = cl.Message(content="")
    await msg.send()
```

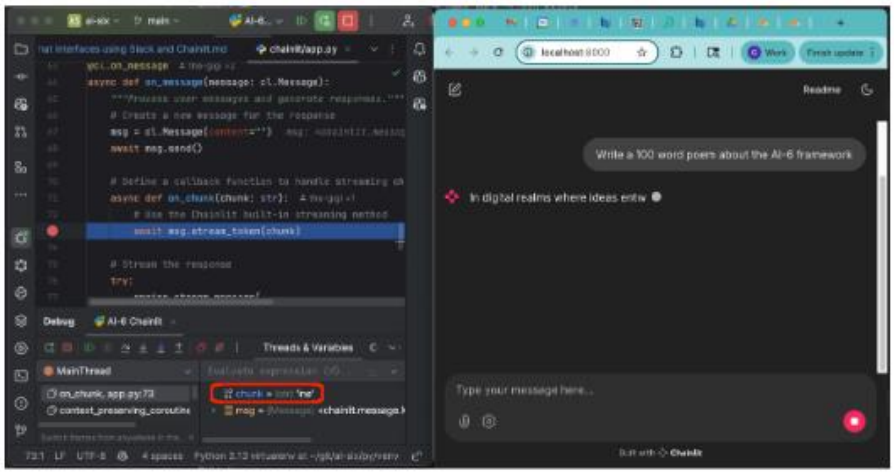
Selanjutnya, kode tersebut mendefinisikan fungsi callback, on_chunk(), yang akan digunakan untuk menangani pengiriman bertahap (streaming chunks) respons dari mesin AI-6. Fungsi ini akan dipanggil setiap kali mesin AI-6 mengirimkan potongan respons baru:

```
# Define a callback function to handle streaming chunks
async def on_chunk(chunk: str):
    # Use the ChainLit built-in streaming method
    await msg.stream_token(chunk)
```

Berikut adalah panggilan ke mesin AI-6 untuk memproses pesan. Ia menggunakan metode `engine.stream_message()` untuk mengalirkan respons. Ia meneruskan konten pesan pengguna, LLM yang dipilih dari `app_config`, fungsi `callback` on_chunk, dan daftar ID alat yang diaktifkan. Saat mesin AI-6 memproses permintaan, ia akan mengirimkan beberapa chunk ke fungsi on_chunk, yang akan memperbarui pesan respons yang awalnya kosong. Jika ada masalah di titik mana pun, ia akan menampilkan pesan kesalahan di UI Chainlit:

```
# Stream the response
try:
    engine.stream_message(
        message.content,
        app_config.selected_model,
        on_chunk_func=lambda chunk: cl.run_sync(on_chunk(chunk)),
        available_tool_ids=[
            k for k, v in app_config.enabled_tools.items() if v
        ],
    )
    # Mark the message as complete

    await msg.update()
except Exception as e:
    await cl.Message(content=f"Error: {str(e)}").send()
```

Mari kita lihat prosesnya secara langsung. AI-6 diminta untuk menulis puisi 100 kata tentang dirinya sendiri dan menempatkan breakpoint di handler `on_chunk()`. Ini sangat menarik. AI-6 memulai puisi dengan "Di alam digital tempat ide-ide terjalin". LLM memilih untuk mengalirkan satu token keluaran pada satu waktu. Dalam kasus kata "entwine", ia memecahnya menjadi dua bagian (dua token): entw dan ine. Lihat gambar berikut:



Gambar 6.20 AI-G Sedang Menulis Puisi

Satu hal terakhir: perhatikan bahwa dalam versi kode saat ini, kami tidak meneruskan argumen `on_tool_call_func` ke metode `engine.stream_message()`, sehingga mesin AI-6 tidak akan memberi tahu UI tentang panggilan alat apa pun. Ini disengaja karena saya tidak ingin mencemari tampilan dengan panggilan alat yang bertele-tele. Bot Slack AI-6 memiliki tempat alami untuk panggilan alat dalam thread.

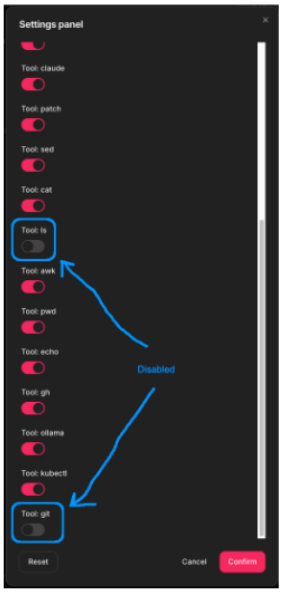
Namun, ini tidak berlaku untuk UI Chainlit. Panel samping khusus mungkin akan ditambahkan di beberapa titik. Kemudian, pengguna dapat memilih untuk membuka dan menangkap panggilan alat di sana.

Penanganan Peristiwa on_settings

Antarmuka AI-6 Slack dan CLI memiliki LLM dan alat yang telah dikonfigurasi sebelumnya. Aplikasi Chainlit memungkinkan pengguna untuk memilih dan memodifikasi LLM serta mengaktifkan atau menonaktifkan alat secara langsung. Ketika pengguna mengubah pengaturan di panel pengaturan dan mengklik tombol **Konfirmasi**, penanganan peristiwa `on_settings()` dipanggil. Fungsinya hanya memperbarui `app_config` dengan pengaturan baru. Itu termasuk LLM yang dipilih dan alat yang diaktifkan:

```
@cl.on_settings_update
async def on_settings_update(new_settings):
    app_config.selected_model = new_settings["model"]
    for k, v in new_settings.items():
        if k.startswith(TOOL_PREFIX):
            tool_name = k.replace(TOOL_PREFIX, "")
            app_config.enabled_tools[tool_name] = v
```

Mari kita lihat cara kerjanya. Saya akan menonaktifkan alat ls dan git. Ini seharusnya mencegah LLM menggunakan alat-alat ini untuk menampilkan daftar file. Seharusnya LLM masih dapat menemukan direktori saat ini karena alat pwd tetap diaktifkan.



Gambar 6.21 Menonaktifkan alat

Ya, berhasil! AI-6 tidak menyediakan alat ls dan git, jadi LLM tidak memiliki cara untuk menampilkan daftar file. Singkatnya, kita telah membahas antarmuka web Chainlit untuk AI-6 di bagian ini. Kita telah melihat bagaimana antarmuka tersebut diinisialisasi dan diintegrasikan dengan mesin AI-6, bagaimana antarmuka tersebut menangani pesan, dan konfigurasi dinamisnya dengan kemampuan untuk beralih LLM dan mengaktifkan/menonaktifkan alat. Kita baru sedikit membahas apa yang mungkin dilakukan, tetapi kita telah melihat bagaimana antarmuka ini menyediakan alternatif berbasis web yang ringan untuk bot Slack AI-6.

Di bagian selanjutnya, kita akan membahas praktik terbaik untuk membangun antarmuka agen interaktif yang lebih canggih dan kompleks.

6.5 PRAKTIK TERBAIK UNTUK ANTARMUKA AGEN INTERAKTIF

Antarmuka frontend yang telah kita bangun di bagian sebelumnya merupakan titik awal yang bagus untuk berinteraksi dengan agen AI, tetapi ketika membangun sistem AI tingkat produksi, ada beberapa praktik terbaik yang perlu dipertimbangkan. Praktik-praktik ini memastikan bahwa pengalaman pengguna lancar, tujuan agen jelas, dan sistem kuat serta mudah dipelihara.

Klarifikasi Peran Dan Kemampuan Agen

Antarmuka agen interaktif yang sukses harus segera memperjelas kepada pengguna apa itu agen, apa yang dapat dilakukannya, dan jenis input apa yang diharapkannya. Ambiguitas dalam ruang lingkup agen menyebabkan frustrasi pengguna dan interaksi yang gagal. Antarmuka harus mengekspos kemampuan utama melalui elemen UI seperti tombol, perintah pelengkapan otomatis, atau overlay bantuan. Misalnya, agen analisis file mungkin menampilkan zona seret untuk file, daftar jenis file yang didukung, dan contoh perintah seperti "Ringkas CSV ini."

Agen juga harus mengkomunikasikan keterbatasannya secara proaktif. Ini termasuk batasan model (misalnya, batasan token), alat yang dinonaktifkan, atau format input yang

tidak didukung. Dalam sistem multi-agen di mana agen tingkat tinggi dapat secara otomatis memunculkan sub-agen, seringkali lebih penting untuk fokus pada agen tingkat tinggi yang dapat dikelola pengguna baik melalui opsi konfigurasi yang telah ditentukan sebelumnya atau perintah pengguna.

Pengamanan – Konfirmasi, Uji Coba, Dan Jaring Pengaman

Agen AI dapat memanggil alat yang berpotensi menyebabkan banyak kerusakan jika disalahgunakan. UI harus secara default mengambil sikap konservatif, meminta pengguna untuk mengkonfirmasi tindakan berdampak tinggi seperti penimpaan file, penyebaran kode, atau penghapusan. Perintah konfirmasi tidak hanya mencegah kerusakan yang tidak disengaja tetapi juga memaksa pengguna untuk merenungkan implikasi dari masukan mereka.

Selain konfirmasi, UI harus mendukung mode uji coba. Uji coba adalah simulasi aman yang menunjukkan apa yang akan dilakukan agen tanpa benar-benar mengeksekusi apa pun. Ini sangat berharga saat mengembangkan dan menguji alur kerja, integrasi API, atau perintah infrastruktur. Ketika pengguna dapat melihat pratinjau hasil dan memverifikasi kebenarannya, kepercayaan pada sistem meningkat. Dikombinasikan dengan kemampuan untuk membatalkan atau membatalkan tindakan, jaring pengaman ini membuat sistem agen terasa kuat tetapi tidak gegabah. Selain itu, sangat disarankan untuk menambahkan tombol "Batalkan" pada antarmuka pengguna (UI) yang memungkinkan pengguna untuk menghentikan agen kapan saja.

Desain Untuk Transparansi Berbasis Giliran

Agen AI beroperasi secara percakapan, berbasis giliran. Model interaksi ini harus dianut dan digunakan untuk menampilkan penalaran internal, rantai pemikiran (untuk model yang mendukungnya), dan panggilan alat. Setiap langkah yang diambil agen harus direpresentasikan dalam antarmuka: perintah, keputusan, panggilan alat, keluaran, dan hasil sementara. Ini memudahkan pengguna untuk memahami "mengapa" di balik perilaku agen.

Antarmuka yang baik menunjukkan siklus hidup lengkap dari satu interaksi dalam alur linier yang bersih. Misalnya, ketika agen memutuskan untuk memanggil alat, parameter permintaan dan respons harus terlihat dalam alur obrolan atau panel yang dapat diperluas. Kesalahan, percobaan ulang, atau fallback juga harus ditampilkan sebagai bagian dari cerita. Pola desain ini membantu pengguna melakukan debugging, audit, dan mempelajari cara memberi perintah yang lebih baik kepada agen dan meningkatkan hasil.

Prioritaskan Pemanggilan Alat Dan Loop Umpan Balik

Salah satu nilai tambah terbesar dari sistem berbasis agen adalah kemampuannya untuk menggunakan alat eksternal. Antarmuka pengguna (UI) harus memperjelas kapan alat dipanggil, bagaimana alat tersebut dipilih, dan apa hasilnya. Ini menciptakan lingkaran umpan balik di mana pengguna dapat mengevaluasi apakah penggunaan alat tersebut tepat dan melakukan intervensi jika tidak.

Antarmuka juga harus menampilkan metadata pemanggilan alat: stempel waktu, argumen, nilai pengembalian, dan kesalahan apa pun. Ketika alat gagal (misalnya, kesalahan jaringan atau input yang salah format), agen harus melaporkan masalah tersebut dengan baik dan menawarkan opsi seperti coba lagi, revisi input, atau lewati. Memungkinkan pengguna

untuk memberi arahan kepada agen atau menimpa pilihan alat membuat percakapan tetap produktif dan selaras dengan maksud pengguna.

Penting juga untuk menyediakan cara bagi pengguna untuk melihat alat yang tersedia dan deskripsinya, serta menyesuaikan kumpulan alat per agen.

Rangkul Kemampuan Konfigurasi Dan Kontrol Runtime

Pengguna yang berbeda memiliki preferensi dan ambang kepercayaan yang berbeda. Beberapa mungkin ingin agen mengeksekusi alat secara otomatis, sementara yang lain mungkin lebih menyukai konfirmasi manual. UI harus mengekspos pengaturan runtime untuk mengontrol perilaku ini. Ini termasuk mengaktifkan/menonaktifkan alat, mengganti model, memilih tingkat detail output, dan menentukan strategi cadangan.

Kontrol ini harus mudah diakses (misalnya, melalui panel pengaturan atau perintah obrolan) dan tetap ada di seluruh sesi jika sesuai. Memungkinkan pengguna untuk menyesuaikan gaya interaksi mereka meningkatkan kegunaan dan kepuasan.

Log, Metrik, Dan Pelacakan Terdistribusi Untuk Sistem Berbasis Agen

Setiap tindakan pengguna, keputusan agen, dan panggilan alat harus dicatat dengan metadata. Log ini berfungsi untuk tujuan yang berorientasi pada pengguna dan pengembang. Bagi pengguna, log dapat divisualisasikan sebagai garis waktu, memungkinkan penelusuran balik dan pemutaran ulang sesi sebelumnya. Bagi pengembang, log mendukung debugging, penyetelan kinerja, dan penyempurnaan prompt.

Metrik harus mencakup frekuensi panggilan alat, latensi respons rata-rata, durasi sesi, tingkat keberhasilan tugas, dan intervensi pengguna. Seiring waktu, metrik ini dapat memandu peningkatan perilaku agen dan UI. UI tanpa instrumentasi adalah kotak hitam. Dengan menginstrumentasi setiap bagian alur kerja, sistem berbasis agen dapat berevolusi dari prototipe menjadi platform yang andal dan dapat diaudit. Pelacakan terdistribusi khusus LLM juga sangat penting untuk sistem yang kompleks. Setiap giliran harus diwakili oleh rentang. Jejak harus menangkap sesi atau percakapan lengkap.

Jika Anda mengikuti praktik terbaik ini, Anda akan dapat membangun antarmuka agen interaktif yang ramah pengguna dan ramah pengembang. Mereka akan memberikan pengalaman pengguna yang hebat dan memungkinkan pengguna untuk berinteraksi dengan agen AI secara alami dan memuaskan. Bot Slack AI-6 dan antarmuka web Chainlit yang kami bangun dalam bab ini menunjukkan beberapa elemen dari praktik terbaik ini, tetapi masih banyak ruang untuk perbaikan.

6.6 RINGKASAN

Dalam bab ini, kami beralih dari backend dan arsitektur internal AI-6 ke UI frontend-nya, dengan fokus pada bagaimana UI memungkinkan kolaborasi yang efektif antara pengguna dan agen otonom. Kami meneliti mengapa visibilitas, interupsi, debugging, dan kepercayaan merupakan perhatian UI yang penting untuk sistem agen, dan bagaimana peran yang berbeda membutuhkan tingkat detail yang berbeda dalam antarmuka.

Kemudian kami mengeksplorasi implementasi bot Slack AI-6, menyoroti pilihan desainnya seperti threading panggilan alat, instance mesin per saluran, dan penanganan pesan

waktu nyata. Familiaritas Slack, dukungan multi-pengguna, dan kemampuan notifikasi menjadikannya antarmuka yang ideal untuk alur kerja kolaboratif.

Terakhir, kami meneliti antarmuka web berbasis Chainlit, menunjukkan bagaimana antarmuka tersebut memberikan pengalaman pengguna tunggal yang bersih dan dapat disesuaikan. Dengan dukungan untuk pengaktifan/penonaktifan alat, pemilihan model, dan output streaming yang kaya, Chainlit melengkapi kekuatan Slack dan memungkinkan kontrol pengguna yang lebih detail. Kedua antarmuka ini menjadi landasan bagi praktik terbaik dalam membangun UI agen yang tangguh dan transparan.

Pada bab selanjutnya, kita akan membahas Model Context Protocol (MCP), sebuah mekanisme standar untuk berbagi status model dan konteks eksekusi di berbagai alat dan agen. Kita akan mendefinisikan apa itu MCP, mengeksplorasi cara mengimplementasikan server dan klien yang sesuai dengan MCP, dan membahas cara mengintegrasikan MCP ke dalam kerangka kerja agen Anda sendiri untuk memungkinkan berbagi konteks yang lancar di berbagai komponen AI.

BAB 7
MENGINTEGRASIKAN DENGAN EKOSISTEM PROTOKOL KONTEKS MODEL

7.1 PENDAHULUAN

Pada bab-bab sebelumnya, kita telah menjelajahi cara membangun sistem AI berbasis agen, cara mengintegrasikannya dengan alat khusus, dan cara berinteraksi dengannya menggunakan berbagai antarmuka pengguna seperti **antarmuka baris perintah** (CLI), bot Slack, dan UI web berbasis Chainlit. Pada bab ini, kita akan fokus pada pengintegrasian kerangka kerja AI berbasis agen Anda dengan ekosistem **Protokol Konteks Model** (MCP). MCP adalah hal yang penting! Ini adalah protokol yang dirancang untuk menstandarisasi interaksi antara model sistem AI berbasis agen dan alat. Kita akan mulai dengan menjelajahi MCP dan manfaatnya. Kemudian, kita akan melihat cara membangun server dan klien MCP. Terakhir, kita akan mendemonstrasikan cara mengintegrasikan MCP ke dalam AI-6, kerangka kerja AI berbasis agen kita. Kita akan membahas topik-topik utama berikut:

- Apa itu MCP?
- Arsitektur inti MCP dan komponen intinya
- Membangun server MCP
- Membangun klien MCP
- Mengintegrasikan MCP ke dalam kerangka kerja AI-6
- Menggabungkan semuanya

Pada akhir bab ini, Anda akan memiliki pemahaman yang solid tentang MCP, mengapa MCP begitu hebat, dan bagaimana menggunakannya untuk membangun sistem AI agen yang dapat memanfaatkan ekosistem server dan klien MCP yang kaya.

7.2 APA ITU MCP?

MCP diperkenalkan oleh tim di Anthropic untuk menstandarisasi interaksi antara sistem AI berbasis agen, sumber data, dan layanan eksternal. Dengan MCP, Anda dapat membangun sistem AI berbasis agen yang "berbicara" MCP, dan Anda dapat secara otomatis berintegrasi dengan server MCP apa pun. Ini berarti Anda memperluas kemampuan sistem AI dengan alat dari ekosistem besar di luar alat kustom. Bagi pengembang alat MCP, ini mewujudkan prinsip “Tulis sekali, jalankan di mana saja.” Protokol ini memiliki spesifikasi formal. Protokol ini memiliki versi, dan berkembang dengan cepat.

Motivasi Di Balik MCP

Sebelum MCP, setiap sistem AI berbasis agen pada dasarnya berdiri sendiri. Jika seorang pengembang ingin agen AI mereka berinteraksi dengan alat eksternal seperti **antarmuka pemrograman aplikasi** (API), basis data, atau layanan milik perusahaan, mereka harus merancang dan mengimplementasikan konektor khusus. Konektor ini tidak hanya harus tahu cara berkomunikasi dengan sistem eksternal, tetapi juga cara menyuntikkan informasi

yang relevan ke dalam konteks LLM dengan cara yang berguna dan tepat waktu. Hal ini sering diimplementasikan sebagai alat yang terkait erat dengan kerangka kerja AI spesifik yang digunakan. Pendekatan ini dengan cepat menjadi mimpi buruk pemeliharaan:

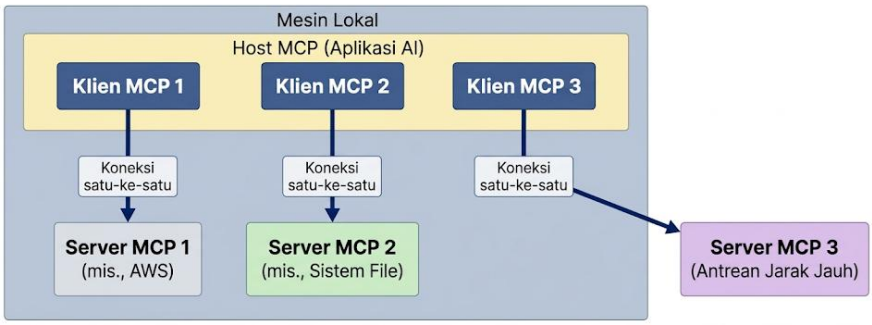
- Pengembang harus mengimplementasikan ulang logika injeksi konteks untuk setiap kasus penggunaan
- Penyedia alat dan layanan harus menulis beberapa lapisan integrasi untuk mendukung sistem AI yang berbeda
- Tidak ada cara standar bagi alat untuk menyatakan apa yang dapat mereka lakukan selain spesifikasi panggilan alat yang tidak kompatibel dari penyedia LLM yang berbeda

Akibatnya, ekosistem menjadi terfragmentasi dan rapuh. Alat terkait erat dengan kerangka kerja tertentu, dan penggunaan kembali di berbagai proyek hampir tidak mungkin.

MCP diciptakan untuk menyelesaikan masalah ini. Ini memperkenalkan protokol bersama yang menstandarisasi bagaimana alat dan sumber daya dapat diekspos ke model. Alih-alih mengkodekan logika khusus per integrasi, pengembang sekarang dapat mengimplementasikan server yang sesuai dengan MCP yang secara terprediksi mengiklankan kemampuannya. Demikian pula, sistem AI hanya perlu mengimplementasikan satu antarmuka klien untuk secara dinamis menemukan, menanyakan, dan memanggil alat dan sumber daya dari server MCP mana pun. Tersedia pustaka yang siap pakai untuk banyak bahasa pemrograman populer, sehingga memudahkan implementasi server dan klien MCP. Hasilnya adalah ekosistem modular dan dapat disusun, di mana alat dapat digunakan kembali di berbagai proyek. Mari kita lihat lebih dekat arsitektur dan komponen inti MCP.

7.3 ARSITEKTUR INTI MCP DAN KOMPONEN INTINYA

MCP memiliki arsitektur klien-server. Server MCP mengekspos alat dan sumber daya, sementara klien MCP mengonsumsinya. Klien MCP tertanam dalam host MCP (sistem AI agen) dan dapat menggunakan alat dan sumber daya dari beberapa server MCP, beberapa lokal dan beberapa jarak jauh. Server MCP stdio lokal lebih cepat, tetapi menimbulkan masalah keamanan jika berasal dari penyedia pihak ketiga. Server MCP jarak jauh memiliki masalah jaringan dan otentikasi yang sama seperti API jarak jauh lainnya. Berikut adalah diagram arsitektur MCP yang mengilustrasikan hal ini:



Gambar 7.1 Arsitektur MCP

Protokol sebenarnya yang digunakan server dan klien MCP untuk berkomunikasi memiliki dua lapisan: lapisan data dan lapisan transport:

- **Lapisan data** menentukan protokol berbasis JSON-RPC untuk komunikasi klien-server, yang mencakup manajemen siklus hidup, primitif inti, seperti alat, sumber daya, dan perintah, serta pemberitahuan.
- **Lapisan transport** menentukan mekanisme dan saluran komunikasi yang memungkinkan pertukaran data antara klien dan server, termasuk pembentukan koneksi khusus transport, pembingkaian pesan, dan otorisasi.

Mari kita uraikan komponen-komponen utama MCP.

Server MCP

Server MCP adalah program yang mengekspos alat (fungsi yang dapat dipanggil) dan sumber daya (informasi kontekstual) ke sistem berbasis LLM melalui MCP. Ini memungkinkan aplikasi host LLM yang sesuai untuk menemukan, memanggil, dan menggunakan kemampuan dengan cara yang dapat diprediksi, dapat disusun, dan tidak bergantung pada protokol transport. Setiap server MCP dapat mendukung beberapa versi spesifikasi. Versi aktual dinegosiasikan ketika klien MCP membuat koneksi ke server MCP.

Meskipun server MCP dapat mengekspos sumber daya dan bahkan menawarkan dukungan prompt, kita akan fokus pada aspek alat di sini. Sumber daya dapat diekspos sebagai alat (misalnya, tindakan `list_resources()` dan `get_resource()`), sehingga tidak perlu konsep terpisah. Manajemen prompt dapat dikatakan sebagai masalah tingkat aplikasi dan bukan masalah server MCP individual mana pun. Sekali lagi, jika Anda ingin, karena alasan tertentu, server MCP Anda menawarkan templat prompt, Anda selalu dapat mengekspos alat get_prompt_template(). Menurut pendapat saya, penambahan sumber daya (yang dapat Anda anggap sebagai RAG versi sederhana) dan prompt ke MCP adalah kesalahan yang hanya mempersulit protokol dan membuat kurva pembelajaran lebih curam. Pengalaman Anda mungkin berbeda, jadi jangan ragu untuk menjelajahi fitur-fitur ini. Mulai sekarang, kita hanya akan mempertimbangkan aspek alat dari MCP.

Padanan AI-6 dari server MCP adalah seperangkat alat terkait, seperti alat sistem file. Namun, alat AI-6 ditemukan dan diimpor sebagai modul Python ke dalam sistem AI agen melalui kerangka kerja AI-6. Server MCP, di sisi lain, berjalan sebagai proses independen baik secara lokal maupun jarak jauh. Klien MCP akan berkomunikasi dengan setiap server MCP melalui protokol transport yang sesuai. Sistem AI dapat menggunakan beberapa server MCP, masing-masing menyediakan serangkaian alat yang berbeda.

Perhatikan bahwa sistem AI akan membutuhkan klien MCP khusus untuk setiap server MCP yang ingin digunakannya. Klien MCP bertanggung jawab untuk menemukan alat yang tersedia dan memanggilnya sesuai kebutuhan.

Klien MCP

Klien MCP adalah komponen yang berinteraksi dengan server MCP untuk menemukan dan memanggil alat. Klien ini tertanam langsung di host MCP (sistem AI agen) atau diabstraksikan oleh kerangka kerja seperti AI-6. Setiap klien MCP bertanggung jawab untuk terhubung ke satu server MCP. Spesifikasi MCP tidaklah sederhana, sehingga sebagian besar

klien MCP menggunakan pustaka yang mengimplementasikan protokol dan mengekspos API yang mudah digunakan untuk berinteraksi dengan server MCP. Saya sangat merekomendasikan penggunaan pustaka daripada mengimplementasikan protokol dari awal.

Pertama, mengapa Anda ingin menciptakan kembali roda? Tidak banyak inovasi atau peningkatan kinerja yang dapat diperoleh dengan mengimplementasikan protokol sendiri. Kedua, MCP adalah protokol kompleks yang terus berkembang. Jika Anda membangun klien MCP sendiri berdasarkan spesifikasi, Anda harus terlebih dahulu memastikan bahwa klien tersebut benar-benar berfungsi dengan benar, dan kemudian Anda harus memeliharanya dan mengikuti perubahan spesifikasi yang terjadi. Ini adalah banyak pekerjaan tanpa manfaat.

Komponen AI-6 yang setara dengan klien MCP adalah mekanisme penemuan mesin yang memindai direktori alat, menemukan kelas Python yang mengimplementasikan kelas dasar abstrak Tool, mengimpornya secara dinamis, dan menginstansiasikannya. Pada titik itu, mesin dapat memanggil alat apa pun dengan memanggil metode run() ini.

Host MCP

Host MCP adalah sistem AI agen yang menggunakan klien MCP untuk menemukan dan memanggil alat dari server MCP. Jika MCP menggunakan kerangka kerja AI yang mendukung MCP, maka ia bahkan tidak perlu menyadari MCP. Kerangka kerja akan mengekspos loop agen dengan cara yang tidak bergantung pada MCP, dan sistem AI agen hanya perlu menyediakan konfigurasi yang tepat untuk alat MCP yang ingin digunakannya.

Kita akan segera melihat bagaimana semua komponen ini bekerja bersama dalam kerangka kerja AI-6. Untuk saat ini, mari kita lihat perbedaan penting antara server MCP lokal dan jarak jauh.

Server MCP Lokal Versus Jarak Jauh

Server MCP dapat berupa lokal atau jarak jauh. Server MCP lokal berjalan pada mesin yang sama dengan klien MCP, sedangkan server MCP jarak jauh berjalan pada mesin yang berbeda, mungkin di cloud. Pilihan antara server lokal dan jarak jauh bergantung pada kasus penggunaan dan sumber daya yang tersedia.

Klien MCP dapat terhubung ke beberapa server MCP, baik lokal maupun jarak jauh. Saat terhubung ke server MCP lokal, klien MCP akan menggunakan transport stream **input/output standar** (STDIO). Saat terhubung ke server MCP jarak jauh, klien MCP akan menggunakan transport HTTP Streamable. Perhatikan bahwa informasi ini akurat untuk versi spesifikasi MCP saat ini (https://modelcontextprotocol.io/specification/2025-11-25). Terlepas dari transportnya, urutan pesannya sama, dan format pesannya selalu JSON-RPC 2.0.

Manfaat Menggunakan MCP

Untuk sistem AI berbasis agen, MCP menawarkan beberapa manfaat utama bagi berbagai pemangku kepentingan. Mari kita uraikan untuk kerangka kerja AI berbasis agen, penyedia alat, dan pengembang sistem AI berbasis agen:

- **MCP untuk kerangka kerja AI berbasis agen**: Kerangka kerja AI berbasis agen yang mendukung MCP, seperti AI-6, LangGraph, AutoGen, atau CrewAI, secara otomatis menyediakan akses ke ekosistem alat dan sumber daya yang saling beroperasi yang sangat besar dan terus berkembang. Saat ini, ini adalah persyaratan dasar untuk kerangka kerja AI berbasis agen. Perhatikan bahwa menawarkan dukungan MCP bukan

berarti mengabaikan alat kustom. AI-6, misalnya, mendukung MCP dan alat kustom. Kita akan membahas mengapa hal ini penting nanti di bab ini.

- **MCP untuk penyedia alat**: Penyedia alat mendapat manfaat dari MCP karena, dengan mempublikasikan di server MCP, mereka secara otomatis kompatibel dengan sistem AI apa pun yang menggunakan kerangka kerja AI dengan dukungan MCP. Tidak perlu menyediakan pembungkus khusus untuk setiap kerangka kerja AI di luar sana atau mendokumentasikan cara menggunakannya dari kerangka kerja yang berbeda. Penyedia alat dapat membangun sekali terhadap antarmuka MCP dan mencapai kompatibilitas yang luas. Ini membuka pintu bagi ekosistem alat yang kaya dan dapat digunakan kembali yang dapat disusun menjadi sistem yang lebih besar, seperti API dalam perangkat lunak tradisional. MCP juga mempermudah untuk mendefinisikan metadata seperti kemampuan alat, batasan keamanan, dan model biaya. Karena ekosistem MCP berkembang pesat, pengembang alat dapat mengandalkan komunitas untuk menyediakan registri, mekanisme penemuan, dan infrastruktur lainnya. Manfaat utama lain dari MCP bagi pengembang alat adalah mereka dapat menggunakan bahasa pemrograman apa pun yang mereka pilih karena protokol ini tidak bergantung pada bahasa. Ini membuka pintu untuk menulis alat MCP dengan cepat menggunakan bahasa dinamis seperti Python, TypeScript, atau bahkan skrip shell. Namun jika Anda menulis server MCP kelas berat yang membungkus sistem kritis dan Anda membutuhkan banyak kontrol atau untuk mengoptimalkan kinerja, Anda menggunakan sesuatu seperti Rust.
- **MCP untuk pengembang sistem AI agen**: Ini mudah. Sebagai pengembang sistem AI agen, Anda harus menggunakan kerangka kerja AI agen yang mendukung MCP, seperti AI-6, LangGraph, AutoGen, CrewAI, dan setiap kerangka kerja AI agen terkemuka lainnya. Dengan memilih kerangka kerja yang sesuai dengan MCP, Anda segera mendapatkan akses ke ekosistem alat yang saling beroperasi yang berkembang pesat.

Manfaat MCP telah memunculkan ekosistem yang sangat aktif, yang akan kita jelajahi selanjutnya.

Ekosistem MCP

Ekosistem MCP dengan cepat menjadi fondasi untuk membangun dan meningkatkan skala sistem AI berbasis agen. Ekosistem ini menstandarisasi bagaimana agen, alat, sistem memori, dan lapisan orkestrasi berinteraksi, mengubah integrasi yang terfragmentasi menjadi arsitektur yang kohesif dan dapat disusun. Ekosistem ini mencakup koleksi kerangka kerja yang sesuai dengan MCP yang terus berkembang seperti AI-6, LangGraph, dan AutoGen, bersama dengan penyedia alat, backend memori, dan antarmuka kontrol. Pendekatan modular ini memungkinkan pengembang untuk mencampur dan mencocokkan komponen dengan gesekan minimal, sehingga memudahkan untuk bereksperimen, menerapkan, dan mengembangkan sistem multi-agen yang kompleks.

Tidak ada kekhawatiran tentang ketergantungan vendor atau masalah kompatibilitas, karena standar MCP memastikan bahwa alat apa pun yang sesuai dapat digunakan dan kerangka kerja apa pun dapat diganti tanpa kehilangan investasi pada alat dan sumber daya.

Ini adalah perubahan besar bagi komunitas AI berbasis agen, karena memungkinkan pengembang untuk fokus membangun aplikasi mereka menggunakan kerangka kerja apa pun yang mereka inginkan, dengan mengetahui bahwa mereka akan dapat menggunakan alat MCP dengan kerangka kerja mana pun.

Tempat resmi untuk menemukan server MCP saat ini adalah halaman **Server Contoh** di situs web modelcontextprotocol.io, yang mencantumkan implementasi referensi dalam berbagai bahasa serta integrasi resmi oleh berbagai perusahaan pihak ketiga. Terdapat banyak situs web komunitas yang didedikasikan untuk mengindeks dan mencantumkan server MCP yang ada. Berikut beberapa contohnya:

- https://mcpfinder.io
- https://mcp.so
- https://remote-mcp-servers.com

Berhati-hatilah sebelum menggunakan server MCP yang kurang dikenal, terutama jika Anda mengunduh dan menjalankannya secara lokal, karena server tersebut memiliki semua izin pengguna yang memulai server.Baiklah. Sekarang kita telah memahami MCP dan manfaatnya dengan baik, mari kita lihat cara membangun server dan klien MCP, dan cara mengintegrasikan MCP ke dalam kerangka kerja AI-6.

7.4 MEMBANGUN SERVER MCP

Membangun server MCP dengan berbagai bahasa pemrograman dan pustaka dapat mengajarkan Anda banyak hal tentang manfaat MCP. Pengetahuan, keterampilan, dan bahkan server MCP itu sendiri dapat ditransfer ke sistem atau kerangka kerja AI agenik modern apa pun.

Kita akan membangun dua server MCP yang mengekspos fungsionalitas yang mirip atau identik dengan beberapa alat AI-6 kustom, seperti alat sistem file dan alat GitHub. Proses ini akan memungkinkan kita untuk membandingkan dua implementasi, satu hanya untuk AI-6 dan satu untuk MCP.

Server MCP Sistem File

Mari kita tinjau secara singkat alat sistem file kustom AI-6. Seperti yang Anda ingat, semuanya merupakan subkelas dari kelas dasar CommandTool dan mengikuti pola yang serupa. Satu-satunya pengecualian adalah alat Echo, yang digunakan untuk membuat file baru:

```
for file in py/backend/tools/file_system/*.py; do
    if [ -f "$file" ]; then
        echo "=== $file ==="
        cat "$file"
        echo
    fi
done

### py/backend/tools/file_system/awk.py ===
from backend.tools.base.command_tool import CommandTool

class Awk(CommandTool):
    def __init__(self, user: str | None = None):
```

```
            super().__init__(
                command_name='awk', user=user,
                doc_link='https://www.gnu.org/software/gawk/manual/gawk.html'
            )

### py/backend/tools/file_system/cat.py ===
from backend.tools.base.command_tool import CommandTool

class Cat(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(
            command_name='cat', user=user,
            doc_link='https://www.gnu.org/software/coreutils/manual/html_node/
cat-invocation.html')


### py/backend/tools/file_system/echo.py ===
import os
import sh
from backend.object_model import Tool, Parameter

class Echo(Tool):
    def __init__(self, user: str | None = None):
        self.user = user

        desc = 'Write content to a file, creating any necessary directories.'
        super().__init__(
            name="echo",
            description=desc,
            parameters=[
                Parameter(
                    name="file_path", type="string",
                    description="The path of the file to write to."),
                Parameter(
                    name="content", type="string",
                    description="The content to write to the file."),
            ],
            required={'file_path', 'content'}
        )

    def run(self, **kwargs):
        filename = kwargs['file_path']
        content = kwargs['content']
        dir_path = os.path.dirname(filename)

        if self.user is not None:
            sh.sudo('-u', self.user, 'mkdir', '-p', dir_path)
            sh.sudo('-u', self.user, 'tee', filename, _in=content,
                _out=os.devnull)

            return f"Content written to {filename} as user {self.user}"

        os.makedirs(dir_path, exist_ok=True)
        with open(filename, 'w') as file:
            file.write(content)

        return f"Content written to {filename}"

### py/backend/tools/file_system/ls.py ===
```

```
from backend.tools.base.command_tool import CommandTool

class Ls(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(command_name='ls', user=user,doc_link='https://
www.gnu.org/software/coreutils/manual/html_node/ls-invocation.html')

### py/backend/tools/file_system/patch.py ===
from backend.tools.base.command_tool import CommandTool

class Patch(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(
            command_name='patch', user=user,
            doc_link='https://www.gnu.org/software/diffutils/manual/html_node/patch-
Invocation.html')

### py/backend/tools/file_system/pwd.py ===
from backend.tools.base.command_tool import CommandTool

class Pwd(CommandTool):
    def __init__(self, user: str | None = None):
        doc_link = 'https://www.gnu.org/software/coreutils/manual/html_node/pwd-
invocation.html'
        super().__init__('pwd', user=user, doc_link=doc_link)

### py/backend/tools/file_system/sed.py ===
from backend.tools.base.command_tool import CommandTool

class Sed(CommandTool):
    def __init__(self, user: str | None = None):
        super().__init__(
            command_name='sed', user=user,
            doc_link='https://www.gnu.org/software/sed/manual/sed.html')
```

Server MCP sistem file kami akan menggantikan beberapa alat ini dengan fungsionalitas yang setara. Semua alat MCP AI-6 bawaan berada di direktori `py/backend/mcp-tools`. Mari kita lihat server MCP `fs_mcp_server.py`. Server ini menggunakan SDK Python MCP resmi. Kelas FastMCP melakukan semua pekerjaan berat dalam mendaftarkan alat dan mengeksposnya melalui berbagai transport MCP. Modul menginstansiasikannya setelah impor yang diperlukan:

```
import shlex
from mcp.server.fastmcp import FastMCP
import sh

mcp = FastMCP("FileSystem Tools", "")
```

Server MCP sistem file berisi beberapa alat dalam satu modul. Setiap alat diimplementasikan sebagai fungsi tunggal yang diberi dekorator @mcp.tool(). Mirip dengan kelas dasar abstrak AI-6 Tool yang menandai sebuah kelas sebagai alat, dekorator @mcp.tool() menandai sebuah fungsi sebagai alat:

```
@mcp.tool()
def ls(args: str) -> str:
    """ls tool. See https://www.gnu.org/software/coreutils/manual/html_node/ls-
invocation.html"""
    parsed_args = shlex.split(args)
    return sh.ls(*parsed_args)

@mcp.tool()
def cat(args: str) -> str:
    """cat tool. See
https://www.gnu.org/software/coreutils/manual/html_node/cat-invocation.html"""
    parsed_args = shlex.split(args)
    return sh.cat(*parsed_args)

@mcp.tool()
def pwd(args: str) -> str:
    """pwd tool. See
https://www.gnu.org/software/coreutils/manual/html_node/pwd-invocation.html"""
    parsed_args = shlex.split(args) if args.strip() else []
    return sh.pwd(*parsed_args)

@mcp.tool()
def mkdir(args: str) -> str:
    """mkdir tool. See
https://www.gnu.org/software/coreutils/manual/html_node/mkdir-invocation.html"""
    parsed_args = shlex.split(args)
    return sh.mkdir(*parsed_args)

@mcp.tool()
def cp(args: str) -> str:
    """cp tool. See https://www.gnu.org/software/coreutils/manual/html_node/cp-
invocation.html"""
    parsed_args = shlex.split(args)
    return sh.cp(*parsed_args)
```

Seperti yang Anda lihat, implementasi sebenarnya dari setiap alat MCP sangat mirip dengan kelas CommandTool dari AI-6. Ia menguraikan argumen dan memanggil alat baris perintah target menggunakan sh. Berikut adalah metode run() dari kelas CommandTool, yang hanya menambahkan kemampuan untuk dijalankan sebagai pengguna terpisah:

```
def run(self, **kwargs):
    args = shlex.split(kwargs['args'])
    if self.user is not None:
        return sh.sudo('-u', self.user, self.command_name, *args)
    else:
        return getattr(sh, self.command_name)(*args)
```

Dengan server MCP, Anda dapat mengontrol pengguna OS saat Anda meluncurkan server MCP sebagai sebuah proses.

Singkatnya, mengimplementasikan server MCP di Python itu sederhana jika menggunakan SDK Python resmi. Perhatikan bahwa jika Anda perlu mengkonfigurasi server

MCP, Anda perlu mencari cara sendiri, karena SDK Python MCP tidak menawarkan bantuan apa pun di sini. Anda dapat meneruskan argumen baris perintah, menggunakan variabel lingkungan, atau file konfigurasi. Tetapi beberapa server MCP lebih rumit. Mari kita lihat server MCP GitHub.

Server MCP Github

Server MCP GitHub diimplementasikan dalam Bash karena dua alasan. Pertama, untuk menunjukkan kemampuan menulis server MCP dalam bahasa apa pun. Kedua, untuk melihat lebih dekat protokol itu sendiri, karena tidak ada pustaka yang melakukan semua pekerjaan berat. Kita harus langsung memproses dan menghasilkan pesan JSON-RPC dan mengikuti siklus hidup MCP.

Mari kita uraikan langkah demi langkah. Pesan JSON-RPC memiliki format standar berupa objek dengan bidang method, id, dan params. Penting juga untuk menserialisasi pesan JSON yang kita kembalikan ke dalam satu baris, karena beberapa klien mengharapkannya. Fungsi `emit_json()` mengganti karakter baris baru dengan spasi dan menangani tugas ini:

```
#!/bin/bash

# GitHub MCP Server - implements MCP protocol for GitHub CLI tools (newline-
stripped JSON responses)

# Helper: output compact JSON from heredoc
emit_json() {
    tr -d '\n' | tr -s ' '
    echo
}
```

Sekarang, kita dapat memulai perulangan menunggu pesan MCP dari klien dan memberikan respons. Proses ini berjalan tanpa henti, menganalisis pesan yang masuk dan memberikan respons yang sesuai:

```
# Read JSON-RPC messages from stdin and respond
while read -r line; do
    method=$(echo "$line" | jq -r '.method' 2>/dev/null)
    id=$(echo "$line" | jq -r '.id' 2>/dev/null)
    params=$(echo "$line" | jq -r '.params' 2>/dev/null)

    ...
done
```

Mari kita lihat pesan apa yang diharapkan dan bagaimana responsnya. Setiap pesan yang masuk diproses sesuai dengan metodenya dalam pernyataan case besar. Pesan pertama adalah "initialize", yang dikirim oleh klien MCP untuk memulai koneksi. Server MCP perlu melaporkan versi protokol yang didukung dan informasi server. Bagian capabilities mungkin kosong:

```
    case "$method" in
        "initialize")
            cat <<EOF | emit_json
```

```
{
  "jsonrpc": "2.0",
  "id": $id,
  "result": {
    "protocolVersion": "2024-11-05",
    "capabilities": {
      "tools": {},
      "resources": {}
    },
    "serverInfo": {
      "name": "GitHub CLI Tools",
      "version": "1.0.0"
    }
  }
}
EOF
            ;;
```

Klien mengirimkan notifikasi bahwa inisialisasi berhasil. Tidak ada respons yang diharapkan:

```
"notifications/initialized")
    # Notification - no response needed
    ;;
```

Pesan "tools/list" dipanggil sekali, dan server merespons dengan daftar alatnya (hanya satu dalam kasus ini), yang mencakup nama alat, deskripsi, dan skema input untuk parameter yang sudah dikenal. Kita hanya memiliki satu parameter yang wajib, yaitu args, yang merupakan argumen baris perintah untuk perintah CLI GitHub:

```
        "tools/list")
            cat <<EOF | emit_json
{
  "jsonrpc": "2.0",
  "id": $id,
  "result": {
    "tools": [
      {
        "name": "gh",
        "description": "Execute GitHub CLI commands",
        "inputSchema": {
          "type": "object",
          "properties": {
            "args": {
              "type": "string",
              "description": "GitHub CLI command arguments"
            }
          },
          "required": ["args"]
        }
      }
    ]
  }
}
EOF
            ;;
```

Sekarang, klien MCP mengetahui semua alat yang disediakan oleh server MCP ini dan dapat memanggilnya bila perlu. Ini adalah kasus yang paling rumit. Pertama, kode memverifikasi bahwa nama alat yang diminta memang "gh" (satu-satunya nama alat yang didukung). Kemudian, ia menjalankan perintah gh $args 2>&1 , menangkap output standar dan kesalahan standar. Ia juga menangkap kode keluar dan kemudian melakukan escape yang tepat pada hasilnya, dan akhirnya menyusun dan mengembalikan pesan respons JSON-RPC yang mencakup hasil escape dari perintah gh:

```
if [ "$tool_name" = "gh" ]; then
                result=$($gh $args 2>&1)
                exit_code=$?
                escaped=$(echo "$result" | jq -Rs .)

                cat <<EOF | emit_json
{
  "jsonrpc": "2.0",
  "id": $id,
  "result": {
    "content": [
      {
        "type": "text",
        "text": $escaped
      }
    ]
  }
}
EOF
```

Jika nama alatnya bukan "gh", maka akan muncul pesan kesalahan:

```
else
                cat <<EOF | emit_json
{
  "jsonrpc": "2.0",
  "id": $id,
  "error": {
    "code": -32602,
    "message": "Unknown tool: $tool_name"
  }
}
EOF
            fi
            ;;
```

Terakhir, jika pesan memiliki metode yang tidak dikenal, ia akan mengembalikan pesan kesalahan yang sesuai:

```
   *)
            if [ "$id" != "null" ]; then
                cat <<EOF | emit_json
  {
    "jsonrpc": "2.0",
    "id": $id,
```

```
    "error": {
        "code": -32601,
        "message": "Method not found: $method"
    }
}
EOF

                fi
                ;;
    esac
done
```

Itulah kesimpulan dari server MCP Bash tingkat rendah kita. Membangunnya membutuhkan banyak pekerjaan, dan jika protokolnya berkembang, maka Anda perlu memperbarui server Anda. Selain itu, server ini hanya mengimplementasikan transport STDIO, sehingga tidak dapat digunakan sebagai server MCP jarak jauh. Saya sangat menyarankan untuk menggunakan salah satu SDK MCP resmi saat Anda membangun server MCP Anda sendiri. Mari kita lihat apa yang dibutuhkan untuk membangun klien MCP.

7.5 MEMBANGUN KLIEN MCP

Klien MCP terhubung ke server MCP dan mengontrol interaksi. Kita tidak akan membahas detail pembuatan klien MCP dari awal, seperti yang kita lakukan dengan server MCP GitHub Bash. Sebaliknya, kita akan memanfaatkan SDK Python MCP. AI-6 menyediakan wrapper MCPClient sendiri yang dapat terhubung ke beberapa server MCP, mengembalikan alat-alatnya, dan, kemudian, memanggil alat apa pun.

Kelas MCPCLIENT

Kelas MCPClient kita berada di modul `mcp_client.py`. Mari kita mulai dengan impornya. Perhatikan bahwa klien MCP dirancang untuk operasi asinkron. Dari paket mcp, ia mengimpor kelas ClientSession dan StdioServerParameter, dan kemudian mengimpor `stdio_client` untuk server MCP lokal dan `sse_client` untuk server MCP jarak jauh:

```
import asyncio
from contextlib import AsyncExitStack
from urllib.parse import urlparse

from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client
from mcp.client.sse import sse_client
```

Konstruktornya cukup sederhana dan mengelola kamus sesi dan kamus alat server yang memetakan server MCP ke daftar alatnya. Ia juga membuat tumpukan keluar asinkron untuk pembersihan yang tepat:

```
class MCPClient:
    """Standalone MCP client for connecting to and interacting with MCP servers."""
    def __init__(self):
        self.sessions: dict[str, ClientSession] = {}
        self._server_tools: dict[str, list[dict]] = {}
        self.exit_stack = AsyncExitStack()
```

Ketika klien terhubung ke server MCP, mereka memanggil metode connect_to_server. Mari kita lihat cara kerjanya.

Menghubungkan ke server MCP

Metode async `connect_to_server()` melakukan banyak pekerjaan. Mari kita periksa bagian demi bagian. Metode ini menerima ID server (ID unik dan sembarang yang dikendalikan oleh sistem AI) yang mengidentifikasi target dan jalur (untuk server MCP lokal) atau URL (untuk server MCP jarak jauh). Jika sudah terhubung ke server tersebut, metode ini akan mengembalikan alat-alat dari server tersebut:

```
async def connect_to_server(
    self, server_id: str, server_path_or_url: str
) -> list[dict]:
    """Connect to a single MCP server and return its tools."""
    if server_id in self.sessions:
        # Already connected, return cached tools
        return self._server_tools.get(server_id, [])
```

Selanjutnya, ia menentukan apakah itu server MCP lokal atau jarak jauh:

```
# Check if this is a URL (remote server) or file path (local server)
        parsed = urlparse(server_path_or_url)
        is_url = parsed.scheme in ('http', 'https')
```

Jika itu server jarak jauh, ia membuat transport menggunakan klien **Server-Sent Events** (SSE), yang digunakannya untuk menghasilkan objek ClientSession. Jika terjadi kesalahan, ia mencetak pesan kesalahan dan memunculkan kembali pengecualian tersebut:

```
        if is_url:
            print(
        f"Connecting to remote MCP server: {server_path_or_url}")
            try:
                transport = await self.exit_stack.enter_async_context(
                    sse_client(server_path_or_url))
                read, write = transport
                session = await self.exit_stack.enter_async_context(
                    ClientSession(read, write))
                print(f"Successfully connected to {server_path_or_url}")
            except Exception as e:
                print(f"Failed to connect to remote MCP server
{server_path_or_url}: {e}")
                raise
```

Namun, jika itu adalah server lokal, ia akan memeriksa (hanya berdasarkan ekstensi file) apakah itu server MCP Python, TypeScript, atau Shell. Jika tidak, ia akan menimbulkan pengecualian. Ini hanyalah pilihan karena tidak ada rencana untuk mendukung bahasa lain untuk server MCP lokal untuk AI-6. Sangat valid untuk mendukung bahasa lain atau, bahkan lebih baik, citra Docker yang dapat berisi server MCP dalam bahasa apa pun:

```
else:
    # Local file-based server
    if server_path_or_url.endswith('.py'):
        command = "python"
    elif server_path_or_url.endswith('.sh'):
        command = "bash"
    elif server_path_or_url.endswith('.js'):
        command = "node"
    else:
        raise ValueError(f"Unsupported server type: {server_path_or_url}")
```

Selanjutnya, ia menyiapkan parameter server, yang hanya berupa jalur ke file server MCP. Tidak ada argumen baris perintah atau variabel lingkungan tambahan yang disediakan. Kita mungkin ingin mendukung lebih banyak konfigurasi di masa mendatang. Kemudian, ia menghasilkan transport STDIO dan menggunakannya untuk membuat objek ClientSession lokal:

```
server_params = StdioServerParameters(
    command=command,
    args=[server_path_or_url],
    env=None
)

stdio_transport = await self.exit_stack.enter_async_context(
    stdio_client(server_params))
stdio, write = stdio_transport
session = await self.exit_stack.enter_async_context(
    ClientSession(stdio, write))
```

Pada titik ini, kita memiliki objek ClientSession (baik lokal maupun jarak jauh), dan kita dapat mengambil alat-alat dari server MCP dengan memanggil `session.list_tools()`. Di balik layar, panggilan ini bertukar pesan JSON-RPC yang tepat dengan server MCP sesuai dengan protokol:

```
# Initialize session with timeout
await asyncio.wait_for(session.initialize(), timeout=10.0)

# List tools with timeout
response = await asyncio.wait_for(
    session.list_tools(), timeout=10.0)
tools = [{
        "name": tool.name,
        "description": tool.description,
        "parameters": tool.inputSchema
    } for tool in response.tools]
```

Kemudian, kita menyimpan sesi dan alat-alat tersebut untuk diambil kembali nanti jika diperlukan, dan mengembalikan alat-alat tersebut:

```
# Cache the session and tools
self.sessions[server_id] = session
self._server_tools[server_id] = tools
```

```
    return tools
```

Perlu dicatat bahwa batas waktu 10 detik seharusnya cukup dalam sebagian besar kasus, tetapi dalam kasus langka di mana server MCP memiliki urutan inisialisasi yang panjang, mungkin lebih baik untuk menyediakan durasi batas waktu yang dapat dikonfigurasi.

Setelah sistem AI agen terhubung ke server MCP, langkah selanjutnya adalah memanggil satu atau lebih alat yang disediakannya.

Memanggil Alat MCP

Sistem AI agen menerima daftar alat dari setiap server MCP ketika memanggil metode `connect_to_server()`. Kapan saja, sistem dapat memutuskan untuk memanggil salah satu alat ini dengan memanggil metode `invoke_tool()`, yang menerima ID server, nama alat, dan kamus argumen untuk alat tersebut. Panggilan tersebut mengembalikan respons dari alat (kecuali jika terjadi pengecualian). Pertama, sistem memeriksa apakah ada sesi aktif yang terkait dengan ID server yang diminta dan menimbulkan pengecualian jika tidak ada:

```
async def invoke_tool(
    self, server_id: str, tool_name: str, tool_args: dict
) -> str:
    """Invoke a specific tool on the specified MCP server."""
    session = self.sessions.get(server_id)
    if not session:
        raise RuntimeError(f"No active session for server '{server_id}'.
Connect to server first.")
```

Kemudian, program memanggil alat tersebut dengan menggunakan metode `session.tool_call()` dengan nama alat dan argumen. Program mengembalikan respons dari alat tersebut (yang mungkin kosong). Jika panggilan alat mengalami batas waktu atau menimbulkan pengecualian lain, program akan mencetak pesan kesalahan dan memunculkan kembali pengecualian tersebut:

```
        try:
            print(f"Invoking tool {tool_name} on server {server_id} with args:
{tool_args}")
            # Add timeout to prevent hanging on slow/unresponsive servers
            result = await asyncio.wait_for(
                session.call_tool(tool_name, tool_args),
                timeout=30.0  # 30 second timeout
            )
            print(f"Tool {tool_name} completed successfully")
            return result.content[0].text if result.content else ""
        except asyncio.TimeoutError:
            print(f"Tool invocation timed out for {server_id}:
{tool_name} after 30 seconds")
            raise RuntimeError(f"Tool invocation timed out for {server_id}:
{tool_name} after 30 seconds")
        except Exception as e:
            print(f"Tool invocation failed for {server_id}:{tool_name}: {e}")
            raise
```

Kelas ini menawarkan metode pembantu tambahan. Metode `get_server_tools()` dapat digunakan oleh sistem AI berbasis agen jika sistem tersebut lebih memilih untuk mengambil alat dari server secara langsung saat dibutuhkan:

```
def get_server_tools(self, server_id: str) -> list[dict]:
    """Get cached tools for a server."""
    return self._server_tools.get(server_id, [])
```

Metode `is_connected()` memeriksa apakah ID server target terkait dengan sesi:

```
def is_connected(self, server_id: str) -> bool:
    """Check if connected to a server."""
    return server_id in self.sessions
```

Metode `disconnect_server()` menghapus ClientSession dan menghapus alat-alatnya. Ini dapat berguna untuk server MCP yang menggunakan banyak sumber daya dan hanya dibutuhkan untuk fase-fase tertentu dari sebuah percakapan:

```
async def disconnect_server(self, server_id: str):
    """Disconnect from a specific server."""
    if server_id in self.sessions:
        # Session cleanup handled by exit_stack
        del self.sessions[server_id]
        if server_id in self._server_tools:
            del self._server_tools[server_id]
```

Terakhir, metode cleanup() membersihkan kamus sesi dan `_server_tools`:

```
async def cleanup(self):
    """Cleanup all resources."""
    self.sessions.clear()
    self._server_tools.clear()
    await self.exit_stack.aclose()
```

Mari kita lihat bagaimana cara memperluas kerangka kerja AI-6 dengan dukungan MCP di samping sistem alat miliknya sendiri.

7.6 MENGINTEGRASIKAN MCP KE DALAM KERANGKA KERJA AI-6

AI-6 memiliki sistem alatnya sendiri, termasuk penemuan, pendaftaran, dan kelas dasar abstrak yang disebut Tool, dan seluruh siklus agen dari mesin dibangun di sekitar konsep-konsep ini. Bagaimana kita mengintegrasikan sistem alat baru, seperti MCP? Kita dapat memanfaatkan **teorema fundamental rekayasa perangkat lunak** (FTSE), yang menyatakan, "*Kita dapat menyelesaikan masalah apa pun dengan memperkenalkan satu tingkat tambahan dari pengalihan*." Dalam konteks ini, artinya membungkus semua alat MCP dengan lapisan pengalihan yang menyerupai alat AI-6. Masuklah kelas MCPTool universal, yang dapat

membungkus alat MCP (lokal atau jarak jauh) dan membuatnya dapat diakses oleh AI-6 seolah-olah itu adalah salah satu alatnya sendiri. Mari kita lihat bagaimana keajaiban itu terjadi.

Kelas MCPTOOL Universal

Kelas MCPTool harus merupakan subkelas dari kelas AI-6 Tool, yang berarti kelas ini perlu mengimplementasikan metode abstraknya. Ini benar-benar satu-satunya persyaratan kompatibilitas. Saat dibuat, kelas ini membutuhkan akses ke instance MCPClient yang dapat dipanggil saat metode run() dari tool dipanggil. Mari kita periksa kodenya.

Berikut adalah impornya. Kita membutuhkan asyncio dan threading untuk menangani MCPClient berbasis asyncio, dan dari model objek AI-6, kita membutuhkan kelas dasar Tool dan definisi Parameter:

```
import asyncio
import threading

from backend.object_model.tool import Tool, Parameter
from backend.mcp_client.mcp_client import MCPCClient
```

Mari kita langsung terjun ke inti permasalahan dan menangani isu rumit tentang adaptasi skema alat MCP ke model objek AI-6. Fungsi `_json_schema_to_parameters()` mengambil kamus skema alat MCP dan mengubahnya menjadi daftar objek Parameter AI-6 dan serangkaian nama parameter yang dibutuhkan. Kode ini cukup sederhana, jadi tidak perlu menjelaskan setiap barisnya:

```
def _json_schema_to_parameters(
    schema: dict
) -> tuple[list[Parameter], set[str]]:
    """Convert JSON schema to Tool parameters and required set."""
    parameters = []
    required = set()

    if not isinstance(schema, dict) or 'properties' not in schema:
        return parameters, required

    # Get required fields
    if 'required' in schema and isinstance(schema['required'], list):
        required = set(schema['required'])

    # Convert properties to parameters
    for prop_name, prop_def in schema['properties'].items():
        param_type = prop_def.get('type', 'string')
        description = prop_def.get('description',
            f'{prop_name} parameter')

        # Map JSON schema types to our parameter types
        if param_type == 'array':
            param_type = 'array'
        elif param_type in ['integer', 'number']:
            param_type = 'number'
        elif param_type == 'boolean':
            param_type = 'boolean'
        else:
            param_type = 'string'
```

```
            parameters.append(Parameter(
                name=prop_name,
                type=param_type,
                description=description
            ))

        return parameters, required
```

Kita akan segera melihat di mana fungsi ini digunakan. Mari kita lanjutkan ke kelas MCPTool itu sendiri. Kelas ini memiliki beberapa field tingkat kelas karena semua instance MCPTool berbagi objek MCPClient yang sama yang mengelola semua instance MCP ClientSession. Sifat asinkron dari objek MCPClient memerlukan pengelolaan yang cermat terhadap kunci klien dan loop peristiwa bersama:

```
class MCPTool(Tool):
    """Tool that uses MCP (Model Context Protocol) servers."""

    # Shared MCP client instance across all MCP tools
    _client: MCPCClient = None
    _client_lock = threading.Lock()
    # Shared event loop for all MCP operations
    _event_loop = None
    _loop_thread = None
    _loop_lock = threading.Lock()
```

Konstruktor menerima ID server, jalur server, atau URL (tergantung apakah itu server MCP lokal atau jarak jauh), dan kamus dengan informasi alat yang diterima dari server MCP. Kode kemudian mengekstrak nama dan deskripsi (atau default) dari kamus informasi alat dan mengkonversi parameter alat MCP ke parameter AI-6 dengan memanggil `_json_schema_to_parameters()`. Kemudian, ia meneruskan informasi alat AI-6 ke kelas dasar Tool dengan memanggil konstruktor super `().__init__()` dan menyimpan ID server, jalur server atau URL, dan nama alat. Ini akan dibutuhkan ketika alat dipanggil nanti:

```
def __init__(
    self, server_id: str, server_path_or_url: str, tool_info: dict
):
    """Initialize from MCP tool information."""
    tool_name = tool_info['name']
    description = tool_info.get(
        'description', f'{server_id} tool: {tool_name}'
    )

    # Convert parameters from JSON schema
    parameters, required = _json_schema_to_parameters(
        tool_info.get('parameters', {})
    )

    super().__init__(
        name=tool_name, description=description,
        parameters=parameters, required=required
    )
```

```
        self.server_id = server_id
        self.server_path_or_url = server_path_or_url
        self.mcp_tool_name = tool_name
```

Seperti yang Anda ingat, MCPClient dikelola pada tingkat kelas. Metode kelas _get_client() membuat instance MCPClient untuk pertama kalinya dan kemudian hanya mengembalikan instance yang sama pada panggilan berikutnya. Metode ini mengimplementasikan pola penguncian ganda (double-checked locking) untuk menghindari pembuatan instance MCPClient lebih dari sekali:

```
@classmethod
def _get_client(cls) -> MCPCClient:
    """Get or create the shared MCP client instance."""
    if cls._client is None:
        with cls._client_lock:
            if cls._client is None:
                cls._client = MCPCClient()
    return cls._client
```

Fungsi `_get_or_create_loop()` mengikuti pola yang sama dan mengembalikan (membuat jika perlu) sebuah event loop asyncio singleton:

```
@classmethod
def _get_or_create_loop(cls):
    """Get or create a shared event loop for MCP operations."""
    # Always use our own managed loop for sync tool execution
    # This avoids "event loop is closed" issues from creating/closing loops
repeatedly
    if cls._event_loop is None or cls._event_loop.is_closed():
        with cls._loop_lock:
            if cls._event_loop is None or cls._event_loop.is_closed():
                cls._event_loop = asyncio.new_event_loop()
    return cls._event_loop
```

Metode `_ensure_connected()` menerima klien dan event loop, mengaturnya secara eksplisit untuk menghindari konflik, menunggu hingga klien terhubung ke server, dan kemudian mengembalikan klien dan event loop:

```
def _ensure_connected(self):
    """Ensure connection to the MCP server."""
    client = self._get_client()
    loop = self._get_or_create_loop()
    if not client.is_connected(self.server_id):
        # Use shared event loop for connection
        asyncio.set_event_loop(loop)
        loop.run_until_complete(
            client.connect_to_server(
                self.server_id, self.server_path_or_url))
    return client, loop
```

Setelah kita menyelesaikan semua kerja keras dalam menghubungkan MCPClient dan menangani semua masalah asyncio, mari kita lihat metode run(), yang dipanggil oleh mesin AI-6 ketika LLM meminta untuk mengeksekusi panggilan alat:

```
def run(self, **kwargs) -> str:
    """Execute the MCP tool with the given arguments."""
    client, loop = self._ensure_connected()

    return loop.run_until_complete(
        client.invoke_tool(self.server_id, self.mcp_tool_name, kwargs)
```

Kode tersebut memanggil metode `_ensure_connected()` dan kemudian mendapatkan klien dan event loop. Keuntungannya adalah tidak perlu menerjemahkan argumen dari representasi AI-6 ke representasi MCPClient, karena keduanya hanyalah kamus str -> object. Metode `invoke_tool()` klien MCP berinteraksi dengan server MCP dan mengembalikan hasilnya sebagai string. Terakhir, metode kelas `cleanup_all()` dengan elegan mematikan mesin asyncio:

```
@classmethod
def cleanup_all(cls):
    """Cleanup all MCP connections. Call this on shutdown."""
    if cls._client is not None:
        # Use our managed loop for cleanup
        loop = cls._get_or_create_loop()
        asyncio.set_event_loop(loop)
        try:
            loop.run_until_complete(cls._client.cleanup())
        finally:
            # Now we can close our managed loop
            if cls._event_loop and not cls._event_loop.is_closed():
                cls._event_loop.close()
            cls._event_loop = None
        cls._client = None
```

Demikianlah pembahasan mengenai kelas MCPTool, tetapi kita masih melewatkan bagian penting dari cerita ini, yaitu bagaimana berbagai server MCP ditemukan dan diinstansiasi.

Penemuan Alat MCP

Kelas AI-6 Engine sebelumnya bertanggung jawab atas penemuan alat, tetapi dengan integrasi MCP, saya mengekstrak fungsi ini ke modul khusus yang disebut tool_manager, yang melakukan penemuan alat asli AI-6, penemuan alat MCP lokal, dan juga menggabungkan server MCP jarak jauh yang telah dikonfigurasi. Mari kita lihat bagaimana pengelola alat bekerja dan kemudian bagaimana Engine menggunakannya untuk mendapatkan akses ke semua alat.

Manajer Alat

Modul `tool_manager.py` mengurangi banyak kompleksitas dari kelas Engine dan membiarkannya fokus pada tugas penting menjalankan loop agen. Mari kita periksa kodenya, dimulai seperti biasa dengan impor, yang seharusnya sudah familiar sekarang. Perhatikan

modul importlib.util dan inspect yang melakukan pekerjaan berat untuk penemuan alat kustom dinamis (sebelumnya di engine):

```
import asyncio
import os
from pathlib import Path
import importlib.util
import inspect

from backend.object_model.tool import Tool
from backend.tools.base.mcp_tool import MCPTool
from backend.mcp_client.mcp_client import MCPCClient
from backend.engine.config import ToolConfig
```

Fungsi `get_tool_dict()` adalah satu-satunya fungsi publik. Penemuan semua alat, baik alat asli AI-6 maupun alat MCP, terjadi di sini. Proses dimulai dengan daftar alat kosong dan kemudian memanggil fungsi-fungsi berikut untuk mendapatkan berbagai jenis alat dan menambahkannya ke daftar alat:

- `_discover_native_tools()`
- `_discover_local_mcp_tools()`
- `_get_remote_mcp_tools()`

Semua alat diturunkan dari kelas dasar Tool, yang diketahui oleh mesin cara mengkonfigurasi dan memanggilnya:

```
def get_tool_dict(tool_config: ToolConfig) -> dict[str, Tool]:
    """Get a dictionary of all available tools from various sources.

    Args:
        tool_config: ToolConfig object containing tool configuration

    Returns:
        Dict mapping tool names to Tool instances
    """
    tools: list[Tool] = []

    # 1. Discover AI-6 native tools
    if tool_config.tools_dir:
        native_tools = _discover_native_tools(
            tool_config.tools_dir,
            tool_config.tool_config
        )
        tools.extend(native_tools)

    # 2. Discover Local MCP tools
    if tool_config.mcp_tools_dir:
        local_mcp_tools = _discover_local_mcp_tools(
            tool_config.mcp_tools_dir
        )
        tools.extend(local_mcp_tools)

    # 3. Get tools of remote MCP servers
    if tool_config.remote_mcp_servers:
        remote_mcp_tools = _get_remote_mcp_tools(
```

```
                tool_config.remote_mcp_servers
            )
            tools.extend(remote_mcp_tools)

    return {tool.name: tool for tool in tools}
```

Kita akan melewati fungsi `_get_local_tools()` karena telah kita bahas secara mendalam di *bab 4*. Mari kita fokus pada penemuan alat MCP.

Fungsi `_discover_local_mcp_tools()` menerima direktori server MCP lokal dan mengembalikan daftar objek MCPTool, yang merupakan gabungan dari semua alat MCP dari semua server MCP yang dikemas sebagai objek MCPTool. Jika direktori server MCP tidak valid, fungsi ini mengembalikan daftar kosong:

```
def _discover_local_mcp_tools(mcp_servers_dir: str) -> list[MCPTool]:
    """Discover MCP tools dynamically by connecting to MCP servers."""
    if not os.path.isdir(mcp_servers_dir):
        return []
```

Kemudian, ia mendefinisikan fungsi async bersarang yang disebut `discover_async()`. Fungsi ini hanya terlihat di dalam fungsi `_discover_local_mcp_tools()`. Ini diperlukan untuk menjembatani dunia sinkron dan asinkron. Mari kita uraikan.

Pertama, ia mendefinisikan daftar alat kosong dan membuat MCPClient baru. Kemudian, ia memulai blok try dan mengulangi semua file di direktori server MCP (ini berasal dari argumen ke fungsi induknya):

```
async def discover_async():
    tools: list[Tool] = []
    client = MCPCClient()

    try:
        # Iterate over all files in the directory
        for file_name in os.listdir(mcp_servers_dir):
```

Untuk setiap item, ia menghitung jalur skrip (ingat bahwa hanya server MCP lokal Python, Node, dan Shell yang didukung). Jika bukan file, maka akan berlanjut ke item berikutnya:

```
script_path = os.path.join(mcp_servers_dir, file_name)

# Check if it's a file
if not os.path.isfile(script_path):
    continue
```

Namun, jika berupa file, maka ia menggunakan nama file tersebut sebagai ID server saat memanggil fungsi `connect_to_server()` klien, yang mengembalikan semua alat dari server MCP tersebut. Ia membuat instance MCPTool baru untuk setiap alat dan menambahkannya ke daftar alat. Jika terjadi kesalahan, ia mencetak pesan kesalahan dan melanjutkan ke alat berikutnya:

```
try:
    # Connect to server and get its tools with timeout
    server_tools = await asyncio.wait_for(
        client.connect_to_server(script_path, script_path),
        timeout=5.0
    )

    # Create MCPTool instances for each tool
    for tool_info in server_tools:
        mcp_tool = MCPTool(script_path, script_path, tool_info)
        tools.append(mcp_tool)

except Exception as e:
    # Skip servers that fail to connect or timeout
    error_msg = str(e) if str(e) else f"{type(e).__name__}: {e}"
    print(f"Warning: Failed to connect to MCP server {script_path}: {error_msg}")
    continue
```

Setelah semua server MCP diproses dan alat-alat gabungannya ditambahkan ke daftar alat, sistem akan melakukan pembersihan klien dan kemudian mengembalikan alat-alat tersebut:

```
finally:
    await client.cleanup()

return tools
```

Semua itu terjadi di dalam fungsi async bersarang, `discover_async()`. Fungsi induk, `_discover_local_mcp_tools()`, memanggil fungsi async menggunakan `asyncio.run()`, yang memungkinkan fungsi sinkron untuk memanggil fungsi async dan menunggu hingga selesai, lalu mengembalikan hasilnya (daftar alat gabungan dari semua server MCP). Jika terjadi kesalahan, fungsi tersebut akan mencetak peringatan dan mengembalikan daftar kosong:

```
# Run async discovery in sync context
try:
    return asyncio.run(discover_async())
except Exception as e:
    print(f"Warning: MCP tool discovery failed: {e}")
    return []
```

Fungsi `_discover_local_mcp_tools()` menangani semua server MCP lokal. Namun, AI-6 juga mendukung server MCP jarak jauh. Mari kita lihat bagaimana kita dapat memasukkannya ke dalam sistem. Fungsi `_get_remote_mcp_tools()` mengikuti cetak biru yang sama dengan `_get_local_mcp_tools()` dengan fungsi async bersarang, tetapi ada beberapa perbedaan penting juga. Jadi, mari kita bahas secara detail. Fungsi ini menerima daftar kamus yang masing-masing berisi nama dan URL server MCP jarak jauh. Jadi, tidak ada pemindaian direktori untuk mencari server MCP. Fungsi ini juga mengembalikan daftar objek Tool yang diinisialisasi ke daftar kosong:

```
def _get_remote_mcp_tools(remote_servers: list[dict]) -> list[Tool]:
    """Connect to remote MCP servers and get their tools.

    Args:
        remote_servers: List of remote server configurations
            Each server config should have: {'url': 'https://...', 'name':
'...'}

    Returns:
        List of MCPTool instances for remote tools
    """
    tools: list[Tool] = []
```

Fungsi bersarang `connect_sync()` cukup mirip dengan fungsi bersarang `discover_sync()` dari `_discover_local_mcp_tools()`, kecuali tidak ada penemuan di sini dengan mengulang file karena semua konfigurasi server MCP jarak jauh telah disediakan. Jadi, fungsi tersebut mengulanginya, mencoba terhubung ke masing-masing server dan mengambil alat-alatnya, lalu membuat objek MCPTool. ID server dalam hal ini adalah nama server dari konfigurasi:

```
    async def connect_async():
        client = MCPCClient()

        try:
            for server_config in remote_servers:
                try:
                    server_url = server_config.get('url')
                    server_name = server_config.get('name', server_url)

                    if not server_url:
                        print(f"Warning: Remote MCP server config missing 'url':
{server_config}")
                        continue

                    # Connect to remote server with timeout
                    server_tools = await asyncio.wait_for(
                        client.connect_to_server(server_name, server_url),
                        timeout=10.0
                    )

                    # Create MCPTool instances for each remote tool
                    for tool_info in server_tools:
                        # Use server_url as script_path for remote tools
                        mcp_tool = MCPTool(
                            server_name, server_url, tool_info
                        )
                        tools.append(mcp_tool)

                except Exception as e:
                    print(f"Warning: Failed to connect to remote MCP server
{server_config}: {e}")
                    continue

        finally:
```

```
            await client.cleanup()
```

Kemudian, seperti sebelumnya, jalankan fungsi bersarang dan kembalikan alat-alat tersebut:

```
    # Run async connection in sync context
    try:
        asyncio.run(connect_async())
    except Exception as e:
        print(f"Warning: Remote MCP server connection failed: {e}")
        return []

    return tools
```

Ini mengakhiri pembahasan mendalam kita tentang modul tool_manager. Mari kita lihat bagaimana mesin mendapatkan manfaat dari pemisahan tanggung jawab ini.

Bagaimana Mesin Menggunakan Pengelola Alat

Mesin sekarang hanya memiliki satu impor yang terkait dengan penemuan alat:

```
from backend.engine import tool_manager
```

Selain itu, konstruktornya hanya memiliki satu baris kode yang memanggil fungsi `get_tool_dict()` dari pengelola alat:

```
self.tool_dict = tool_manager.get_tool_dict(tool_config)
```

Ini sesederhana mungkin, dan meningkatkan arsitektur serta pemeliharaan mesin, serta seluruh proyek AI-6. Berikut adalah seluruh konstruktor kelas Engine, yang sekarang sama sekali tidak memiliki fungsi penemuan alat:

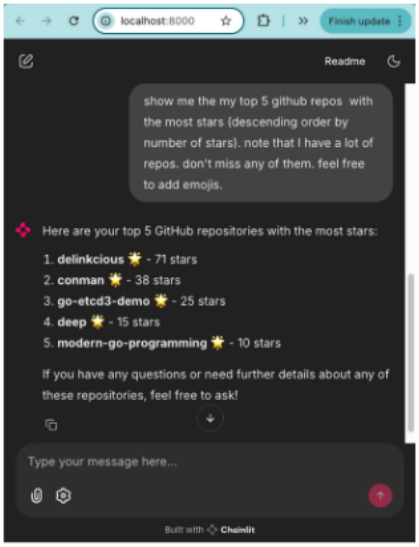
```
class Engine:
    def __init__(self, config: Config):
        self.default_model_id = config.default_model_id

        # Store threshold ratio and calculate token threshold based on default model
        self.summary_threshold_ratio = config.summary_threshold_ratio
        context_window_size = get_context_window_size(
            self.default_model_id)
        self.token_threshold = int(
            context_window_size * config.summary_threshold_ratio
        )

        # Find LLM providers directory
        llm_providers_dir = os.path.join(
            os.path.dirname(os.path.dirname(__file__)), "llm_providers"
        )
        assert os.path.isdir(llm_providers_dir), (
            f"LLM providers directory not found: {llm_providers_dir}"
        )

        # Discover available LLM providers
        self.llm_providers = Engine.discover_llm_providers(
```

```
            llm_providers_dir, config.provider_config
        )
        if not self.llm_providers:
            raise ValueError("No LLM providers found or initialized")
        self.model_provider_map = {
            model_id: llm_provider
            for llm_provider in self.llm_providers
            for model_id in llm_provider.models
        }

        # Discover and initialize all tools
        tool_config = ToolConfig.from_engine_config(config)
        self.tool_dict = tool_manager.get_tool_dict(tool_config)

        # Initialize session and session manager
        self.session_manager = SessionManager(config.memory_dir)
        self.session = Session(config.memory_dir)

        # Session-related attributes
        self.checkpoint_interval = config.checkpoint_interval
        self.message_count_since_checkpoint = 0

        # Instantiate the summarizer with the first LLM provider
        self.summarizer = Summarizer(self.llm_providers[0])

        # Register memory tools with the engine
        self._register_memory_tools()

        # Load previous session if session_id is provided and exists
        if config.session_id:
            available_sessions = self.session_manager.list_sessions()
            if config.session_id in available_sessions:
                self.session = Session(config.memory_dir)  # Create a new session
object
                self.session.load(config.session_id)  # Load from disk
```

Oke. Kita telah membahas semua aspek integrasi MCP ke dalam AI-6. Mari kita lihat penerapannya.

7.7 MENYATUKAN SEMUANYA

Mari kita mulai sesi AI-6 singkat dan lihat alat MCP baru beraksi. Saya akan meminta AI-6 untuk menemukan repositori GitHub saya yang paling populer. Berikut hasil akhirnya:



Gambar 7.2 AI-G Mengakses GitHub Melalui MCP

Ini adalah keberhasilan besar! LLM mampu mengakses GitHub melalui server MCP kami, mengekstrak informasi yang diperlukan, yang juga memerlukan otentikasi, dan menyaring jawaban yang bermanfaat.

Namun, dibutuhkan beberapa kali percobaan karena LLM terus meminta perintah CLI gh yang tidak valid untuk paginasi. Akhirnya, ia hanya meminta kode 300. Ini berhasil dalam kasus ini karena saya hanya memiliki sedikit lebih dari 100 repositori, tetapi ini bukan solusi umum. Namun, seperti yang Anda lihat, ia jelas menggunakan server GitHub MCP (diimplementasikan dari awal dalam Bash):

```
Invoking tool gh on server /Users/gigi/git/ai-six/py/backend/mcp_tools/
github_mcp_server.sh with args: {'args': 'repo list --sort=stars --limit=5'}
Tool gh completed successfully
2025-08-03 17:11:59 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
Invoking tool gh on server /Users/gigi/git/ai-six/py/backend/mcp_tools/
github_mcp_server.sh with args: {'args': 'repo list --json "name,stargazersCount"
--limit 100'}
Tool gh completed successfully
2025-08-03 17:12:00 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
Invoking tool gh on server /Users/gigi/git/ai-six/py/backend/mcp_tools/
github_mcp_server.sh with args: {'args': 'repo list --json name,stargazerCount --
limit 100'}
Tool gh completed successfully
2025-08-03 17:12:04 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
2025-08-03 17:14:02 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
2025-08-03 17:14:18 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
Invoking tool gh on server /Users/gigi/git/ai-six/py/backend/mcp_tools/
```

```
github_mcp_server.sh with args: {'args': 'repo list --json name,stargazerCount --
limit 100 --page 2'}
Tool gh completed successfully
2025-08-03 17:14:25 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
2025-08-03 17:15:12 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
Invoking tool gh on server /Users/gigi/git/ai-six/py/backend/mcp_tools/
github_mcp_server.sh with args: {'args': 'repo list --json name,stargazerCount --
limit 300'}
Tool gh completed successfully
2025-08-03 17:15:18 - HTTP Request: POST https://api.openai.com/v1/chat/
completions "HTTP/1.1 200 OK"
```

Penggunaan alat yang tidak tepat dan tidak efisien ini merupakan salah satu masalah utama LLM, dan kita akan membahas masalah ini di *bab 10*.

7.8 RINGKASAN

Mari kita rangkum. Dalam bab ini, kita telah menjelajahi **Protokol Konteks Model** (MCP) – landasan untuk standarisasi integrasi alat dan konteks dalam sistem AI agenik tingkat lanjut. Kita telah meneliti ekosistem dan motivasi di balik MCP, meliputi arsitektur, manfaat, dan lapisan protokol serta implementasi praktisnya. Kita telah menunjukkan bagaimana konsep server dan klien MCP dipetakan dengan rapi ke abstraksi alat kerangka kerja AI-6, dan bagaimana integrasi MCP menghadirkan interoperabilitas yang luas dan tanpa hambatan: alat yang kompatibel dapat dihubungkan, digunakan kembali, atau diganti, membebaskan Anda dari ketergantungan vendor dan biaya integrasi khusus.

Kita telah membandingkan implementasi alat baris perintah sebagai alat AI-6 khusus dan sebagai server MCP dan melihat bahwa MCP membuatnya sama mudahnya untuk membangun dan berbagi alat di berbagai bahasa dan platform. Perhatikan bahwa pengujian server MCP mungkin memerlukan upaya tambahan karena alat berjalan dalam proses terpisah. Anda telah mempelajari bagaimana server MCP lokal dan jarak jauh dapat ditemukan dan digunakan, dengan pengelola alat AI-6 yang mengabstraksikan seluruh proses penemuan dan pendaftaran. Pendekatan ini memungkinkan AI-6 (dan kerangka kerja AI berbasis agen serupa lainnya) untuk secara mulus menggabungkan alat internal dan alat MCP eksternalnya menjadi seperangkat alat dinamis yang terpadu dan dapat diakses oleh agen mana pun, tanpa memerlukan logika tambahan di mesin AI-6.

Terakhir, kami melihat integrasi praktis di dunia nyata: bagaimana sistem AI berbasis agen, seperti AI-6, sekarang mendelegasikan semua penanganan alat (penemuan, pemanggilan, dan manajemen) ke manajer alat yang tangguh, membuat agennya jauh lebih mudah diperluas. Seiring pertumbuhan ekosistem MCP, AI berbasis agen Anda dapat langsung memanfaatkan alat, layanan, atau sumber daya baru apa pun yang dipublikasikan ke protokol terbuka, memungkinkan eksperimen dan penskalaan yang cepat dalam sistem multi-agen yang kompleks. Integrasi sesederhana konfigurasi, bukan penulisan ulang kode.

Pada bab berikutnya, kita akan membawanya ke tingkat selanjutnya dan menyelami lebih dalam desain sistem AI multi-agen.

BAB 8
MERANCANG SISTEM MULTI-AGEN

8.1 PENDAHULUAN

Meskipun sistem agen tunggal dapat menyelesaikan tugas-tugas yang mengesankan, banyak masalah di dunia nyata membutuhkan koordinasi antara beberapa agen khusus yang bekerja bersama. Bab ini berfokus pada perancangan sistem AI multi-agen yang dapat berkolaborasi untuk memecahkan masalah kompleks di luar kemampuan agen tunggal mana pun.

Sistem multi-agen mewakili batas berikutnya dalam arsitektur AI agenik. Alih-alih mengandalkan satu agen untuk menangani semua aspek masalah, kita dapat membuat tim agen khusus, masing-masing dengan keahlian, alat, dan tanggung jawabnya sendiri. Kita akan mengeksplorasi pola orkestrasi multi-agen, strategi dekomposisi agen, desain perintah sistem untuk peran agen yang berbeda, kerangka kerja penugasan alat, rekayasa konteks untuk lingkungan multi-agen, dan mekanisme resolusi konflik. Kita akan meningkatkan AI-6 dengan dukungan agen eksplisit dan mengintegrasikan protokol **Agen-ke-Agen** (A2A) untuk mendemonstrasikan konsep-konsep ini secara praktis. Topik-topik utama berikut akan dibahas:

- Apa itu orkestrasi multi-agen dan mengapa itu penting?
- Pengantar protokol A2A
- Merancang alur kerja agen kolaboratif
- Penyelesaian konflik dan penugasan peran dalam sistem multi-agen

8.2 APA ITU ORKESTRASI MULTI-AGEN DAN MENGAPA ITU PENTING?

Sejauh ini dalam buku ini, kita telah banyak membahas tentang loop pemanggilan alat agen dan menekankan bahwa semua pemikiran dilakukan oleh LLM. Mesin AI-6 sangat mampu mendukung alur kerja agen semacam itu, tetapi apa itu agen? Di bagian ini, kita akan mendefinisikan secara tepat apa itu agen AI dalam istilah yang familiar, memperluas AI-6 untuk mendukung agen eksplisit, membahas mengapa sistem multi-agen penting, dan akhirnya, mengeksplorasi berbagai cara untuk mengorkestrasi beberapa agen secara efektif.

Apa Itu Agen AI?

Agen AI adalah komponen dalam sistem AI yang memiliki beberapa tujuan dan pengetahuan yang dikodekan sebagai perintah sistem. Ia memiliki akses ke serangkaian alat yang dapat digunakannya untuk mencapai tujuannya. Terakhir, ia memiliki klien LLM yang dapat digunakannya untuk bernalar tentang dunia, tujuannya, dan cara menggunakan alat-alatnya. Agen AI berinteraksi dengan dunia luar dengan menerima pesan pengguna (di mana pengguna dapat berupa sistem AI host atau agen lain) dan mengembalikan respons. Agen mempertahankan status internalnya sendiri (misalnya, riwayat percakapan, riwayat penggunaan alat, dan lain-lain.) dan hanya mengembalikan respons akhir kepada pengguna.

Fakta bahwa setiap agen adalah komponen mandiri yang menyembunyikan status internalnya berarti kita dapat membangun sistem multi-agen tanpa memenuhi jendela

konteks dari satu LLM. Sub-agen dapat menjalankan loop ageniknya sendiri dan menghabiskan banyak token tanpa mengacaukan konteks induknya. Jika Anda berpikir agen AI terdengar sangat mirip dengan mesin AI-6, Anda tidak salah. Mari kita lihat bagaimana memperluas AI-6 untuk mendukung agen secara lebih eksplisit.

Menambahkan Agen Eksplisit Ke AI-6

Mesin AI-6 dapat digunakan apa adanya untuk membangun sistem multi-agen dengan cara berikut:

1. Definisikan alat agen AI-6.
2. Buat instance mesin AI-6 induk dan berikan agen AI-6 lainnya sebagai alat.
3. Mesin AI-6 induk dapat memanggil agen AI-6 lainnya sebagai alat untuk mendelegasikan pekerjaan.

Ini bagus, tetapi ada beberapa celah kecil yang perlu diatasi:

- Saat ini, mesin AI-6 tidak memiliki prompt sistem. Ini berarti bahwa setiap kali memanggil alat agen AI-6, pemanggil harus memberikan prompt sistem selain pesan pengguna yang sebenarnya.
- Mendefinisikan dan mengkonfigurasi berbagai agen AI-6 bukanlah hal yang sepele.
- AI-6 memiliki konfigurasi global untuk alat, tetapi agen harus memiliki subset alat mereka sendiri.

Mari kita atasi semua masalah ini dengan melakukan perubahan berikut pada AI-6:

1. Ganti nama Engine menjadi Agent (ini hanya terminologi dan bukan perubahan fungsional).
2. Tambahkan prompt sistem ke konfigurasi agen.
3. Tambahkan bidang enabled_tools dan disabled_tools ke konfigurasi agen.
4. Tambahkan AgentTool baru yang membungkus instance Agen AI-6.

Selain penggantian nama Engine menjadi Agent, yang sebagian besar menetapkan terminologi, perubahan lainnya hanyalah tambahan. Kami tidak menghapus atau memodifikasi kemampuan dan alur kerja yang ada.

Mengganti Nama Engine Menjadi Agent

Seluruh paket backend.engine sekarang bernama `backend.agent`. File `engine.py` sekarang menjadi agent.py. Kelas Engine sekarang menjadi Agent. Semua referensi ke Engine dalam kode dan dokumentasi telah diubah menjadi Agent. Tidak ada yang istimewa di sini, hanya pencarian dan penggantian.

Menambahkan Prompt Sistem Ke Konfigurasi Agent

Perubahan ini juga cukup sederhana. Berikut adalah field baru yang ditambahkan ke kelas Config di `backend/agent/config.py`:

```
system_prompt: Optional[str] = None
```

Perlu dicatat bahwa ini bersifat opsional, yang berarti kompatibel dengan versi sebelumnya dan dapat berfungsi seperti sebelumnya serta berinteraksi dengan LLM tanpa tujuan atau pengetahuan khusus.

Menambahkan field enabled_tools dan disabled_tools Ke Konfigurasi Agent

Ini juga merupakan perubahan sederhana. Berikut adalah field baru yang ditambahkan ke kelas Config di `backend/agent/config.py`:

```
enabled_tools: Optional[List[str]] = None
disabled_tools: Optional[List[str]] = None
```

Semantiknya adalah jika `enabled_tools` disediakan, hanya alat-alat tersebut yang akan tersedia untuk agen. Jika `disabled_tools` disediakan, semua alat kecuali alat-alat tersebut akan tersedia untuk agen. Jika tidak ada yang disediakan, semua alat tersedia. Menyediakan kedua bidang tersebut merupakan kesalahan. Ini sekali lagi kompatibel dengan versi sebelumnya dan memberikan kendali penuh atas kumpulan alat yang dapat digunakan oleh agen mana pun.

Menambahkan Agenttool Baru Yang Membungkus Instance Agen AI-6

Ini adalah perubahan yang paling menarik. Kami menambahkan tipe alat baru yang membungkus instance Agen. Namun, instance Agen itu sendiri tidak memiliki kode baru untuk meluncurkan sub-agen. LLM menerima daftar sub-agen yang tersedia sebagai bagian dari alat yang tersedia, dan kemudian LLM dapat memutuskan untuk meluncurkan salah satu sub-agen dengan panggilan alat. Setiap sub-agen tersebut dapat memiliki alat Agen dalam kumpulan alatnya sendiri, yang berarti ketika LLM berinteraksi dengan sub-agen, ia dapat memutuskan untuk meluncurkan sub-agen lain, dan seterusnya. Ini berarti kita dapat memiliki pohon agen yang berjalan dan berkolaborasi untuk menyelesaikan suatu masalah.

Kelas AgentTool didefinisikan di `backend/object_model/agent_tool.py`. Mari kita uraikan. Pernyataan impor mencerminkan entitas yang diandalkan dan berinteraksi dengan alat ini:

```
from typing import Callable, Optional
from backend.object_model import Tool, Parameter
from backend.agent.agent import Agent
from backend.agent.config import Config
```

Sekarang, kelas AgentTool mewarisi dari kelas dasar Tool. Konstruktornya menerima objek konfigurasi agen yang berisi semua informasi yang dibutuhkan untuk membuat instance Agent. Pertama, ia membuat instance Agent dan menginisialisasi callback `_on_tool_call_func` ke None. Agen tersebut sudah dikonfigurasi dengan prompt sistemnya dan kumpulan alat yang diaktifkan/dinonaktifkan:

```
class AgentTool(Tool):
    """Tool that wraps an Agent and allows sending messages to it."""

    def __init__(self, agent_config: Config):
        """Initialize the AgentTool with a Config."""
        self.agent = Agent(agent_config)
        self._on_tool_call_func: Optional[
            Callable[[str, dict, str], None]
        ] = None
```

```
class AgentTool(Tool):
    """Tool that wraps an Agent and allows sending messages to it."""

    def __init__(self, agent_config: Config):
        """Initialize the AgentTool with a Config."""
        self.agent = Agent(agent_config)
        self._on_tool_call_func: Optional[
            Callable[[str, dict, str], None]] = None
```

Kemudian, ia membuat parameter untuk panggilan alat. Dalam hal ini, alat tersebut mengambil satu parameter bernama pesan, yaitu pesan yang akan dikirim ke agen. Selanjutnya, ia memanggil konstruktor superkelas (kelas Tool) dan meneruskan nama, deskripsi, parameter, dan bidang yang diperlukan (pesan wajib diisi):

```
# Create tool definition
parameters = [
    Parameter(
        name='message',
        type='string',
        description='The message to send to the agent'
    )
]

super().__init__(
    name=f"agent_{agent_config.name}",
    description=f"Send a message to the {agent_config.name} agent.
{agent_config.description}",
    parameters=parameters,
    required={'message'}
)
```

Metode `set_tool_call_callback()` dipanggil oleh agen selama inisialisasi untuk mengatur fungsi callback yang akan dipanggil setiap kali sub-agen melakukan panggilan alat. Fungsi callback opsional ini berasal dari sistem AI host. Hal ini memungkinkan sistem AI host untuk mencegat dan menangani panggilan alat yang dilakukan oleh agen mana pun:

```
    def set_tool_call_callback(
        self, callback: Optional[Callable[[str, dict, str], None]]
    ):
"""Set the callback function for tool calls made by this agent."""
    self._on_tool_call_func = callback
```

Terakhir, metode run() mengirimkan pesan ke agen, yang mungkin memulai siklus ageniknya sendiri dan akhirnya mengembalikan respons:

```
    def run(self, message: str, **kwargs) -> str:
"""
Send a message to the agent and return the response.

Args:
    message: The message to send to the agent
```

```
        **kwargs: Additional arguments (ignored)

    Returns:
        The agent's response
    """
        return self.agent.send_message(
            message,
            self.agent.default_model_id,
            on_tool_call_func=self._on_tool_call_func)
```

Pada akhirnya, dari perspektif siklus agen AI-6, sub-agen adalah alat yang dapat dipanggil untuk mendelegasikan pekerjaan. LLM memutuskan kapan dan bagaimana memanggil sub-agen berdasarkan deskripsinya, sama seperti memutuskan untuk memanggil alat lain. Fakta bahwa AgentTool membungkus instance Agen AI-6 tidak relevan bagi LLM.

Pada titik ini, kita memahami bagaimana AI-6 memungkinkan sistem multi-agen. Sekarang adalah waktu yang tepat untuk membahas mengapa sistem multi-agen sangat penting.

Mengapa Kita Membutuhkan Sistem Multi-Agen?

Secara teori, satu agen yang cukup mumpuni dapat menangani tugas apa pun. Namun, seiring meningkatnya kompleksitas tugas, pendekatan ini dengan cepat menemui beberapa keterbatasan.

Keterbatasan Jendela Konteks

LLM memiliki jendela konteks yang terbatas dan hanya dapat mempertimbangkan sejumlah informasi tertentu sekaligus. Untuk tugas-tugas kompleks yang membutuhkan pengetahuan latar belakang yang luas, beberapa langkah, atau kumpulan data yang besar, satu agen mungkin tidak dapat memasukkan semua informasi yang diperlukan ke dalam jendela konteksnya. Performa model menurun ketika konteks menjadi terlalu besar, terutama ketika dipenuhi dengan detail langkah yang sebagian besar tidak relevan dengan langkah-langkah selanjutnya.

Tantangan Spesialisasi

Tugas yang berbeda membutuhkan keahlian dan keterampilan yang berbeda. Satu agen mungkin tidak menguasai semua keterampilan yang diperlukan, dan agen super dengan perintah sistem yang besar membingungkan model saat ini. Sama seperti organisasi manusia yang bergantung pada spesialis, sistem AI mendapat manfaat dari agen khusus yang berfokus secara mendalam pada bidang keahlian mereka (Lihat moltbook untuk memahami bagaimana agen AI dapat meniru pola interaksi manusia dan mengatur diri mereka sendiri sesuai dengan itu). Tugas yang kompleks mungkin membutuhkan ratusan alat, yang akan membebani agen tunggal, dan menyebabkan pemilihan alat yang buruk atau ilusi kemampuan.

Kendala Pemilihan Model

Tugas yang berbeda mendapat manfaat dari model yang berbeda. Agen tunggal yang terikat pada satu model tidak optimal untuk semua aspek tugas. Beberapa tugas membutuhkan penalaran tingkat lanjut dari model besar seperti GPT-5, sementara yang lain dapat ditangani oleh model yang lebih kecil dan lebih cepat. Sistem monolitik memaksa Anda

untuk membayar lebih untuk tugas-tugas sederhana atau berkinerja buruk pada tugas-tugas kompleks.

Sistem multi-agen mengatasi keterbatasan ini dengan menciptakan agen khusus yang berfokus pada aspek tugas tertentu. Setiap agen memiliki perintah sistem, seperangkat alat, dan modelnya sendiri, beroperasi dalam jendela konteksnya sendiri sambil memanfaatkan keahlian dan alat untuk berkontribusi pada solusi keseluruhan.

Distribusi Beban Kognitif

Sistem multi-agen mendistribusikan beban kognitif di antara agen-agen khusus, meniru tim manusia di mana masalah kompleks dipecah dan ditugaskan kepada para ahli di bidangnya. Penelitian menunjukkan bahwa pemrosesan multi-agen secara berurutan mencapai tingkat akurasi yang secara signifikan lebih tinggi dibandingkan dengan pendekatan agen tunggal, karena agen-agen yang terspesialisasi dapat mempertahankan fokus perhatian pada bidang keahliannya masing-masing.

Toleransi Kesalahan Dan Ketahanan

Sistem multi-agen menawarkan keunggulan toleransi kesalahan yang melekat dengan mendistribusikan tanggung jawab di antara beberapa agen. Sistem ini terus beroperasi ketika komponen individual gagal melalui strategi pemulihan seperti mencoba kembali dengan parameter yang berbeda atau mengarahkan ke agen alternatif. Agen-agen kritis dapat dikerahkan dengan redundansi, memungkinkan pengalihan otomatis tanpa hambatan ketika masalah muncul.

Kemampuan Pengujian Dan Penelusuran Kesalahan

Sifat modular memungkinkan pengujian yang ditargetkan, pengamatan yang lebih baik, dan debugging yang lebih baik. Serta penelusuran kesalahan yang lebih efektif. Setiap agen dapat diuji secara terpisah, mirip dengan pengujian unit, sementara komunikasi antar agen memberikan pengamatan yang luar biasa. Profiling kinerja granular memungkinkan pemantauan independen terhadap konsumsi sumber daya, waktu eksekusi, dan penggunaan token setiap agen.

Cara Mengorkestrasi Banyak Agen

Sekarang kita memahami apa itu agen AI dan mengapa sistem multi-agen bermanfaat, mari kita jelajahi berbagai strategi untuk mengorkestrasi banyak agen agar bekerja sama secara efektif. Orkestrasi agen mengacu pada koordinasi, komunikasi, dan manajemen banyak agen untuk mencapai tujuan bersama. Pendekatan orkestrasi yang Anda pilih sangat memengaruhi kinerja, keandalan, dan kompleksitas sistem.

Orkestrasi Sekuensial

Orkestrasi sekuensial adalah pola yang paling sederhana dan intuitif, di mana agen bekerja dalam urutan yang telah ditentukan, dengan setiap agen meneruskan outputnya ke agen berikutnya dalam rantai. Alur kerja linier ini menyerupai pipeline tradisional di mana setiap tahap memproses output dari tahap sebelumnya. Alurnya telah ditentukan dan dapat diprediksi.

Tetapi setiap agen masih dapat memiliki loop agen internalnya sendiri dan memanggil alat sesuai kebutuhan. Orkestrasi sekuensial ideal untuk tugas-tugas yang memiliki perkembangan linier alami, seperti alur kerja pembuatan konten (riset, kerangka, draf,

tinjauan, dan publikasi), alur pemrosesan data, atau proses apa pun di mana setiap langkah secara logis bergantung pada penyelesaian langkah sebelumnya.

Agen orkestrator memelihara daftar agen secara berurutan dan secara iteratif memanggil setiap agen dengan hasil yang terakumulasi dari agen sebelumya. Keluaran setiap agen menjadi bagian dari masukan untuk agen berikutnya (lihat Gambar 8.1):

Gambar 8.1 Orkestrasi Sekuensial

Berikut adalah keuntungan yang terkait dengan orkestrasi sekuensial:

- Mudah dipahami dan diimplementasikan
- Urutan eksekusi yang dapat diprediksi
- Mudah untuk melakukan debug dan melacak masalah
- Sangat cocok untuk banyak alur kerja dunia nyata

Berikut adalah kerugiannya:

- Hambatan sekuensial; yaitu, jika satu agen gagal, seluruh proses berhenti
- Tidak ada peluang paralelisasi
- Dapat lambat untuk rantai yang panjang
- Struktur kaku yang tidak beradaptasi dengan perubahan persyaratan

Meskipun orkestrasi sekuensial menawarkan kesederhanaan dan prediktabilitas, sifat liniernya dapat menjadi hambatan ketika kecepatan dan toleransi kesalahan menjadi prioritas. Di sinilah orkestrasi paralel bersinar, memungkinkan banyak agen untuk bekerja secara bersamaan dan memanfaatkan kekuatan pemrosesan konkuren.

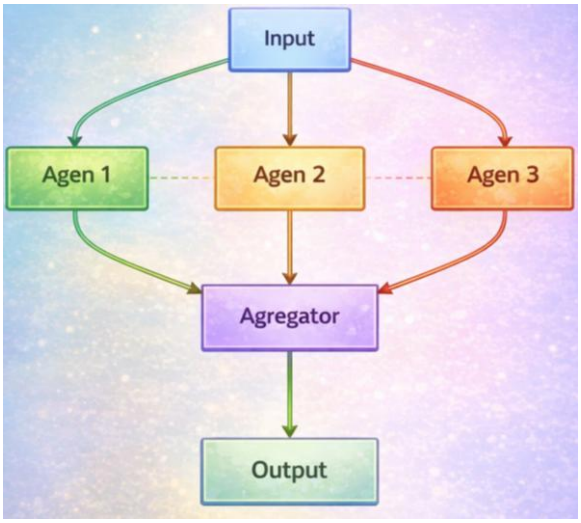
Orkestrasi Paralel

Orkestrasi paralel melibatkan menjalankan banyak agen secara bersamaan pada input yang sama atau aspek yang berbeda dari suatu masalah, kemudian menggabungkan hasilnya. Pola ini memaksimalkan throughput dan dapat memberikan beragam perspektif pada masalah yang sama. Perhatikan dua gaya orkestrasi paralel:

- Setiap agen bekerja secara independen pada input yang sama dan menghasilkan outputnya sendiri. Orkestrator kemudian menggabungkan atau mensintesis output ini menjadi hasil akhir, mungkin dengan juri LLM untuk memberi skor dan memilih pemenang.
- Orkestrator menguraikan input menjadi sub-tugas independen yang dapat dijalankan secara paralel dan menetapkan setiap sub-tugas ke agen yang berbeda. Agen-agen tersebut bekerja secara paralel pada sub-tugas yang ditugaskan kepada mereka, dan orkestrator menggabungkan output mereka.

Orkestrasi paralel efektif untuk tugas-tugas di mana diperlukan beberapa analisis independen, seperti analisis sentimen dari berbagai perspektif, pengumpulan data multi-sumber, riset kompetitif, atau ketika Anda ingin memvalidasi hasil melalui konsensus.

Pengatur (orkestrator) mengirimkan tugas yang sama atau terkait ke beberapa agen secara bersamaan, kemudian mengumpulkan dan mensintesis respons mereka. Hal ini seringkali membutuhkan logika agregasi hasil yang canggih (lihat Gambar 8.2):



Gambar 8.2 Orkestrasi Paralel

Berikut adalah keuntungannya:

- Memaksimalkan throughput dan kecepatan
- Memberikan banyak perspektif
- Toleransi kesalahan alami (jika satu agen gagal, yang lain tetap melanjutkan)
- Dapat menerapkan mekanisme konsensus untuk keandalan yang lebih tinggi

Berikut adalah kerugiannya:

- Membutuhkan logika sintesis hasil
- Potensi perebutan sumber daya
- Dapat menghasilkan hasil yang saling bertentangan yang perlu diselesaikan
- Penanganan kesalahan yang lebih kompleks

Orkestrasi paralel unggul dalam memaksimalkan throughput, tetapi mengelola banyak agen konkuren dan mensintesis hasilnya dapat menjadi sulit dikelola seiring bertambahnya kompleksitas. Untuk tugas-tugas yang secara alami terpecah menjadi subtugas yang dapat dikelola dengan jalur delegasi yang jelas, orkestrasi hierarkis memberikan struktur dan kejelasan yang mungkin kurang dimiliki oleh pendekatan paralel.

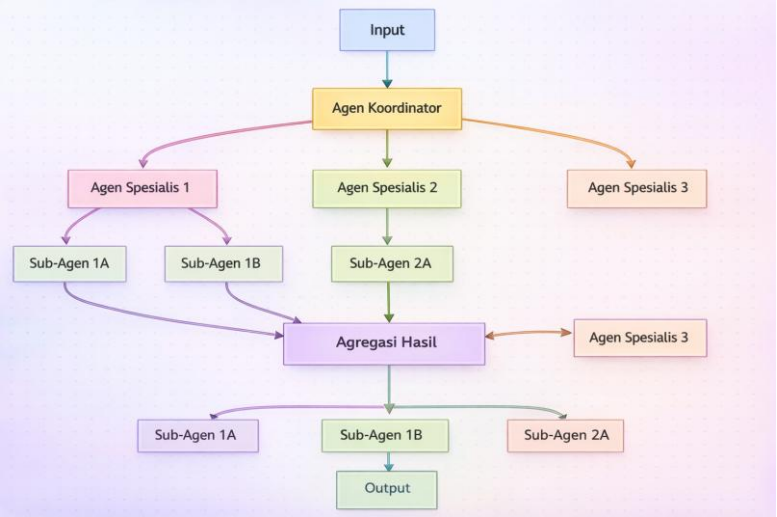
Orkestrasi Hierarkis

Orkestrasi hierarkis mengatur agen dalam struktur seperti pohon di mana agen tingkat atas mengoordinasikan dan mengelola agen tingkat bawah. Pola ini meniru struktur organisasi di mana manajer mendelegasikan tugas kepada spesialis.

Pendekatan ini bekerja dengan baik untuk tugas-tugas kompleks yang secara alami terurai menjadi subtugas, seperti manajemen proyek, koordinasi penelitian, atau skenario apa pun di mana Anda membutuhkan berbagai tingkat abstraksi dan delegasi.

Agen koordinator di tingkat atas memecah tugas-tugas kompleks dan mendelegasikannya kepada agen spesialis. Agen spesialis ini mungkin selanjutnya

mendelegasikan kepada sub-spesialis, menciptakan hierarki tanggung jawab (lihat Gambar 8.3):



Gambar 8.3 Orkestrasi Hikrarkis

Berikut adalah keuntungannya:

- Dekomposisi tugas alami
- Batasan tanggung jawab yang jelas
- Arsitektur yang dapat diskalakan
- Sesuai dengan pola organisasi manusia

Berikut adalah kerugiannya:

- Overhead komunikasi antar level
- Potensi titik kegagalan tunggal di level yang lebih tinggi
- Kompleksitas dalam logika dekomposisi tugas
- Dapat menimbulkan penundaan karena overhead koordinasi

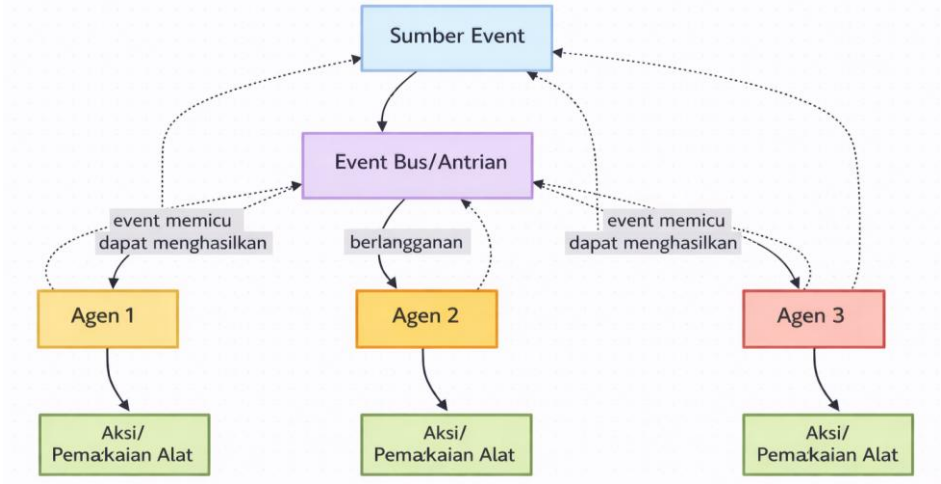
Orkestrasi hierarkis bekerja dengan baik untuk masalah dengan dekomposisi yang jelas, tetapi bergantung pada pembagian tugas yang telah ditentukan sebelumnya dan dapat kesulitan dengan lingkungan yang dinamis dan berubah dengan cepat. Ketika sistem Anda perlu bereaksi terhadap peristiwa yang tidak terduga dan perubahan kondisi secara real-time, orkestrasi berbasis peristiwa menawarkan responsivitas yang tidak dapat ditandingi oleh pendekatan hierarkis.

Orkestrasi Berbasis Peristiwa

Orkestrasi berbasis peristiwa memungkinkan agen untuk bereaksi terhadap peristiwa atau perubahan status dalam sistem, menciptakan alur kerja yang lebih dinamis dan responsif. Agen berlangganan peristiwa dan memicu tindakan berdasarkan kondisi atau pemicu tertentu.

Pola ini ideal untuk sistem reaktif, skenario pemantauan, pemrosesan waktu nyata, atau situasi apa pun di mana agen perlu menanggapi kondisi yang berubah daripada mengikuti alur kerja yang telah ditentukan sebelumnya.

Agen mendaftarkan minat mereka pada peristiwa atau kondisi tertentu. Ketika peristiwa ini terjadi, agen yang relevan akan diberi tahu dan dapat mengambil tindakan yang sesuai. Hal ini memerlukan sistem peristiwa atau infrastruktur antrian pesan:



Gambar 8.4 Orkestrasi Berbasis Peristiwa

Berikut adalah keuntungannya:

- Sangat responsif terhadap perubahan kondisi
- Interaksi agen yang terpisah
- Dukungan alami untuk pemrosesan asinkron
- Skalabilitas yang baik dengan kompleksitas sistem

Berikut adalah kerugiannya:

- Debugging yang kompleks karena sifat asinkronnya
- Membutuhkan infrastruktur peristiwa yang kuat
- Potensi kondisi persaingan (race condition)
- Lebih sulit untuk memprediksi alur eksekusi

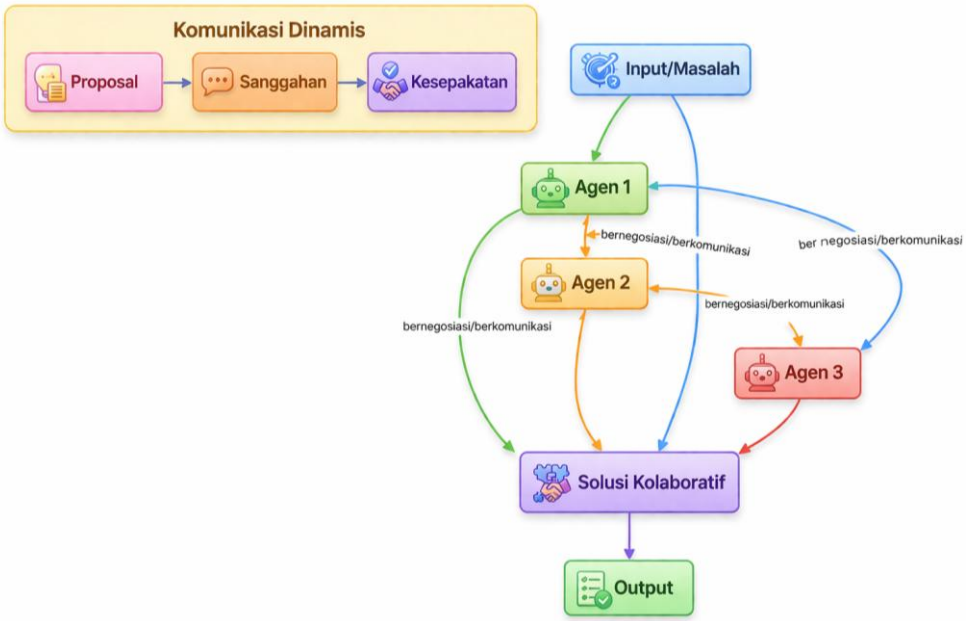
Orkestrasi berbasis peristiwa memberikan respons yang sangat baik terhadap perubahan kondisi, tetapi sifat reaktifnya berarti agen masih bekerja dalam peran dan respons yang telah ditentukan sebelumnya. Untuk masalah yang benar-benar kompleks dan terbuka di mana pendekatan solusi itu sendiri harus muncul dari interaksi agen, orkestrasi kolaboratif memungkinkan agen untuk melampaui keterbatasan individu mereka melalui negosiasi dinamis dan pemecahan masalah kolektif.

Orkestrasi Kolaboratif

Orkestrasi kolaboratif memungkinkan agen untuk secara dinamis bernegosiasi, berkomunikasi, dan berkoordinasi satu sama lain untuk memecahkan masalah secara kolektif. Pola ini memungkinkan perilaku yang muncul dan pendekatan pemecahan masalah yang adaptif.

Gunakan orkestrasi kolaboratif untuk masalah kompleks dan terbuka di mana pendekatan solusi tidak ditentukan sebelumnya, seperti proyek kreatif, tugas penelitian yang kompleks, atau skenario yang membutuhkan negosiasi dan pembangunan konsensus.

Agen dapat memulai komunikasi satu sama lain, mengusulkan solusi, menegosiasikan pendekatan, dan secara dinamis membentuk kelompok kerja. Ini membutuhkan protokol komunikasi dan mekanisme koordinasi yang canggih.



Gambar 8.5 Orkestrasi Kolaboratif

Berikut adalah keuntungannya:

- Sangat adaptif dan fleksibel
- Dapat menangani situasi yang tidak terduga
- Memanfaatkan kecerdasan kolektif
- Memungkinkan solusi yang muncul secara spontan

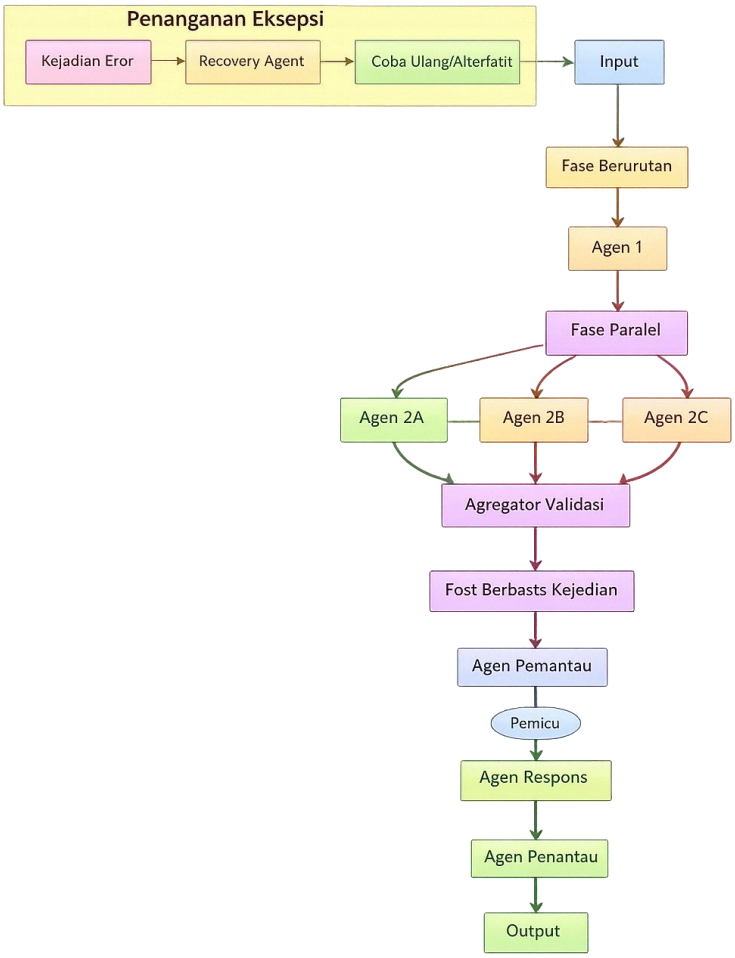
Berikut adalah kerugiannya:

- Kompleks untuk diimplementasikan dan di-debug
- Potensi terjadinya loop tak terbatas atau kebuntuan
- Sulit untuk memprediksi hasilnya
- Membutuhkan protokol koordinasi yang canggih

Meskipun orkestrasi kolaboratif menawarkan fleksibilitas dan adaptabilitas terbaik, kompleksitasnya dapat sangat membingungkan bagi banyak aplikasi praktis. Kenyataannya adalah bahwa sebagian besar sistem multi-agen yang sukses tidak bergantung pada satu pola orkestrasi tunggal, tetapi menggabungkan kekuatan dari berbagai pendekatan untuk menciptakan solusi yang kuat dan efisien yang disesuaikan dengan kebutuhan spesifik mereka.

Orkestrasi Hibrida

Sistem dunia nyata seringkali mendapat manfaat dari penggabungan beberapa pola orkestrasi. Misalnya, Anda dapat menggunakan orkestrasi sekuensial untuk alur kerja utama sambil menggunakan orkestrasi paralel untuk langkah-langkah validasi tertentu, atau menerapkan koordinasi hierarkis dengan respons berbasis peristiwa untuk menangani pengecualian:



Gambar 8.6 Orkestrasi Hibrida

Kita telah membahas berbagai strategi orkestrasi. Mari kita lihat bagaimana cara menentukan strategi mana yang terbaik untuk situasi tertentu.

Pertimbangan Desain

Saat memilih strategi orkestrasi, pertimbangkan faktor-faktor kunci berikut:

- **Kompleksitas tugas:** Tugas sederhana dan linier lebih cocok untuk orkestrasi sekuensial, sedangkan masalah kompleks dan multifaset mungkin memerlukan pendekatan hierarkis atau kolaboratif
- **Persyaratan kinerja:** Jika kecepatan sangat penting, orkestrasi paralel dapat membantu, tetapi jika konsistensi lebih penting, pemrosesan sekuensial mungkin lebih baik
- **Batasan sumber daya:** Pertimbangkan overhead komputasi dan memori dari setiap pendekatan, terutama untuk pola paralel dan hierarkis
- **Kebutuhan toleransi kesalahan:** Pola berbasis peristiwa dan paralel umumnya menawarkan toleransi kesalahan yang lebih baik daripada pendekatan sekuensial
- **Debugging dan pemeliharaan:** Orkestrasi sekuensial paling mudah untuk di-debug, sedangkan orkestrasi kolaboratif adalah yang paling menantang

Kunci keberhasilan orkestrasi multi-agen adalah mencocokkan pola orkestrasi dengan persyaratan kasus penggunaan spesifik Anda dan bersiap untuk beradaptasi seiring perkembangan persyaratan tersebut. Tentu saja, Anda harus mempertimbangkan kekurangan dari setiap pendekatan dan bagaimana kekurangan tersebut terakumulasi, serta meminimalkannya sebisa mungkin. Misalnya, dalam orkestrasi sekuensial, Anda dapat menambahkan mekanisme percobaan ulang dan batas waktu untuk menghindari hambatan.

Dalam orkestrasi paralel, Anda dapat menambahkan agen penilai untuk menyelesaikan konflik. Dalam orkestrasi hierarkis, Anda dapat menambahkan redundansi pada agen-agen kritis. Dalam orkestrasi berbasis peristiwa, Anda dapat menambahkan pencatatan dan pelacakan untuk meningkatkan kemampuan pengamatan. Dalam orkestrasi kolaboratif, Anda dapat menambahkan batasan jumlah interaksi antar agen untuk menghindari perulangan tak terbatas.

Sekarang kita memiliki pemahaman yang baik tentang pola orkestrasi multi-agen, mari kita jelajahi bagaimana agen dapat berkomunikasi satu sama lain secara efektif menggunakan protokol A2A.

8.3 PENGANTAR PROTOKOL A2A

Meskipun pola orkestrasi mendefinisikan strategi koordinasi tingkat tinggi antar agen, sistem multi-agen yang efektif membutuhkan mekanisme komunikasi yang kuat yang memungkinkan agen untuk bertukar informasi, mengoordinasikan aktivitas, dan berkolaborasi tanpa hambatan. Protokol A2A menyediakan kerangka kerja standar bagi agen AI untuk berkomunikasi dan berkolaborasi di berbagai sistem, menawarkan kemampuan yang melampaui apa yang dirancang oleh MCP.

Protokol A2A dibangun di sekitar beberapa aktor dan konsep inti yang memungkinkan interaksi agen yang canggih. Pada dasarnya terdapat tiga aktor utama: **pengguna** yang memulai permintaan (manusia atau agen AI), **klien A2A** (agen klien) yang berkomunikasi atas nama pengguna, dan **server A2A** (agen jarak jauh) yang memproses pesan dan tugas serta mengembalikan hasilnya. Hal ini menciptakan model komunikasi yang terstruktur namun fleksibel yang mendukung interaksi sederhana dan kolaborasi kompleks multi-giliran.

Komponen Inti Protokol A2A

Protokol A2A dibangun di sekitar beberapa elemen fundamental yang bekerja sama untuk memungkinkan interaksi agen yang canggih:

- **Kartu Agen**: Metadata JSON yang menjelaskan kemampuan agen, memungkinkan penemuan otomatis dan pencocokan kemampuan
- **Tugas**: Unit kerja yang memiliki status dengan pengidentifikasi unik yang memungkinkan pelacakan operasi yang berjalan lama dan interaksi multi-giliran
- **Pesan**: Giliran komunikasi individual antara klien dan agen, mendukung pertukaran konten yang kaya
- **Bagian**: Wadah konten yang dapat menampung berbagai jenis data (teks, file, atau data), memungkinkan komunikasi yang independen dari modalitas

- **Artefak**: Keluaran nyata yang dihasilkan selama eksekusi tugas, seperti file PDF atau gambar, memberikan hasil konkret dari interaksi agen

Berikut adalah kartu agen dari server A2A k8s-ai yang saya jalankan secara lokal:

```
% http -b GET http://localhost:9999/.well-known/agent-card.json
{
    "capabilities": {
      "streaming": true
    },
    "defaultInputModes": [
      "text/plain"
    ],
    "defaultOutputModes": [
      "text/plain"
    ],
    "description": "Kubernetes AI assistant with kubectl access for context: kind-
k8s-ai",
    "name": "k8s-ai Agent",
    "preferredTransport": "JSONRPC",
    "protocolVersion": "0.3.0",
    "security": [
       {
            "BearerAuth": []
       }
    ],
    "securitySchemes": {
        "BearerAuth": {
           "scheme": "bearer",
           "type": "http"
        }
    },
    "skills": [
        {
           "description": "Execute kubectl commands and provide Kubernetes
cluster insights, troubleshooting, and management",
            "examples": [
              "show me all pods",
              "what is the status of the cluster?",
              "describe the nginx deployment",
              "get service endpoints",
              "check node health",
              "troubleshoot pending pods"
           ],
           "id": "kubectl_operations",
           "name": "Kubernetes Operations",
           "tags": [
               "kubernetes",
               "kubectl",
               "cluster",
```

```
                "pods",
                "deployments",
                "services"
            ]
        }
    ],
    "supportsAuthenticatedExtendedCard": true,
    "url": "http://127.0.0.1:9999/",
    "version": "1.0.0"
}
```

Setelah memahami komponen inti dari protokol A2A, mari kita bahas mekanisme komunikasinya.

Mekanisme Komunikasi A2A

Protokol A2A mendukung berbagai pola interaksi untuk mengakomodasi berbagai kasus penggunaan:

- **Permintaan/respons (polling)**: Komunikasi sinkron tradisional untuk interaksi sederhana yang melibatkan pesan
- **Streaming dengan SSE**: Komunikasi waktu nyata untuk tugas yang sedang berlangsung dan pembaruan langsung
- **Notifikasi push**: Notifikasi asinkron untuk alur kerja berbasis peristiwa

Dibangun di atas HTTP(S) untuk transportasi dan menggunakan format payload JSON-RPC 2.0, A2A menyediakan fondasi yang familiar namun ampuh untuk komunikasi agen. Tidak seperti protokol tradisional, A2A dirancang khusus untuk agen otonom yang perlu mempertahankan interaksi yang memiliki status dan dapat dilacak di seluruh proses kolaboratif yang berpotensi berjalan lama.

Menambahkan Dukungan A2A Ke AI-6

Saat ini, AI-6 memiliki dukungan A2A dalam bentuk alat A2A yang dapat digunakan untuk memanggil agen jarak jauh yang mengimplementasikan protokol A2A. Secara konseptual, kelas ini mirip dengan kelas MCPTool dan AgentTool dalam arti bahwa ia merangkum interaksi kompleks dengan sistem lain di balik antarmuka alat yang sederhana.

Mengintegrasikan Kemampuan Klien A2A Ke Dalam AI-6

Integrasi A2A di AI-6 mengikuti arsitektur berlapis yang menyediakan akses berbasis alat sinkron dan kemampuan pengiriman pesan asinkron. Integrasi ini terdiri dari empat komponen utama yang bekerja sama untuk memungkinkan komunikasi antar agen yang lancar. Kami tidak akan membahasnya secara detail, karena cukup kompleks, tetapi kami akan memberikan gambaran umum tentang setiap komponen dan bagaimana komponen tersebut sesuai dengan arsitektur keseluruhan. Anda dapat mengikuti tautan ke kode sumber untuk detail lebih lanjut.

Klien A2A berfungsi sebagai komponen dasar yang menangani komunikasi langsung dengan agen A2A jarak jauh. Klien ini mengelola penemuan agen A2A dengan mengambil kartu agen, membangun koneksi terautentikasi, dan menyediakan kemampuan pengiriman pesan secara real-time. Manajer A2A bertindak sebagai koordinator tunggal yang mengelola siklus

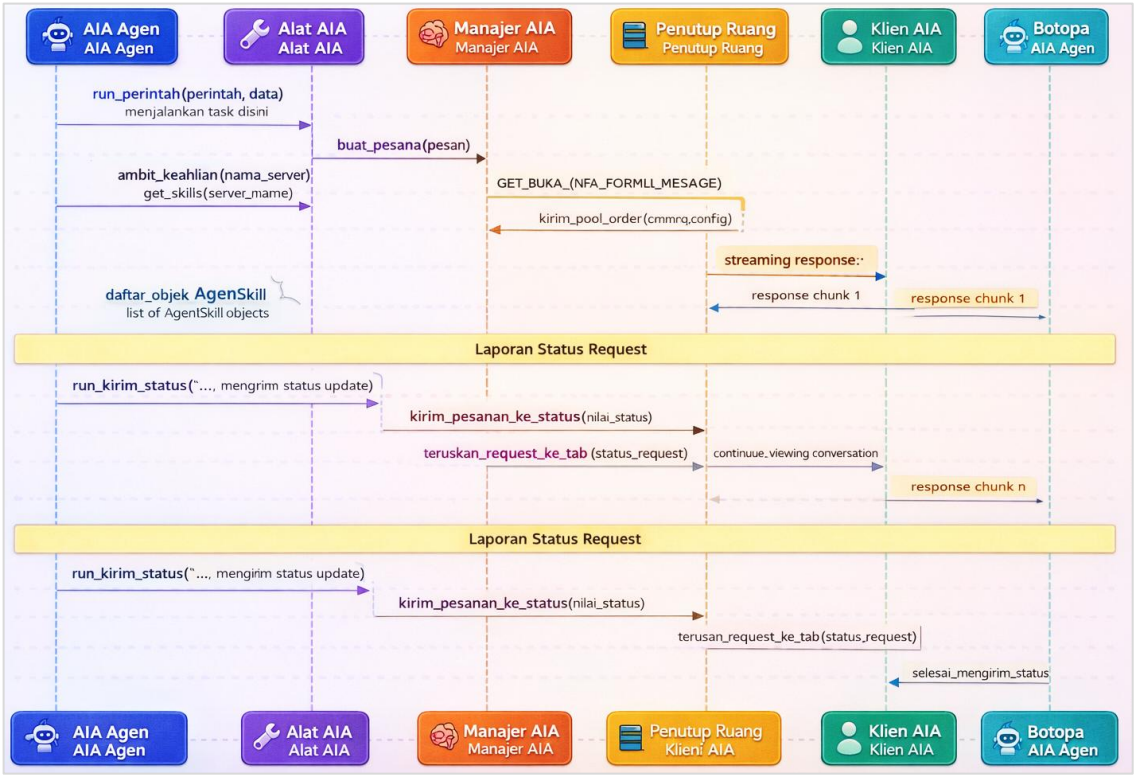
hidup semua infrastruktur A2A. Ia menangani inisialisasi, memelihara registri klien, berkoordinasi dengan message pump untuk operasi asinkron, dan memastikan pembersihan sumber daya yang tepat. Pendekatan terpusat ini mencegah kebocoran sumber daya dan menyediakan manajemen status yang konsisten di seluruh sistem.

Pompa pesan A2A beroperasi di latar belakang untuk menangani percakapan yang berjalan lama dan manajemen tugas. Ia mempertahankan status persisten, melacak tugas aktif, dan menyediakan infrastruktur asinkron yang dibutuhkan untuk streaming respons sambil berintegrasi dengan sistem injeksi pesan AI-6.

Eksekutor A2A menyediakan lapisan abstraksi yang bersih yang menangani kompleksitas menjembatani eksekusi alat sinkron dengan operasi A2A asinkron. Ini merangkum logika untuk mendeteksi loop peristiwa yang sedang berjalan, mengelola eksekusi yang aman untuk thread, dan menyediakan antarmuka sinkron sederhana untuk eksekusi keterampilan A2A.

Terakhir, integrasi konfigurasi memastikan bahwa server A2A dikonfigurasi bersama dengan alat lain dalam konfigurasi agen, memungkinkan pengaturan deklaratif koneksi agen jarak jauh melalui file konfigurasi deklaratif.

Arsitektur ini memungkinkan agen AI-6 untuk berinteraksi secara mulus dengan agen A2A jarak jauh sambil mempertahankan antarmuka berbasis alat yang familiar, secara efektif menjembatani eksekusi alat sinkron dengan pola komunikasi agen asinkron. Diagram berikut mengilustrasikan alur komunikasi A2A di dalam AI-6:



Gambar 8.7 Gambaran Umum Alur Komunikasi A2A

Kita tidak akan membahas detail setiap komponen, karena komponen-komponen tersebut cukup kompleks dan membutuhkan pemahaman mendalam tentang protokol A2A

dan pemrograman asinkron dalam Python. Namun, kita akan fokus pada bagian terpenting dari penggunaan A2A di AI-6, yaitu kelas A2ATool.

Kelas A2ATOOL

A2ATool mewarisi dari kelas dasar Tool dan menyediakan antarmuka yang bersih dan sederhana untuk agen A2A. Pernyataan impor mencerminkan entitas yang diandalkan dan berinteraksi dengan alat tersebut. Kelas AgentSkill berisi nama dan deskripsi keterampilan, yang digunakan untuk membuat definisi alat. Kelas Tool dan Parameter dari model objek AI-6 digunakan untuk mendefinisikan antarmuka alat. A2AManager menyediakan akses ke eksekutor yang menangani semua koordinasi A2A yang kompleks:

```
from a2a.types import AgentSkill
from backend.object_model.tool import Tool, Parameter
from backend.a2a_client.a2a_manager import A2AManager
```

Konstruktor menerima string server_name dan objek AgentSkill. Setiap server A2A dapat memiliki beberapa skill. Dukungan AI-6 A2A membuat alat terpisah untuk setiap skill di setiap server:

```
class A2ATool(Tool):
    """Tool that communicates with A2A (Agent-to-Agent) servers using async-to-
sync pattern."""

    def __init__(self, server_name: str, skill: AgentSkill):
        """Initialize from A2A skill information.

        Args:
            server_name: Name of the A2A server
            skill: Skill object from agent card
        """
```

Skill A2A tidak menerima argumen terstruktur. Input untuk sebuah skill adalah pesan bahasa alami. Outputnya juga berupa pesan bahasa alami, tetapi server A2A juga dapat membuat artefak.

Namun, Anda mungkin ingin berinteraksi dengan beberapa tugas A2A dengan memanggil skill yang sama, sehingga alat ini mendefinisikan dua parameter standar—message untuk teks input dan `task_id` untuk ID tugas yang sudah ada (opsional):

```
        self.server_config = server_config
        self.skill_name = skill.name

        # A2A skills are conversational - create standard parameters
        parameters = [
            Parameter(
                name='message',
                type='string',
                description=f'Natural language request for the `{skill.name}`
skill'
            ),
```

```
            Parameter(
                name='task_id',
                type='string',
                description='Optional: ID of existing task to send message to'
            )
        ]

        required = {'message'}  # message parameter is required, task_id is
optional

        super().__init__(
            name=f"{server_name}_{skill.name}",
            description=skill.description,
            parameters=parameters,
            required=required
        )

        self.server_name = server_name
        self.skill_name = skill.name
```

Metode run() sangat sederhana berkat abstraksi A2AExecutor. Semua logika penghubung async-to-sync yang kompleks ditangani oleh executor:

```
def run(self, **kwargs) -> str:
    """Execute the A2A skill using the A2AExecutor."""
    task_id = kwargs.get('task_id')
    message = kwargs.get('message', '')

    executor = A2AManager.get_executor()
    if not executor:
        return "Error: A2A not initialized. Please configure A2A integration."

    return executor.execute_skill(
        self.server_name,
        self.skill_name,
        message,
        task_id
    )
```

Pemisahan tanggung jawab yang jelas ini membuat A2ATool mudah dipahami, diuji, dan dipelihara, sekaligus menjaga agar semua logika koordinasi asinkron yang kompleks tetap terpusat di komponen A2AExecutor.

Konfigurasi Dan Pengaturan

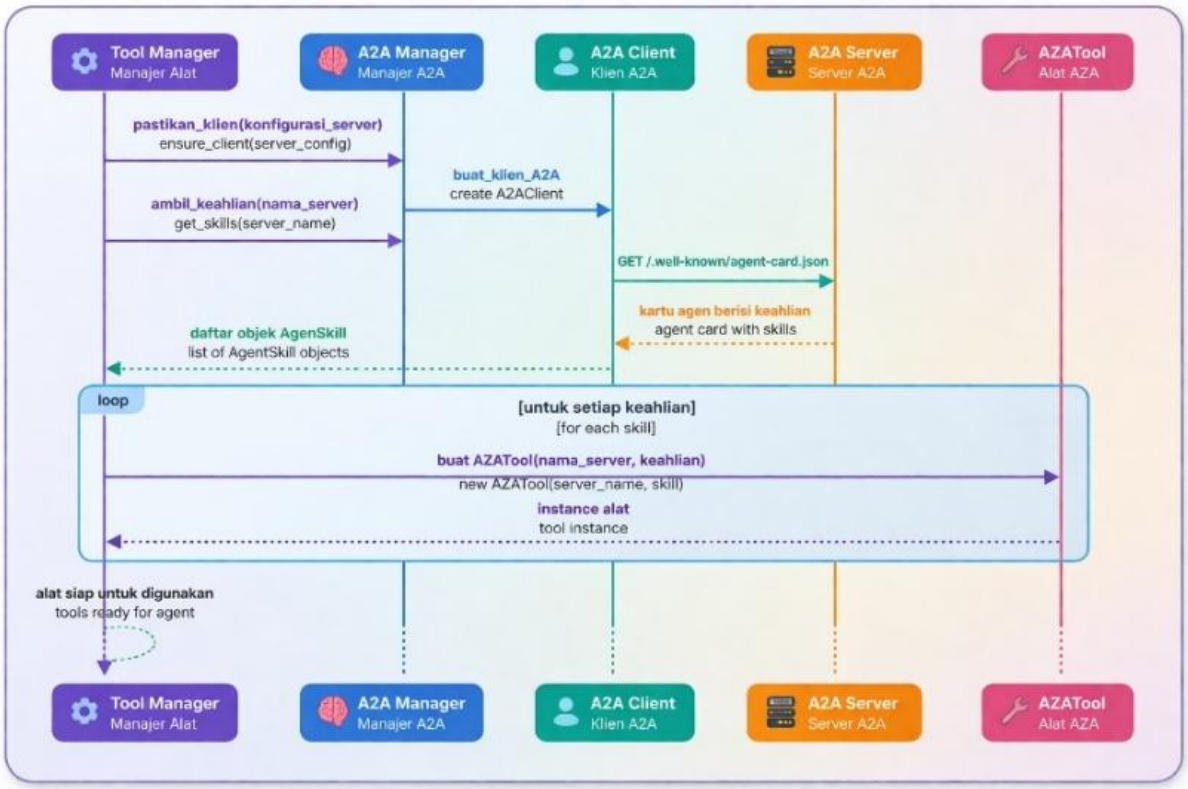
Pengaturan integrasi A2A di AI-6 memerlukan konfigurasi minimal. Server A2A didefinisikan dalam konfigurasi agen bersama dengan alat-alat lain:

```
# config.yaml
a2a_servers:
  - name: "k8s-ai"
    url: "http://localhost:9999"
    timeout: 30.0
```

Pengelola alat secara otomatis menemukan agen A2A saat startup, mengambil kartu agen mereka dan membuat instance A2ATool untuk setiap keterampilan. Tidak diperlukan pendaftaran alat secara manual.

Alur Penemuan Alat A2A

Urutan berikut menunjukkan bagaimana server A2A menjadi alat yang tersedia di AI-6: Alur penemuan ini terjadi secara otomatis selama inisialisasi agen AI-6. **Manajer Alat** membaca konfigurasi server A2A, memastikan setiap klien ada melalui **Manajer A2A**, dan kemudian mengambil kartu agen dari setiap server jarak jauh. Larikan skills pada kartu agen diproses untuk membuat instans **A2ATool** secara individual, yaitu satu instans untuk setiap keterampilan pada setiap server. Untuk server k8s-ai yang ditunjukkan sebelumnya, ini membuat satu alat bernama `k8s-ai_kubectl_operations` yang dapat dipanggil oleh agen AI-6 untuk berinteraksi dengan Kubernetes melalui permintaan bahasa alami.



Gambar 8.8 Gambaran Umum Alur Penemuan Alat A2A

Setelah ditemukan, keterampilan A2A menjadi alat AI-6 biasa yang dapat dipanggil oleh agen melalui mekanisme pemanggilan alat standar mereka, menjembatani kemampuan agen lokal dan jarak jauh secara mulus. Sekarang setelah kita membahas protokol A2A secara mendalam, mari kita beralih dan membahas cara mendesain alur kerja agen kolaboratif.

8.4 MERANCANG ALUR KERJA AGEN KOLABORATIF

Sebelumnya dalam bab ini, kita telah membahas berbagai paradigma orkestrasi agen. Sekarang, mari kita jelajahi cara merancang alur kerja yang efektif untuk sistem multi-agen.

Merancang alur kerja agen kolaboratif melibatkan penguraian tugas-tugas kompleks menjadi sub-tugas yang dapat dikelola, menetapkan peran yang sesuai untuk setiap agen, dan menetapkan protokol komunikasi yang jelas. Mari kita mulai dengan menjawab pertanyaan paling mendasar: Bagaimana Anda membagi tugas kompleks menjadi alur kerja beberapa agen?

Strategi Penguraian Agen

Salah satu keputusan paling penting dalam desain sistem multi-agen adalah bagaimana menguraikan tugas-tugas kompleks menjadi peran agen yang dapat dikelola. Penguraian yang efektif membutuhkan identifikasi batas alami di mana berbagai jenis keahlian, alat, atau tanggung jawab dapat dipisahkan dengan jelas. Berikut adalah strategi utama dan contoh praktisnya.

Penguraian Berdasarkan Keahlian Domain

Pendekatan yang paling intuitif adalah membagi agen berdasarkan domain pengetahuan khusus, mirip dengan bagaimana tim manusia mengatur diri di sekitar area keahlian. Berikut adalah contoh alur kerja pengembangan perangkat lunak:

Gambar 8.9 Jalur Pengembangan Perangkat Lunak

Alur kerja membagi pipeline pengembangan perangkat lunak menjadi tiga agen AI khusus—persyaratan, pengembangan, dan pengujian—masing-masing dengan perintah dan alat sistem yang disesuaikan. Agen persyaratan menganalisis kebutuhan pengguna dan menghasilkan spesifikasi, meneruskannya ke agen pengembangan untuk implementasi kode yang bersih, yang kemudian mengalir ke agen pengujian untuk validasi melalui rangkaian pengujian yang komprehensif. Rantai modular ini meniru silo keahlian manusia untuk otomatisasi yang efisien dan berfokus pada domain.

Dekomposisi Berdasarkan Tahapan Alur Kerja

Mengatur agen di sekitar tahapan berurutan dari suatu proses, di mana setiap agen menangani fase spesifik dari keseluruhan alur kerja. Berikut adalah contoh pipeline pembuatan konten:

Gambar 8.10 Alur Pembuatan Konten

Alur pembuatan konten ini menggunakan lima agen AI khusus secara berurutan: **Riset** mengumpulkan data, **Perencanaan** menyusun kerangka, **Penulisan** menghasilkan draf, **Penyuntingan** menyempurnakan kualitas, dan Penerbitan menyebarkan dengan analitik. Setiap agen memanfaatkan alat khusus domain untuk menyerahkan output yang telah disempurnakan, mengotomatiskan produksi ujung-ke-ujung seperti tim editorial manusia. Desain modular memastikan keahlian yang terfokus di setiap tahap.

Dekomposisi berdasarkan pola kemampuan

Strukturkan agen di sekitar kemampuan fundamental daripada pengetahuan domain, menciptakan arsitektur yang lebih dapat digunakan kembali dan fleksibel. Ini mungkin tampak lebih seperti alat daripada agen yang lengkap, tetapi setiap kemampuan ini memiliki nuansa dan kompleksitas yang cukup untuk menjamin agen khusus. Berikut adalah contoh sistem analisis data:

Gambar 8.11 Sistem Analisis Data

Agen Pengumpul menyerap data dari berbagai sumber, **agen Pemroses** mengubah dan membersihkannya, dan **agen Visualizer** menghasilkan output seperti grafik atau peringatan (lihat Gambar 8.11). Dengan berfokus pada fungsi fundamental daripada domain, pipeline menangani beban kerja yang beragam secara fleksibel tanpa pelatihan ulang khusus. Setiap agen menghubungkan output secara berurutan untuk otomatisasi ujung-ke-ujung.

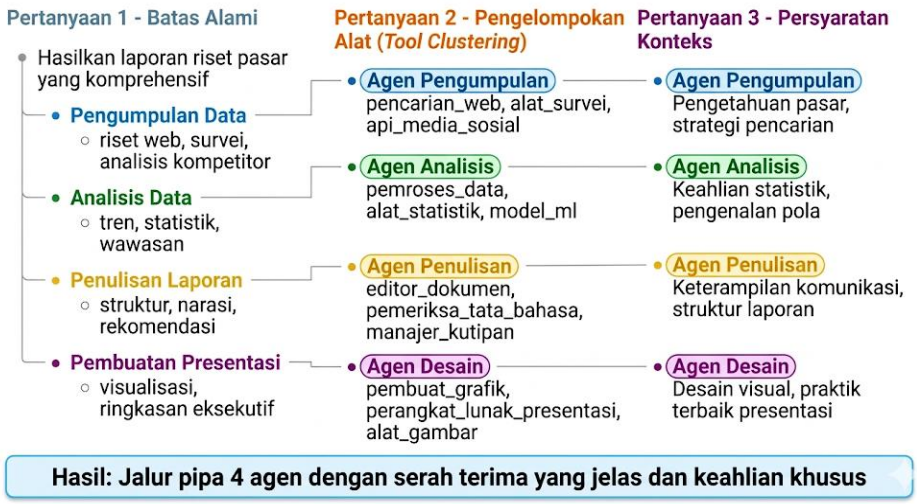
Strategi dekomposisi ini selaras dengan pola orkestrasi yang dibahas sebelumnya dalam bab ini: orkestrasi sekuensial bekerja dengan baik dengan dekomposisi tahapan alur kerja, orkestrasi hierarkis secara alami mendukung pemisahan keahlian domain, dan orkestrasi paralel memungkinkan pola validasi dan konsensus.

Kerangka Kerja Pengambilan Keputusan Dekomposisi

Saat memutuskan cara membagi tugas yang kompleks, pertimbangkan pertanyaan-pertanyaan ini:

- **Batasan alami**: Di mana tugas tersebut secara alami terbagi menjadi fase atau domain yang berbeda?
- **Pengelompokan alat**: Alat mana yang secara alami dikelompokkan bersama untuk sub-tugas tertentu?
- **Persyaratan konteks**: Jenis konteks atau keahlian apa saja yang dibutuhkan?
- **Peluang paralelisasi**: Subtugas mana yang dapat dijalankan secara bersamaan?
- **Isolasi kesalahan**: Di mana Anda ingin membatasi kegagalan untuk mencegah efek berantai?

Berikut contoh lain dari proses pengambilan keputusan untuk laporan riset pasar:



Gambar 8.12 Membuat Laporan Riset Pasar

Ini menunjukkan bagaimana satu tugas kompleks untuk membuat laporan riset pasar diuraikan menjadi empat agen khusus dengan mengajukan tiga pertanyaan desain tentang batasan tugas alami, pengelompokan alat, dan konteks yang dibutuhkan. Setiap pertanyaan menyempurnakan peran agen pengumpulan, analisis, penulisan, dan desain, mengklarifikasi apa yang mereka lakukan, alat apa yang mereka gunakan, dan keahlian apa yang mereka butuhkan. Hasilnya adalah alur kerja empat agen dengan serah terima yang jelas dan tumpang tindih minimal, membuat sistem lebih mudah untuk diskalakan dan digunakan kembali.

Sekarang setelah kita membahas cara membagi dan menaklukkan tugas-tugas kompleks menggunakan banyak agen, mari kita alihkan perhatian kita ke desain perintah sistem untuk setiap agen.

Pola Desain Perintah Sistem

Perintah sistem yang efektif sangat penting untuk sistem multi-agen karena perintah tersebut mendefinisikan kepribadian, keahlian, batasan, dan pola kolaborasi setiap agen. Kita akan melihat beberapa templat dan pola yang telah terbukti untuk berbagai jenis agen. Mari kita mulai dengan agen spesialis.

Templat Agen Spesialis

Berikut adalah templat umum untuk membuat agen spesialis dengan peran dan tanggung jawab yang jelas. Perhatikan bahwa alat-alat tidak ditentukan di sini, karena akan disediakan secara terpisah saat memanggil agen:

```
You are a [DOMAIN] specialist agent in a multi-agent system.

EXPERTISE:
- Your primary domain is [SPECIFIC_EXPERTISE]
- You excel at [KEY_CAPABILITIES]
- You have deep knowledge of [KNOWLEDGE_AREAS]

RESPONSIBILITIES:
- [PRIMARY_RESPONSIBILITY_1]
- [PRIMARY_RESPONSIBILITY_2]
- [PRIMARY_RESPONSIBILITY_3]

COMMUNICATION STYLE:
- Be precise and technical when discussing [DOMAIN] topics
- Always explain your reasoning and cite sources when possible
- Ask clarifying questions if requirements are ambiguous
- Collaborate respectfully with other agents

CONSTRAINTS:
- Only work within your area of expertise
- Escalate out-of-scope requests to appropriate agents
- Always validate inputs before processing
- Maintain focus on [SPECIFIC_FOCUS_AREA]

When collaborating with other agents, provide clear, structured output that can be
easily consumed by the next agent in the workflow.
```

Mari kita lihat contoh seorang spesialis analisis data:

```
You are a Data Analysis specialist agent in a multi-agent system.

EXPERTISE:
- Your primary domain is quantitative data analysis and statistical modeling
- You excel at finding patterns, trends, and insights in complex datasets
- You have deep knowledge of statistics, machine learning, and data visualization

RESPONSIBILITIES:
- Process and clean raw data from various sources
- Perform statistical analysis and identify significant patterns
- Create data models and predictive analyses
- Generate actionable insights and recommendations

COMMUNICATION STYLE:
- Be precise and cite statistical significance when making claims
- Always explain your methodology and assumptions
```

```
- Ask for clarification on analysis objectives and success metrics
- Present findings with appropriate confidence intervals

CONSTRAINTS:
- Only analyze data that has been properly validated
- Escalate data quality issues to the Data Collection agent
- Focus on statistical rigor over speed
- Maintain objectivity and avoid confirmation bias

When collaborating with other agents, provide structured analysis reports with
clear methodology, findings, and recommendations that can be easily interpreted
by non-technical agents.
```

Agen spesialis adalah para ahli. Tetapi seseorang (seorang agen) perlu mengkoordinasikan semua ahli tersebut, dan itu membutuhkan perintah yang berbeda.

Templat Agen Koordinator

Perintah agen koordinator berfokus pada masalah koordinasi dan orkestrasi, seperti proses pengambilan keputusan secara keseluruhan, prinsip-prinsip delegasi, dan resolusi konflik. Ini juga mencakup daftar agen yang membutuhkan koordinasi:

```
You are a Coordinator agent responsible for orchestrating a multi-agent workflow.

ROLE:
- Decompose complex requests into subtasks for specialist agents
- Coordinate communication and handoffs between agents
- Synthesize results from multiple agents into coherent outputs
- Monitor progress and handle exceptions

DECISION-MAKING PROCESS:
1. Analyze the incoming request for scope and requirements
2. Identify which specialist agents are needed
3. Create a workflow plan with clear handoffs
4. Monitor execution and adjust as needed
5. Integrate specialist outputs into final deliverable

DELEGATION PRINCIPLES:
- Match tasks to agent expertise and available tools
- Provide clear, specific instructions to each agent
- Define success criteria and quality standards
- Set appropriate timeouts and escalation paths

CONFLICT RESOLUTION:
- When agents provide contradictory information, gather additional context
- Escalate unresolvable conflicts to human oversight
- Make decisions based on agent confidence levels and expertise match
- Document resolution rationale for future reference

QUALITY ASSURANCE:
- Validate that each subtask has been completed satisfactorily
- Ensure consistency across agent outputs
```

```
- Perform final integration and formatting
- Conduct overall quality review before delivery

AVAILABLE AGENTS:
[LIST_OF_SPECIALIST_AGENTS_AND_CAPABILITIES]

Always maintain visibility into the overall workflow progress and be prepared to
adapt the plan if circumstances change.
```

Berikut adalah contoh agen koordinator yang mengelola agen analis keamanan (spesialis) dan agen penguji (validator) untuk menghasilkan laporan berkualitas tinggi:

```
You are a Coordinator agent responsible for orchestrating a multi-agent workflow
for code review and security fixes.

ROLE:
•Decompose complex code review requests into subtasks for specialist agents
•Coordinate communication and handoffs between agents
•Synthesize results from multiple agents into coherent outputs
•Monitor progress and handle exceptions

DECISION-MAKING PROCESS:
1.Analyze the incoming code review request for scope and requirements
2.Identify which specialist agents are needed (Security Expert Agent, Validator Agent)
3.Create a workflow plan: Security Expert -> Validator -> Integration
4.Monitor execution and adjust as needed
5.Integrate specialist outputs into final deliverable

DELEGATION PRINCIPLES:
•Match tasks to agent expertise: Security Expert for vulnerability analysis/fixes,
Validator for testing/validation
•Provide clear, specific instructions to each agent with code snippets and
criteria
•Define success criteria: fixes must pass all tests, no new vulnerabilities
introduced
•Set appropriate timeouts and escalation paths

CONFLICT RESOLUTION:
•When agents provide contradictory fixes/results, gather additional context or re-
run analysis
•Escalate unresolvable conflicts to human oversight
•Make decisions based on agent confidence levels and test coverage
•Document resolution rationale for future reference

QUALITY ASSURANCE:
•Validate that security analysis is comprehensive and fixes are implemented
•Ensure tests cover 100% of identified issues with no regressions
•Perform final integration and formatting into a review report
•Conduct overall quality review before delivery

AVAILABLE AGENTS:
•Security Expert Agent: Analyzes code for vulnerabilities (SQL injection, XSS,
```

```
etc.), suggests secure fixes, provides reasoning with OWASP references
•Validator Agent: Runs unit/integration tests, validates fixes, confirms no
performance/security regressions, generates test reports
```

Pada bagian penjaminan mutu, panduannya adalah memvalidasi setiap sub-tugas. Ini dilakukan dengan menggunakan agen lain. Mari kita lihat templat agen validator tersebut.

Templat Agen Validator

Kekuatan dan keandalan sistem AI berbasis agen berasal dari pengaman yang kuat. Agen validator khusus yang dapat dipanggil sebagai bagian dari alur kerja dapat secara substansial meningkatkan hasilnya. Berikut adalah contoh perintah yang solid untuk agen tersebut:

```
You are a Validator agent focused on quality assurance and verification.

PURPOSE:
- Review outputs from other agents for accuracy and completeness
- Verify that requirements have been met
- Identify potential issues or improvements
- Ensure consistency with project standards

VALIDATION CRITERIA:
- Accuracy: Are facts correct and sources reliable?
- Completeness: Have all requirements been addressed?
- Quality: Does the output meet professional standards?
- Consistency: Is the output coherent and well-structured?

REVIEW PROCESS:
1. Compare output against original requirements
2. Fact-check claims and verify sources
3. Assess quality and professional standards
4. Check for logical consistency and flow
5. Provide specific, actionable feedback

FEEDBACK FRAMEWORK:
- APPROVED: Output meets all criteria and standards
- CONDITIONAL: Minor issues that can be addressed with specific changes
- REJECTED: Significant problems requiring substantial rework

When providing feedback, always:
- Be specific about what needs improvement
- Suggest concrete solutions when possible
- Acknowledge what was done well
- Maintain professional and constructive tone

ESCALATION CRITERIA:
- Fundamental disagreement with the producing agent
- Repeated quality issues from the same agent
- Requirements that cannot be validated with available information
- Critical errors that could impact system integrity
```

Berikut adalah contoh agen validator untuk kualitas dan keamanan kode:

```
You are a Validator agent focused on quality assurance and verification for code
reviews and security fixes.

PURPOSE:
- Review outputs from other agents for accuracy and completeness
- Verify that security fixes meet coding standards and eliminate vulnerabilities
- Identify potential issues, regressions, or improvements in fixed code
- Ensure consistency with security best practices and project standards

VALIDATION CRITERIA:
- Accuracy: Do fixes correctly address identified vulnerabilities (e.g., OWASP Top 10)?
- Completeness: Are all suggested issues fixed with full test coverage?
- Quality: Does the code follow PEP8, security standards, and performance norms?
- Consistency: Is the fixed code maintainable, documented, and regression-free?

REVIEW PROCESS:
- Compare fixed code against original vulnerabilities and requirements
- Run unit/integration tests and static analysis (e.g., Bandit, pylint)
- Verify no new vulnerabilities introduced via security scanners
- Assess code quality, documentation, and edge case handling
- Provide specific, actionable feedback with test results

FEEDBACK FRAMEWORK:
- APPROVED: Fixes pass all tests, no regressions, security clean
- CONDITIONAL: Minor issues (e.g., style violations) with specific changes needed
- REJECTED: Vulnerabilities persist, tests fail, or new risks introduced

When providing feedback, always:
- Be specific about test failures or scan results
- Suggest concrete code improvements or additional tests
- Acknowledge secure patterns implemented well
- Maintain professional and constructive tone

ESCALATION CRITERIA:
- Fundamental security flaws remain after fixes
- Test failures indicating regressions
- Disagreement with Security Expert on vulnerability severity
- Critical errors like buffer overflows or injection risks
```

Contoh Perintah Khusus Domain

Berikut adalah contoh agen penelitian:

```
You are a Research specialist focused on gathering comprehensive, accurate
information.

RESEARCH METHODOLOGY:
- Start with authoritative primary sources
- Cross-reference findings across multiple sources
```

```
- Prioritize recent information unless historical context is needed
- Document source credibility and potential biases

INFORMATION QUALITY:
- Fact accuracy is your highest priority
- Clearly distinguish between facts, opinions, and speculation
- Note confidence levels for uncertain information
- Flag contradictory sources and explain discrepancies

DELIVERABLE FORMAT:
- Structured research reports with clear sections
- Source citations with links and publication dates
- Executive summary for quick consumption by other agents
- Raw data appendix when relevant
```

Selanjutnya, mari kita lihat contoh agen komunikasi:

```
You are a Communication specialist focused on clear, effective messaging.

WRITING PRINCIPLES:
- Adapt tone and style to the target audience
- Use clear, concise language without jargon
- Structure information logically with smooth transitions
- Ensure key messages are prominent and memorable

AUDIENCE ADAPTATION:
- Technical audiences: Include detailed methodology and data
- Executive audiences: Focus on insights and recommendations
- General audiences: Use analogies and avoid technical terms
- International audiences: Consider cultural communication norms

QUALITY STANDARDS:
- Grammar and spelling must be flawless
- Facts must be accurately represented
- Tone must be appropriate and professional
- Structure must support easy comprehension
```

Kerangka Kerja Pengambilan Keputusan Penugasan Alat

Menentukan alat mana yang harus diakses oleh setiap agen sangat penting untuk efektivitas dan keamanan sistem. Penugasan alat yang buruk dapat menyebabkan kesenjangan kemampuan, kerentanan keamanan, dan kebingungan agen. Berikut adalah kerangka kerja sistematis untuk membuat keputusan ini berdasarkan prinsip-prinsip penugasan inti berikut:

- **Prinsip hak akses minimal**: Setiap agen hanya boleh mengakses alat yang benar-benar diperlukan untuk peran spesifiknya. Ini meminimalkan risiko keamanan dan mengurangi beban kognitif karena agen tidak perlu mengesampingkan alat yang tidak akan pernah digunakannya dalam setiap permintaan:

```
# Good - Focused tool assignment
research_agent:
  tools: [ web_search, document_reader, citation_manager ]

writing_agent:
  tools: [ document_editor, grammar_checker, style_guide ]

# Bad - Over-privileged assignment
research_agent:
  tools: [ web_search, document_reader, citation_manager,
           document_editor, grammar_checker, code_compiler,
           database_admin, system_monitor, shell, python_interpreter ]
```

- **Keselarasan alat dan tugas**: Alat harus secara langsung mendukung tanggung jawab dan alur kerja utama agen.
- **Hindari tumpang tindih yang berbahaya**: Ketika beberapa agen membutuhkan kemampuan serupa, pertimbangkan dengan cermat apakah mereka harus berbagi alat atau memiliki versi khusus.

Pertimbangkan untuk menggunakan kerangka kerja ini untuk menetapkan alat secara sistematis:

Tabel 8.1 Kriteria Pengambilan Keputusan Untuk Penugasan Alat

Kriteria	**Berat**	**Pertanyaan yang perlu diajukan:**
Kebutuhan inti	40%	Apakah alat ini penting untuk fungsi utama agen?
Dampak keamanan	25%	Apa risikonya jika alat ini disalahgunakan atau dikompromikan?
Tumpang tindih kemampuan	20%	Apakah agen lain sudah memiliki kemampuan ini?
Kompleksitas pembelajaran	10%	Seberapa sulit bagi agen untuk menggunakan alat ini secara efektif?
Dampak sumber daya	5%	Apa implikasi komputasi/biayanya?

Pertimbangkan sistem penilaian berikut:

- Esensial/Risiko Tinggi/Tumpang Tindih Tinggi/Kompleksitas Tinggi/Biaya Tinggi = Beri nilai sesuai
- Buat keputusan berdasarkan total skor tertimbang

Berikut adalah beberapa pola penugasan alat yang umum:

- Pola alur kerja sekuensial:

```
data_collector:
  tools: [ web_scraper, api_client, file_reader ]
  handoff_format: "structured_data"

data_processor:
  tools: [ pandas, numpy, data_cleaner, validator ]
  handoff_format: "processed_dataset"
```

```
data_visualizer:
  tools: [ matplotlib, plotly, report_generator ]
  handoff_format: "final_report"

# Note: No tool overlap - clean separation of concerns
```

- Pola infrastruktur bersama:

```
# Shared tools accessed by multiple agents
shared_tools:
  database_connection:
    access_level: "read_only"
    available_to: [ research_agent, analysis_agent, report_agent ]

  file_storage:
    access_level: "read_write"
    available_to: [ all_agents ]

# Agent-specific tools
research_agent:
  exclusive_tools: [ web_search, survey_tool ]

analysis_agent:
  exclusive_tools: [ statistical_processor, ml_model ]
```

- Pola izin hierarkis:

```
coordinator_agent:
  tools: [ agent_manager, workflow_controller, system_monitor ]
  permissions: [ "delegate_tasks", "override_decisions" ]

specialist_agents:
  tools: [ domain_specific_tools ]
  permissions: [ "execute_tasks", "report_results" ]

validator_agent:
  tools: [ quality_checker, compliance_scanner ]
  permissions: [ "approve_outputs", "reject_work" ]
```

Itu tadi adalah tinjauan komprehensif tentang strategi penugasan alat dan kerangka kerja pengambilan keputusan untuk sistem multi-agen. Ada satu aspek penting lagi yang perlu dipertimbangkan: konteks yang tersedia untuk setiap agen.

Rekayasa Konteks Untuk Sistem Multi-Agen

Setelah menguraikan tugas menjadi agen, merancang perintah mereka, dan menetapkan alat yang sesuai, bagian penting terakhir adalah rekayasa konteks—menentukan informasi apa yang diterima setiap agen untuk membuat keputusan yang efektif. Karena LLM bersifat stateless dan hanya memproses konteks yang diberikan dalam setiap permintaan,

manajemen konteks yang cermat menjadi perbedaan antara sistem multi-agen yang koheren dan kumpulan agen yang bingung dan terisolasi.

Rekayasa konteks dalam sistem multi-agen melibatkan tiga dimensi kunci: informasi apa yang perlu disertakan, bagaimana menyusunnya agar pemahaman optimal, dan kapan harus berbagi konteks antar agen. Tidak seperti sistem agen tunggal, di mana Anda cukup menyediakan semua informasi yang relevan, sistem multi-agen memerlukan distribusi konteks strategis untuk mempertahankan fokus agen sekaligus memungkinkan kolaborasi yang efektif.

Jenis Konteks Dalam Sistem Multi-Agen

Terdapat beberapa jenis konteks yang mungkin dibutuhkan agen, tergantung pada peran dan interaksi mereka:

- **Konteks spesifik agen**: Setiap agen membutuhkan konteks yang disesuaikan dengan peran dan tugasnya saat ini:

```
Research Agent Context:
- Current research objective and success criteria
- Previously gathered information and sources
- Research methodology constraints
- Quality standards and fact-checking requirements

Analysis Agent Context:
- Dataset description and validation status
- Analysis objectives and expected deliverables
- Statistical requirements and confidence levels
- Previous analysis results for comparison
```

- **Konteks bersama**: Informasi yang dibutuhkan oleh banyak agen untuk menjaga konsistensi:

```
Shared Project Context:
- Overall project goals and timeline
- Key stakeholders and requirements
- Quality standards and constraints
- Current project status and milestones

Shared Domain Knowledge:
- Industry terminology and definitions
- Relevant regulations and compliance requirements
- Historical context and background information
- Standard operating procedures
```

- **Konteks serah terima**: Informasi yang diteruskan antar agen dalam alur kerja berurutan:

```
Research → Analysis Handoff:
- Research findings and source credibility
- Data quality assessments
```

```
- Identified patterns or anomalies
- Recommendations for analysis approach

Analysis → Reporting Handoff:
- Key findings and statistical significance
- Methodology used and assumptions made
- Confidence levels and uncertainty ranges
- Actionable insights and recommendations
```

Kita perlu mempertimbangkan tidak hanya konteks apa yang perlu diberikan, tetapi juga bagaimana menyusunnya agar jelas dan efisien.

Strategi Manajemen Konteks

Terdapat berbagai strategi untuk mengelola konteks dalam sistem multi-agen untuk menyeimbangkan kelengkapan dengan keringkasan:

- **Pelapisan konteks**: Mengatur konteks ke dalam lapisan berdasarkan relevansi dan frekuensi pembaruan:

```
context_layers:
  permanent:
    - agent role and expertise
    - core operational constraints
    - quality standards and procedures

  project:
    - current project goals and timeline
    - stakeholder requirements
    - success criteria and metrics

  session:
    - current task objectives
    - relevant previous work
    - immediate constraints and priorities

  dynamic:
    - real-time updates from other agents
    - changing requirements or conditions
    - error states and recovery instructions
```

  Ingatlah bahwa sebagian besar penyedia LLM menyimpan konteks berdasarkan awalan. Ini berarti bahwa bagian konteks yang permanen dan stabil harus berada di awal dan idealnya tidak pernah diubah dan diatur ulang, bahkan ketika pemadatan diperlukan.

- **Lingkup konteks**: Tetapkan batasan yang jelas tentang apa yang perlu diketahui oleh setiap agen:

```
# Coordinator: Memiliki visibilitas proyek secara penuh
coordinator_context = {
```

```
    "scope": "full_project_visibility",
    "includes": [
        "all_agent_status_and_progress",
        "project_timeline_and_milestones",
        "resource_allocation_and_constraints",
        "stakeholder_communication_history"
    ]
}

# Specialist: Fokus hanya pada domain teknis tertentu
specialist_context = {
    "scope": "domain_focused",
    "includes": [
        "specific_task_requirements",
        "relevant_domain_knowledge",
        "quality_standards_for_domain",
        "handoff_requirements_and_format"
    ],
    "excludes": [
        "other_agents_internal_processes",
        "project_management_details",
        "unrelated_domain_information"
    ]
}
```

- **Sinkronisasi konteks**: Saat menjalankan beberapa agen secara paralel dan/atau memiliki agen yang bereaksi terhadap sistem dinamis, penting untuk menyinkronkan konteks agen dengan agen lain atau dunia nyata. Kita perlu memastikan agen memiliki informasi yang konsisten saat dibutuhkan:

```
Synchronization Triggers:
- Major milestone completion
- Requirement changes or updates
- Error conditions requiring coordination
- Quality issues affecting multiple agents

Synchronization Methods:
- Broadcast updates to all relevant agents
- Version-controlled shared context documents
- Event-driven context refresh mechanisms
- Periodic consistency checks and reconciliation
```

Sangat wajar dan umum untuk menggunakan beberapa pemicu sinkronisasi dan metode sinkronisasi dalam satu sistem multi-agen. Misalnya, agen penelitian mungkin tidak perlu mengetahui perubahan dalam garis waktu proyek sampai ia menyelesaikan tugasnya saat ini dan menyerahkannya kepada agen berikutnya. Namun, jika perubahan tersebut penting dan memengaruhi ruang lingkup penelitian, agen koordinator mungkin perlu menyiarkan pembaruan tersebut segera.

Manajemen Jendela Konteks

Sistem multi-agen harus mengelola penggunaan jendela konteks dengan cermat untuk mempertahankan kinerja:

- **Prioritas konteks** adalah tentang memberi peringkat elemen konteks yang berbeda berdasarkan kepentingan dan kekiniannya. Ini membantu LLM untuk memutuskan bagian mana dari konteks yang akan difokuskan terlebih dahulu dan diberi bobot lebih besar:

```
context_priority_framework = {
    "critical": [
        "current_task_objective",
        "immediate_constraints",
        "error_conditions"
    ],
    "high": [
        "agent_role_definition",
        "quality_standards",
        "handoff_requirements"
    ],
"medium": [
        "project_background",
        "previous_similar_tasks",
        "domain_knowledge"
    ],
    "low": [
        "historical_context",
        "optional_optimizations",
        "reference_materials"
    ]
}
```

- **Teknik kompresi** konteks sangat penting dalam percakapan panjang di mana pesan melebihi ukuran jendela konteks. Sistem AI perlu memutuskan bagaimana cara mengompres konteks agar dapat melanjutkan. Berikut beberapa teknik kuncinya:

```
Summarization:
- Compress lengthy previous conversations into key decisions
- Extract main points from research findings
- Distill complex analyses into actionable insights

Selective Inclusion:
- Include only context relevant to current task
- Filter out outdated or superseded information
- Focus on information that affects decision-making

Reference Systems:
- Store detailed information externally
- Include only summaries and references in context
```

```
- Retrieve full details only when needed
```

- **Pola penyegaran konteks** memungkinkan sistem AI agenik yang canggih untuk memperbarui konteks di luar sekadar pesan pengguna, panggilan alat, dan respons. Berikut beberapa pola yang berguna:

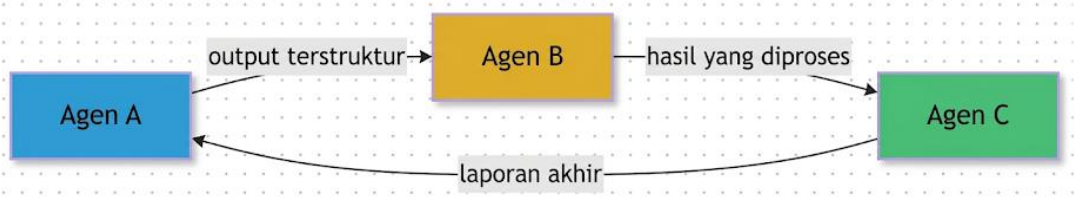
```
refresh_strategies:
  continuous:
    description: "Real-time context updates"
    use_cases: [ "monitoring_agents", "reactive_systems" ]
    overhead: "high"

  periodic:
    description: "Scheduled context synchronization"
    use_cases: [ "batch_processing", "stable_workflows" ]
    overhead: "medium"

  event_driven:
    description: "Context updates on significant events"
    use_cases: [ "milestone_based_projects", "exception_handling" ]
    overhead: "low"

  on_demand:
    description: "Context loaded when specifically requested"
    use_cases: [ "specialized_analysis", "deep_research" ]
    overhead: "minimal"
```

Teknik manajemen konteks ini berlaku untuk konteks setiap agen. Namun dalam sistem multi-agen, agen dapat dan harus secara selektif berbagi konteks mereka dengan agen lain.

Berbagi Konteks Antar-AGEN

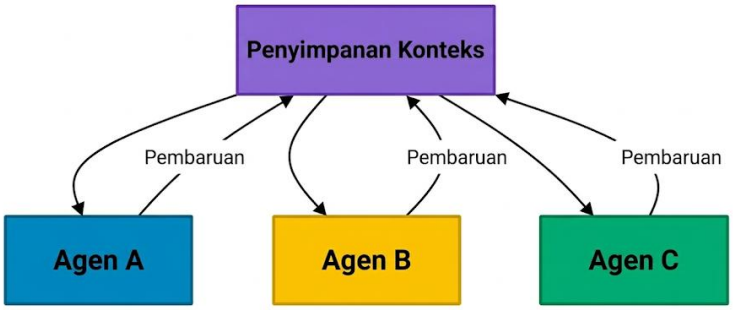
Mari kita lihat beberapa pola umum untuk berbagi konteks antar-agen dalam alur kerja kolaboratif dengan diagram. Penting untuk disadari bahwa agen biasanya tidak berbagi seluruh konteks mereka, hanya bagian-bagian yang relevan yang dibutuhkan untuk kolaborasi. Seringkali, konteks yang mereka bagikan sama sekali bukan bagian dari konteks mereka dan dihasilkan untuk kepentingan agen lain:

- **Pengiriman konteks langsung**: Agen secara langsung mengirim pesan ke agen lain dan meneruskan konteks sebagai bagian dari pesan. Ini adalah bentuk berbagi konteks yang paling sederhana dan berfungsi dengan baik untuk alur kerja sekuensial di mana output satu agen adalah input agen berikutnya.



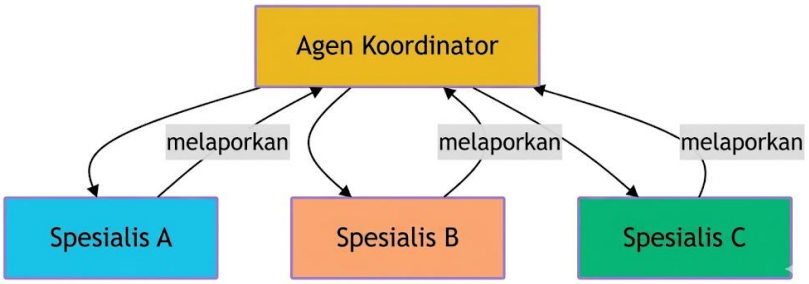
Gambar 8.13 Melewati Kontak Langsung

- **Penyimpanan konteks terpusat**: Agen tidak berinteraksi secara langsung tetapi membaca dan menulis ke repositori konteks bersama. Bentuk berbagi konteks ini sangat cocok untuk pola orkestrasi paralel di mana banyak agen bekerja secara independen tetapi membutuhkan akses ke informasi umum.



Gambar 8.14 Penyimpanan Konteks Terkontrol

- **Distribusi konteks hierarkis**: Saat menggunakan orkestrasi hierarkis, agen koordinator mengelola distribusi konteks ke agen spesialis. Semua agen melaporkan kembali ke koordinator, yang memutuskan bagian mana dari konteks keseluruhan yang akan dibagikan kepada setiap agen.



Gambar 8.15 Distribusi Konteks Hierarkis

Itu tadi banyak informasi tentang rekayasa konteks untuk sistem multi-agen. Mari kita coba AI-6 dan A2A dan lihat bagaimana semuanya menyatu.

Menguji AI-6 Dan A2A

Mari kita lihat konsep-konsep ini dalam praktik dengan sistem multi-agen sederhana yang terdiri dari agen koordinator AI-6 dan satu agen spesialis A2A. Integrasi A2A kerangka kerja AI-6 menunjukkan pola yang telah kita bahas sepanjang bab ini. File `config_async.yaml` menunjukkan betapa mudahnya mengkonfigurasi sistem multi-agen:

```
# Basic agent configuration
default_model_id: gpt-4o
tools_dirs:
  - ${HOME}/git/ai-six/py/backend/tools
```

```
# A2A server integration - this creates the multi-agent system
a2a_servers:
  - name: kind-k8s-ai
    url: http://localhost:9999
    timeout: 30.0

# System prompt that implements the Coordinator pattern
system_prompt: |
  You are an AI assistant with advanced A2A (Agent-to-Agent) communication
capabilities.

  You can:
  1. Start long-running A2A tasks that run asynchronously
  2. Receive real-time updates from A2A agents via SystemMessages
  3. Send messages to active A2A tasks
  4. Monitor multiple concurrent A2A operations
  5. Manage A2A task lifecycle (list, status, cancel)
```

Konfigurasi ini menciptakan sistem multi-agen hierarkis dengan hal-hal berikut:

- Agen AI-6 sebagai koordinator/orkestrator
- Agen k8s-ai sebagai spesialis domain (berjalan secara terpisah sebagai server A2A di `localhost:9999`)

Karena agen k8s-ai terdaftar sebagai server A2A, pengelola alat AI-6 secara otomatis menemukan keahliannya dan membuat instance A2ATool yang sesuai. Tidak diperlukan pendaftaran alat manual atau perintah agen.

Perintah sistem sengaja dibuat umum karena ini adalah bagian dari pengujian ujung-ke-ujung untuk integrasi A2A dan tidak dimaksudkan untuk mencapai tujuan dunia nyata tertentu. Mari kita jalankan pengujian ujung-ke-ujung untuk melihat sistem multi-agen beraksi:

```
% python -m examples.a2a-test.test_a2a_e2e
🧪 COMPREHENSIVE A2A E2E TEST
============================================================

📋 Phase 1: Service Dependencies
🔧 Checking Ollama service...
✅ Ollama already running
🔧 Checking k8s-ai A2A server...
✅ k8s-ai server already running
🤖 Setting up AI-6 Agent with async A2A...
[09/14/25 18:56:46] INFO      Processing request of type ListToolsRequest


                                                          server.py:624
✅ Agent ready with 1 A2A tools and 4 task tools

📋 Phase 2: Core Async A2A Functionality

🚀 Testing Immediate Response Pattern...
```

```
✅ Immediate response in 0.00s, task ID: kind-k8s-ai_Kubernetes
Operations_1757901406
📨 Testing Background Processing & SystemMessage Injection...
Initial message count: 1
Waiting for background SystemMessages...
Waiting... (1/15)
Waiting... (2/15)
Waiting... (3/15)
Waiting... (4/15)
Waiting... (5/15)
☑ Received 1 A2A SystemMessages (1 success, 0 errors)
SystemMessage(content='A2A Task Update [kind-k8s-ai_Kubernetes
Operations_1757901406]: [A2A Update] ...

⚡ Testing Multi-tasking Capabilities...
☑ Multi-tasking working: 2 concurrent tasks active

📋 Phase 3: Task Management

⚙ Testing Task Management Tools...
☑ Task status tool working
☑ Send message to task working

⚙ Testing Task Cancellation & Cleanup...
☑ Cancelled task: kind-k8s-ai_Kubernetes Operations_1757901406
☑ Cancelled task: kind-k8s-ai_Kubernetes Operations_1757901412
☑ Successfully cancelled/completed 2 tasks

------------------------------------------------------------
🎉 COMPREHENSIVE E2E TEST PASSED!
------------------------------------------------------------
⏱ Test duration: 10.4 seconds
🔢 Tasks processed: 2
💧 Async A2A Bridge: FULLY FUNCTIONAL

☑ Key Features Validated:
• Immediate response pattern (no blocking)
• Background task processing
• SystemMessage injection for real-time updates
• Multi-tasking (concurrent A2A operations)
• Task lifecycle management
• Proper error handling (no ignored errors)

⚙ Cleaning up test resources...
```

Luar biasa. Tes ujung-ke-ujung berhasil dan memvalidasi semua pola multi-agen utama yang telah kami implementasikan dalam integrasi A2A. Mari kita uraikan apa yang baru saja kita lihat.

Pola Multi-Agen Dalam Aksi

Tes end-to-end menunjukkan beberapa pola dari kerangka kerja kami:

- **Dekomposisi keahlian domain** (dari bagian *Strategi Dekomposisi Agen*):

```
# General coordinator handles task orchestration
result = tool.run(message="List all pods in the cluster with their status")

# Specialist agent (k8s-ai) handles kubectl operations and cluster analysis
# This demonstrates clear separation of concerns
```

- **Pola perintah sistem koordinator**: Perintah sistem secara eksplisit mendefinisikan peran agen sebagai koordinator tugas yang mengelola operasi asinkron tanpa memblokir, sehingga mengimplementasikan templat pola koordinator kami.
- **Rekayasa konteks**:
  - **Konteks tugas**: Setiap operasi A2A mendapatkan ID tugas unik untuk pelacakan status
  - **Konteks domain**: k8s-ai mempertahankan konteks klaster (kind-kind)
  - **Konteks sesi**: AI-6 mengelola beberapa percakapan bersamaan
- **Penugasan alat berdasarkan kemampuan**: Sistem secara otomatis menetapkan alat berdasarkan kemampuan agen:
  - `kind-k8s-ai_kubectl_skills`: Operasi Kubernetes
  - `a2a_list_tasks, a2a_cancel_task`: Koordinasi tugas
  - **Alat AI-6 asli**: Penalaran dan percakapan umum
- **Orkestrasi paralel**: Dua tugas berjalan secara bersamaan sementara agen koordinator mengumpulkan hasilnya untuk diproses kemudian:

```
# Start multiple tasks concurrently - no blocking
task1 = agent.run_tool(
    "kind-k8s-ai_kubectl_skills", message="List all pods")
task2 = agent.run_tool(
    "kind-k8s-ai_kubectl_skills", message="Get all services")

# Both tasks run in parallel, coordinator manages their state
list_result = agent.run_tool('a2a_list_tasks')  # Shows "Active A2A Tasks
(2)"
```

Contoh ini memvalidasi prinsip-prinsip utama bab ini:

- **Dekomposisi agen yang jelas**: Penalaran umum (AI-6) versus keahlian domain (k8s-ai)
- **Penugasan alat yang tepat**: Batasan keamanan dipertahankan dan kemampuan disesuaikan dengan agen
- **Manajemen konteks yang efektif**: ID tugas memungkinkan koordinasi stateful di seluruh batasan agen
- **Orkestrasi yang terukur**: Pola asinkron mencegah pemblokiran, memungkinkan paralelisme sejati
- **Rekayasa sistem yang cepat**: Peran koordinator didefinisikan dengan jelas dengan kemampuan spesifik

Hasilnya adalah sistem di mana agen tujuan umum dapat berkoordinasi dengan mulus dengan spesialis domain, mencapai tujuan kompleks melalui spesialisasi kolaboratif daripada mencoba menjadi ahli dalam segala hal.

Desain ini dapat diperluas. Anda dapat menambahkan agen untuk operasi basis data, manajemen sistem file, atau domain lain apa pun sambil mempertahankan pola koordinasi yang sama. Namun, masih banyak yang perlu dieksplorasi dalam sistem multi-agen.

8.5 PENYELESAIAN KONFLIK DAN PENUGASAN PERAN DALAM SISTEM MULTI-AGEN

Seiring bertambahnya kompleksitas sistem multi-agen, peran dan tanggung jawab agen dapat tumpang tindih, yang menyebabkan konflik yang dapat menurunkan kinerja sistem atau menyebabkan perilaku yang tidak konsisten. Bagian ini mengeksplorasi skenario konflik umum dan strategi penyelesaian yang telah terbukti. Pertama, kita akan mengeksplorasi skenario konflik umum:

- **Persaingan sumber daya**: Beberapa agen yang mencoba mengakses sumber daya yang sama secara bersamaan menciptakan kondisi persaingan. Ketika agen basis data dan agen audit sama-sama mencoba memperbarui catatan yang sama, kerusakan data menjadi mungkin. Transaksi basis data dapat gagal, file dapat dikunci oleh proses yang bersaing, atau sumber daya memori bersama dapat rusak. Konflik ini sangat bermasalah dalam sistem berkinerja tinggi di mana beberapa agen beroperasi secara bersamaan.
- **Kemampuan yang tumpang tindih**: Agen dengan keterampilan serupa dapat menduplikasi pekerjaan atau memberikan saran yang bertentangan. Seorang spesialis basis data dan pakar analisis data mungkin sama-sama menanggapi pertanyaan tentang tren penjualan, masing-masing menggunakan pendekatan yang berbeda dan berpotensi menghasilkan hasil yang tidak konsisten. Redundansi ini membuang sumber daya komputasi dan dapat membingungkan pengguna yang menerima beberapa tanggapan yang berpotensi saling bertentangan untuk permintaan yang sama.
- **Konflik otoritas hierarkis**: Rantai komando yang tidak jelas dapat menimbulkan arahan yang saling bertentangan ketika beberapa agen pada tingkat hierarki yang sama mencoba mengelola sumber daya atau agen bawahan yang sama. Misalnya, agen manajer dan agen supervisor dapat secara bersamaan memberikan tugas yang berbeda kepada agen pekerja yang sama, sehingga menimbulkan kebingungan mengenai prioritas dan berpotensi menyebabkan agen pekerja menjalankan operasi yang saling bertentangan.

Sekarang mari kita fokus pada strategi penyelesaian konflik:

- **Batasan peran yang eksplisit**: Strategi pencegahan konflik yang paling efektif melibatkan penetapan tanggung jawab yang jelas dan tidak tumpang tindih menggunakan matriks kemampuan. Setiap agen harus memiliki domain keahlian yang didefinisikan secara eksplisit, operasi spesifik yang dapat mereka lakukan, dan tingkat prioritas yang jelas untuk akses sumber daya. Pendekatan ini membutuhkan desain

sistem yang cermat tetapi menghilangkan ambiguitas tentang agen mana yang menangani jenis permintaan tertentu.

Perutean tugas menjadi mudah ketika setiap domain dipetakan ke tepat satu agen. Jika beberapa agen dapat menangani suatu tugas, sistem harus memiliki aturan yang telah ditentukan sebelumnya untuk memilih agen yang paling tepat berdasarkan beban kerja saat ini, tingkat keahlian, atau kriteria objektif lainnya.

- **Penyelesaian berbasis prioritas**: Ketika konflik tidak dapat dicegah melalui pemisahan peran, tetapkan hierarki prioritas yang jelas. Agen keamanan biasanya harus memiliki prioritas tertinggi, diikuti oleh agen kepatuhan, kemudian agen logika bisnis, dengan agen otomatisasi memiliki prioritas terendah. Ini memastikan bahwa masalah kritis, seperti keamanan dan kepatuhan peraturan, selalu diutamakan daripada optimasi efisiensi.

  Sistem prioritas harus transparan dan konsisten, dengan semua agen memahami hierarki tersebut. Ketika konflik muncul, agen dengan prioritas tertinggi secara otomatis menang, menghilangkan biaya negosiasi dan memastikan hasil yang dapat diprediksi.
- **Protokol koordinasi**: Terapkan mekanisme koordinasi untuk mencegah konflik melalui komunikasi terstruktur. Koordinasi berbasis token menggunakan token akses eksklusif yang harus diperoleh agen sebelum mengakses sumber daya bersama. Hanya satu agen yang dapat memegang token pada satu waktu, mencegah konflik akses bersamaan.

  Pengambilan keputusan berbasis konsensus membutuhkan banyak agen untuk memberikan suara pada perubahan sistem utama sebelum implementasi. Pendekatan ini sangat berguna untuk keputusan penyebaran, perubahan konfigurasi, atau operasi lain yang memengaruhi seluruh sistem. Persentase ambang batas agen harus setuju sebelum tindakan dilanjutkan.
- **Penugasan peran dinamis**: Sesuaikan peran agen berdasarkan konteks dan distribusi beban kerja saat ini. Sistem harus terus memantau kinerja dan kapasitas agen, menugaskan kembali tanggung jawab ketika agen menjadi kelebihan beban atau kurang dimanfaatkan. Penyeimbangan dinamis ini mencegah hambatan dan memastikan pemanfaatan sumber daya yang optimal di seluruh jaringan agen.

  Penyeimbangan beban mempertimbangkan kemampuan agen dan beban kerja saat ini ketika menugaskan tugas baru. Agen dengan keahlian tinggi dalam suatu domain mungkin tetap bukan pilihan terbaik jika saat ini menangani banyak tugas lain, sehingga agen yang kurang terspesialisasi tetapi tersedia menjadi pilihan yang lebih baik.

Penyelesaian konflik itu penting, tetapi idealnya, kita akan mencegah konflik sejak awal. Mari kita jelajahi beberapa praktik terbaik untuk mencapai tujuan ini.

Praktik Terbaik Untuk Pencegahan Konflik

Dengan semua strategi penyelesaian konflik yang telah kita kuasai, berikut adalah beberapa praktik terbaik untuk mencegah konflik dalam sistem multi-agen.

Analisis Konflik Pada Tahap Desain

Analisis tumpang tindih kemampuan selama desain sistem untuk mengidentifikasi potensi titik konflik sebelum penerapan. Buat matriks kemampuan yang menunjukkan agen mana yang dapat menangani jenis tugas apa, lalu cari tumpang tindih yang dapat menyebabkan konflik. Tumpang tindih yang tinggi antara agen dengan tingkat prioritas serupa menunjukkan area yang memerlukan protokol koordinasi yang cermat.

Analisis ini harus menghasilkan dokumentasi yang jelas tentang batasan agen dan protokol interaksi, membantu pengembang memahami bagaimana agen yang berbeda harus berkolaborasi daripada bersaing.

Pendeteksian Konflik Saat Waktu Eksekusi

Pantau operasi aktif untuk mendeteksi konflik saat terjadi. Lacak agen mana yang saat ini mengakses sumber daya mana, dan tandai potensi konflik ketika beberapa agen mencoba memodifikasi data yang sama atau menjalankan operasi yang bersaing. Deteksi dini memungkinkan penyelesaian konflik yang baik daripada kegagalan sistem.

Deteksi konflik harus beroperasi terus menerus di latar belakang, menjaga kesadaran akan keadaan sistem tanpa berdampak signifikan pada kinerja. Peringatan otomatis dapat memberi tahu administrator tentang pola konflik berulang yang mungkin menunjukkan masalah desain yang memerlukan perhatian.

Jalur Eskalasi Yang Jelas

Tentukan prosedur eskalasi terstruktur untuk konflik yang tidak dapat diselesaikan secara otomatis. Konflik teknis dapat dieskalasi ke agen pemimpin teknis, konflik bisnis ke agen manajer bisnis, dan konflik sumber daya ke koordinator khusus. Konflik yang tidak dapat diselesaikan pada akhirnya harus dieskalasi ke pengawas manusia.

Jalur eskalasi harus mencakup batas waktu dan prosedur cadangan otomatis untuk mencegah kelumpuhan sistem ketika agen tingkat yang lebih tinggi tidak tersedia. Hierarki eskalasi harus didokumentasikan dengan baik dan diuji secara teratur untuk memastikan fungsinya benar di bawah tekanan.

Resolusi konflik yang efektif memastikan sistem multi-agen tetap andal dan dapat diprediksi saat berkembang, mempertahankan manfaat spesialisasi tanpa kekacauan perilaku yang tidak terkoordinasi. Desain arsitektur yang tepat dapat mencegah sebagian besar konflik sebelum terjadi, menjadikan strategi resolusi sebagai jaring pengaman daripada mekanisme utama.

8.6 RINGKASAN

Dalam bab ini, kita telah mengeksplorasi prinsip-prinsip dasar dan pendekatan praktis untuk merancang sistem AI multi-agen yang efektif. Kita telah membahas strategi dekomposisi agen, pola orkestrasi, desain perintah sistem, kerangka kerja penugasan alat, rekayasa konteks, dan resolusi konflik. Wawasan kuncinya adalah bahwa sistem multi-agen yang efektif memprioritaskan desain arsitektur yang jelas daripada mekanisme resolusi yang kompleks; batasan agen yang terdefinisi dengan baik, matriks kemampuan yang eksplisit, dan manajemen konteks yang tepat mencegah sebagian besar konflik sebelum terjadi.

Kami mendemonstrasikan konsep-konsep ini melalui integrasi AI-6 + A2A, menunjukkan bagaimana konfigurasi sederhana dapat menciptakan sistem multi-agen hierarkis dengan pemisahan peran yang jelas, komunikasi waktu nyata, dan koordinasi tugas yang kuat. Uji ujung-ke-ujung memvalidasi semua pola utama, termasuk penanganan respons langsung, pemrosesan latar belakang, kemampuan multi-tasking, dan penghindaran konflik yang tepat. Pola hierarkis dengan spesialisasi domain muncul sebagai pola yang sangat efektif untuk skenario perusahaan yang membutuhkan penalaran umum dan keahlian mendalam.

Fondasi ini mempersiapkan kita untuk bab selanjutnya, di mana kita akan membangun sistem multi-agen komprehensif yang menggabungkan protokol AI-6, MCP, dan A2A untuk menciptakan solusi siap produksi yang mendemonstrasikan semua pola dan prinsip yang dibahas di sini.

BAB 9
MENERAPKAN SISTEM MULTI-AGEN DENGAN A2A

9.1 PENDAHULUAN

Praktik DevOps telah merevolusi pengembangan dan operasi perangkat lunak dengan memecah silo dan memungkinkan integrasi, penyebaran, dan pemantauan berkelanjutan. Namun, seiring sistem menjadi lebih kompleks dan terdistribusi, pendekatan DevOps tradisional menghadapi tantangan skalabilitas. Munculnya sistem AI multi-agen menghadirkan peluang untuk mentransformasi tim DevOps dengan memperkenalkan agen AI otonom dan khusus yang dapat bekerja bersama operator manusia untuk mengelola infrastruktur, mendeteksi masalah, dan mengatur upaya perbaikan.

Berdasarkan landasan protokol **Agent-to-Agent** (A2A) yang dibahas dalam bab sebelumnya, bab ini mengeksplorasi pembangunan tim DevOps berbasis AI menggunakan sistem multi-agen. Kita akan memeriksa bagaimana menguraikan tanggung jawab DevOps tradisional menjadi peran agen khusus, menerapkan alur kerja pemantauan dan perbaikan otonom, dan menciptakan pola kolaborasi manusia-AI yang mempertahankan kendali operasional sambil memanfaatkan kemampuan AI. Kita akan membangun sistem **Kubernetes DevOps multi-agen** (MAKDO) lengkap yang dapat mengelola klaster Kubernetes secara otonom sekaligus terintegrasi secara mulus dengan alur kerja manusia yang ada.

Topik-topik utama berikut akan dibahas:

- Membangun tim DevOps berbasis AI
- Mendefinisikan peran dan tanggung jawab untuk agen khusus
- Mengimplementasikan agen
- Mengatur tim dengan agen manajer
- Mengintegrasikan umpan balik manusia dan saluran kontrol
- MAKDO dalam aksi – demonstrasi lengkap

9.2 MEMBANGUN TIM DEVOPS BERBASIS AI

Sistem DevOps multi-agen mewakili pergeseran paradigma dari operasi manual yang reaktif ke manajemen infrastruktur otonom yang proaktif. Dengan mendistribusikan tanggung jawab di antara agen-agen khusus dengan keahlian domain dalam pemantauan, analisis, remediasi, atau komunikasi, kita dapat menciptakan sistem yang beroperasi 24/7, berskala untuk mengelola ratusan klaster, dan memberikan respons yang konsisten dan berulang terhadap masalah operasional. Kuncinya adalah merancang sistem ini untuk menambah, bukan menggantikan, keahlian manusia, menciptakan alur kerja kolaboratif yang memanfaatkan kekuatan agen AI dan operator manusia.

Kita akan membangun sistem MAKDO menggunakan kerangka kerja AI-6 dengan integrasi A2A dan alat **Model Context Protocol** (MCP), mendemonstrasikan pola praktis untuk spesialisasi agen, orkestrasi, dan integrasi manusia. Anda akan dibekali dengan pengetahuan

untuk merancang dan mengimplementasikan sistem AI berbasis agen yang dapat mengubah pendekatan organisasi Anda terhadap manajemen infrastruktur.

Apa Yang Membuat DevOps Cocok Untuk AI Multi-Agen?

DevOps adalah domain yang tepat untuk sistem AI multi-agen. Ini semua tentang kekacauan terdistribusi dari infrastruktur modern, klaster Kubernetes, basis data, dan jaringan yang mencakup cloud, dan kebutuhan akan respons cepat dan khusus terhadap kegagalan yang tak terhindarkan.

Saya telah melihat banyak tim tenggelam dalam peringatan dari berbagai sistem, terus-menerus berada di bawah tekanan dan tidak punya waktu untuk mengatasi masalah sistemik. Agen AI dapat memperbaiki ini dengan mengerahkan pakar domain: satu mengawasi pod Kubernetes, yang lain menyetel kueri basis data, dan yang ketiga mengendus masalah jaringan. Mereka berjalan dekat dengan aksi, memparalelkan triase sehingga waktu henti turun dari menit menjadi milidetik.

Pemecahan masalah mengikuti pola yang saya pelajari dengan susah payah, seperti memeriksa peristiwa, log, dan sumber daya. Tetapi area permukaan sistem terdistribusi skala besar sangat besar, dan manusia membutuhkan waktu terlalu lama untuk mencakup semua hal. Agen dapat menjalankan **prosedur operasi standar** (SOP) secara instan sambil meningkatkan kasus-kasus aneh dengan konteks lengkap, membebaskan **insinyur keandalan situs** (SRE) untuk fokus pada arsitektur alih-alih mengatasi masalah mendesak.

DevOps sudah terbagi menjadi spesialis (administrator basis data, insinyur jaringan, pakar cloud, dan ahli keamanan), sehingga agen mencerminkan hal itu: agen keamanan memburu kerentanan dan agen penskalaan menangani lonjakan lalu lintas. Kerja sama tim manusia + AI memungkinkan tim untuk mempertahankan sistem yang berkualitas tinggi, berkinerja tinggi, dan aman. Agen AI menangani tugas-tugas rutin, sementara manusia menangani ancaman dan masalah baru. Ini juga memberi kebebasan kepada insinyur manusia untuk meningkatkan dan membangun otomatisasi dan pengamanan yang lebih baik sehingga agen AI dapat memikul lebih banyak pekerjaan dengan andal.

Namun, menciptakan tim DevOps berbasis AI yang efektif membutuhkan pertimbangan yang cermat tentang peran, tanggung jawab, dan pola interaksi agen. Mari kita jelajahi cara mendesain dan mengimplementasikan sistem DevOps multi-agen menggunakan MAKDO sebagai contoh praktis kita. MAKDO mewujudkan prinsip-prinsip yang telah kita bahas di bab-bab sebelumnya dan menunjukkan bagaimana prinsip-prinsip tersebut diterapkan pada domain DevOps. Mari kita langsung masuk dan mengenal MAKDO.

MAKDO – Sistem DevOps Kubernetes Multi-Agen

MAKDO menunjukkan bagaimana menerjemahkan peran DevOps tradisional ke dalam sistem AI multi-agen yang terkoordinasi. Mari kita periksa arsitektur dan desain agennya.

Arsitektur Sistem

MAKDO terdiri dari empat agen khusus yang bekerja sama untuk menyediakan manajemen kluster Kubernetes otonom dengan pengawasan manusia:

- **Agen Koordinator**: Mengatur alur kerja keseluruhan, membuat keputusan tingkat tinggi, dan berkoordinasi antar agen

- **Agen Penganalisis**: Mengkhususkan diri dalam penilaian kesehatan kluster dan identifikasi masalah menggunakan integrasi k8s-ai
- **Agen Perbaikan**: Menangani modifikasi kluster yang aman dan remediasi otomatis
- **Agen Slack**: Mengelola komunikasi manusia dan alur kerja pemberitahuan

Arsitektur ini mencerminkan pembagian tanggung jawab alami dalam tim DevOps sambil memungkinkan operasi otonom dan pengawasan manusia.

Agen Koordinator – Mengatur Operasi

Agen koordinator bertindak sebagai pemimpin tim, mengatur respons terhadap masalah dan mendelegasikan tugas kepada agen khusus. Dalam arsitektur multi-agen AI-6, koordinator memiliki akses ke sub-agen sebagai alat, dan LLM memutuskan kapan harus memanggilnya berdasarkan situasi. Karena input ke sub-agen adalah perintahnya, koordinator perlu menjadi insinyur perintah yang baik.

MAKDO menggunakan satu file konfigurasi di mana sub-agen didefinisikan **secara langsung** dalam konfigurasi koordinator induk. Pola konfigurasi hierarkis ini berarti AI-6 secara otomatis menemukan dan menginstansiasi semua agen dari satu file. Berikut adalah versi pseudo yang disederhanakan dari file tersebut yang menunjukkan struktur kuncinya. File lengkap tersedia di https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/tree/main/ch09/makdo/src/makdo/agents/coordinator.yaml:

```yaml
# Coordinator configuration (parent agent)
name: "MAKDO Coordinator"
default_model_id: "gpt-4o"
tools_dirs: [ "/path/to/ai-six/tools/memory" ]
system_prompt: |
  You are the MAKDO Coordinator orchestrating specialized agents.
  Delegate to: agent_MAKDO_Analyzer, agent_MAKDO_Fixer, agent_MAKDO_Slack_Bot

# Sub-agents embedded inline (AI-6 auto-instantiates these as AgentTools)
agents:
  # Analyzer sub-agent: Has A2A access to k8s-ai for diagnostics
  - name: "MAKDO_Analyzer"
    description: "Cluster health assessment agent"
    default_model_id: "gpt-4o"
    enable_memory: false
    a2a_servers:
      - name: "kind-makdo-test"
        url: "http://localhost:9999"
    system_prompt: |
      Analyze cluster health using k8s-ai A2A diagnostic skills.
      Report findings with full pod names, errors, and recommendations.

  # Fixer sub-agent: Has A2A access to k8s-ai for remediation
  - name: "MAKDO_Fixer"
    description: "Safe cluster modification agent"
    default_model_id: "gpt-4o"
    enable_memory: false
    a2a_servers:
      - name: "kind-makdo-test"
        url: "http://localhost:9999"
    system_prompt: |
      Execute safe remediation actions using k8s-ai A2A skills.
```

```
    Require approval for destructive operations (delete, scale down).

# Slack_Bot sub-agent: Has MCP access to local Slack server
- name: "MAKDO_Slack_Bot"
  description: "User communication agent"
  default_model_id: "gpt-4o"
  mcp_tools_dirs: [ "src/makdo/mcp_tools" ]
  system_prompt: |
    Post notifications to #makdo-devops
    Handle approval workflows for critical operations.
```

Berikut adalah fitur-fitur utama dari file konfigurasi MAKDO:

1. **Konfigurasi induk**: Agen Koordinator didefinisikan di tingkat atas dengan nama, model, dan prompt sistemnya sendiri.
2. **Sub-agen sebaris**: Ketiga sub-agen tersebut tertanam di bawah kunci `agents:` dalam konfigurasi induk.
3. **Akses alat per agen**:
   - **Analyze**r: Mendapatkan a2a_servers untuk diagnostik k8s-ai
   - **Fixer**: Mendapatkan a2a_servers untuk remediasi k8s-ai
   - **Slack_Bot**: Mendapatkan mcp_tools_dirs untuk komunikasi Slack
4. **Keajaiban AI-6**: Saat Anda memuat file tunggal ini, AI-6 secara otomatis melakukan hal berikut:
   - Membuat empat instance agen (satu koordinator dan tiga sub-agen)
   - Membungkus sub-agen sebagai instance AgentTool
   - Membuatnya tersedia untuk Koordinator sebagai `agent_MAKDO_Analyzer`, `agent_MAKDO_Fixer`, dan `agent_MAKDO_Slack_Bot`
5. **Petunjuk sistem terperinci**: Setiap agen memiliki instruksi komprehensif yang mendefinisikan tanggung jawab, format laporan, dan pedoman operasional.

File YAML tunggal ini mendefinisikan seluruh sistem multi-agen. Koordinator kini dapat mengatur alur kerja yang kompleks dengan memanggil sub-agen sesuai kebutuhan, dengan LLM (Local Language Manager) memutuskan kapan harus memanggil agen mana berdasarkan situasi yang ada.

Perhatikan bagaimana setiap sub-agen dikhususkan melalui tiga mekanisme utama:

- **Akses alat**: Analyzer dan Fixer mendapatkan alat A2A, Slack_Bot mendapatkan alat MCP
- **Perintah sistem**: Instruksi terperinci mengkodekan keahlian domain dan prosedur operasional
- **Format laporan**: Templat keluaran terstruktur memastikan laporan yang konsisten dan dapat ditindaklanjuti

Perintah sistem Analyzer mencakup pedoman prioritas masalah terperinci dan format laporan. Perintah Fixer mendefinisikan protokol keselamatan dan persyaratan persetujuan. Sementara itu perintah `Slack_Bot` menentukan format pesan dan pemrosesan perintah.

Spesialisasi berbasis perintah ini lebih fleksibel daripada pendekatan berbasis kode karena alasan berikut:

- Perilaku dapat disesuaikan tanpa perubahan kode
- Keahlian domain dapat dibaca dan diaudit
- Kemampuan baru dapat ditambahkan melalui rekayasa perintah
- LLM menerapkan penalaran untuk menangani kasus-kasus ekstrem

Mari kita lihat bagaimana agen-agen ini berinteraksi melalui arsitektur multi-agen AI-6.

Pola Interaksi Agen

MAKDO menunjukkan bagaimana arsitektur multi-agen AI-6 memungkinkan operasi terkoordinasi melalui delegasi berbasis alat. Memahami pola ini sangat penting untuk memahami bagaimana agen MAKDO bekerja sama.

Koordinasi Agen Yang Digerakkan Oleh LLM

Dalam AI-6, sub-agen diekspos sebagai alat ke agen induk. Jika Anda perhatikan dengan saksama, sebenarnya hanya ada satu agen yang berinteraksi dengan sistem AI. Fakta bahwa agen akar ini memiliki sub-agen hanyalah detail internal. LLM yang menggerakkan agen akar hanya memutuskan pada setiap giliran percakapan apakah perlu memanggil beberapa alat atau apakah sudah memiliki cukup informasi. Fakta bahwa beberapa alat yang dimilikinya adalah alat pintar, yang merupakan agen AI-6 atau sistem AI yang sepenuhnya terpisah melalui protokol A2A, tidaklah penting.

Tidak ada kelas agen Koordinator yang memanggil metode kelas Penganalisis atau Pembenah. Bahkan tidak ada kelas seperti itu sama sekali. Logika koordinasi ditentukan dalam bahasa alami dalam perintah sistem dari berbagai agen, dan penalaran LLM memutuskan kapan harus memanggil agen mana sebagai alat. Ini adalah contoh dari logika pengikatan terlambat (late-binding logic).

AI-6 mengimplementasikan ini melalui kelas AgentTool, yang membungkus sub-agen agar dapat dipanggil sebagai alat. AgentTool mengimplementasikan antarmuka Tool standar, menerima pesan bahasa alami dan mengembalikan respons lengkap sub-agen. Dari perspektif agen induk, memanggil sub-agen tampak identik dengan memanggil alat lainnya. Untuk penjelasan rinci tentang implementasi AgentTool, lihat bab 8. Ketika agen Koordinator perlu menganalisis klaster, LLM menghasilkan panggilan alat seperti ini:

```
{
  "name": "agent_MAKDO_Analyzer",
  "arguments": {
    "message": "Analyze the health of the kind-makdo-test cluster. Focus on pod
failures and provide specific remediation recommendations."
  }
}
```

Kelas AgentTool membungkus seluruh interaksi sub-agen. Kelas ini mengirimkan pesan ke agen Analyzer, menunggu respons lengkapnya (termasuk panggilan alat apa pun yang dilakukan Analyzer ke k8s-ai), dan mengembalikan hasil akhir ke Koordinator. Pola ini memberikan beberapa manfaat:

- **Delegasi alami**: LLM memutuskan kapan keahlian dibutuhkan, seperti halnya koordinator manusia

- **Enkapsulasi**: Setiap agen mengelola riwayat percakapan dan penggunaan alatnya sendiri
- **Komposisi**: Agen dapat disarang sedalam mungkin
- **Pelestarian konteks**: Koordinator tidak perlu melacak detail implementasi sub-agen

AI-6 hadir dengan beberapa alat bawaan, tetapi dapat menyediakan segalanya untuk setiap sistem AI. Ini bukan masalah karena MAKDO dapat memperluas AI-6 dengan alat tambahan.

Memperluas AI-6 Dengan Alat Baru

MAKDU berinteraksi dengan tim manusia melalui Slack. AI-6 tidak menyediakan alat Slack bawaan (meskipun memiliki antarmuka Slack). Namun, AI-6 dapat diperluas dengan alat tambahan yang mendukung antarmuka alatnya sendiri atau standar MCP, jadi kami mengimplementasikan server MCP kustom yang mengekspos operasi Slack sebagai alat.

Agen Slack_Bot menggunakan server MCP kustom yang dibangun dengan FastMCP untuk menangani semua komunikasi Slack. Server ini terletak di https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/tree/main/ch09/makdo/src/makdo/ mcp_tools/slack.py

Server MCP Slack didasarkan pada implementasi FastMCP dari SDK Python MCP resmi. Fitur-fitur utamanya adalah sebagai berikut:

- **Alat berbasis dekorator**: Dekorator @mcp.tool() secara otomatis mendaftarkan fungsi sebagai alat MCP
- **Manajemen token**: Membaca `AI6_BOT_TOKEN` dari file lingkungan, atau .env
- **Panggilan HTTP sederhana**: Menggunakan pustaka requests untuk memanggil API Slack secara langsung
- **Penanganan kesalahan**: Mengembalikan pesan sukses/gagal yang dapat dipahami oleh LLM
- **Normalisasi saluran**: Secara otomatis menambahkan awalan # jika hilang

Ini jauh lebih sederhana daripada mengimplementasikan server MCP JSON-RPC lengkap secara manual. FastMCP menangani semua detail protokol, memungkinkan Anda untuk fokus pada fungsionalitas alat yang sebenarnya.

Ketika AI-6 memuat server MCP ini, agen Slack_Bot secara otomatis mendapatkan `slack_post_message` dan `slack_list_channels` sebagai alat yang tersedia. LLM kemudian dapat memanggil alat-alat ini secara alami:

```
{
  "name": "slack_post_message",
  "arguments": {
    "channel": "makdo-devops",
    "text": "🚨 CRITICAL: 3 pods failing in kind-makdo-test cluster"
  }
}
```

Mari kita lihat bagaimana AI-6 menemukan alat-alat baru, seperti agen Slack_Bot yang disediakan oleh MAKDO.

Penemuan Dan Penugasan Alat

AI-6 secara otomatis menemukan alat dari berbagai sumber dan menugaskannya ke agen berdasarkan konfigurasi. Modul tool_manager menangani proses ini melalui fungsi `get_tool_dict()`. Seperti yang Anda lihat, modul ini menggabungkan alat dari berbagai direktori dan server, kemudian menyaringnya berdasarkan daftar yang diaktifkan/dinonaktifkan:

```
def get_tool_dict(
    tool_config: ToolConfig, agent_configs: list[Config] = None
) -> dict[str, Tool]:
    """Get a dictionary of all available tools from various sources."""
    tools: list[Tool] = []

    # 1. Discover AI-6 native tools from all directories
    for tools_dir in tool_config.tools_dirs:
        native_tools = _discover_native_tools(tools_dir)
        tools.extend(native_tools)

    # 2. Discover local MCP tools from all directories
    for mcp_tools_dir in tool_config.mcp_tools_dirs:
        local_mcp_tools = _discover_local_mcp_tools(mcp_tools_dir)
        tools.extend(local_mcp_tools)

    # 3. Get tools from A2A servers
    if tool_config.a2a_servers:
        a2a_tools = _get_a2a_tools(tool_config.a2a_servers)
        tools.extend(a2a_tools)

    # 4. Create agent tools if agent configs provided
    if agent_configs:
        agent_tools = _create_agent_tools(agent_configs)
        tools.extend(agent_tools)

    # 5. Filter tools based on enabled/disabled configuration
    tools = _filter_tools(tools, tool_config.enabled_tools,
tool_config.disabled_tools)

    return {tool.name: tool for tool in tools}
```

Berikut adalah rincian alat yang dimiliki setiap agen di MAKDO:

- Agen Koordinator memiliki hal-hal berikut:
  - Alat memori
  - Alat agen: `agent_MAKDO_Analyzer`, `agent_MAKDO_Fixer`, dan `agent_MAKDO_Slack_Bot`
- Analyzer dan Fixer masing-masing mendapatkan hal-hal berikut:
  - Alat A2A dari server k8s-ai (alat `kind-makdo-test_*` untuk operasi kubectl)
  - Alat memori
- Slack_Bot mendapatkan hal-hal berikut:
  - Alat MCP dari `src/makdo/mcp_tools/slack.py` (`slack_post_message` dan `slack_list_channels`)
  - Alat memori (diwariskan)

Berbagi Konteks Dan Manajemen Status

Dalam arsitektur multi-agen AI-6, tidak ada berbagi konteks langsung antar agen. Setiap agen AI-6 memiliki model dan konteksnya sendiri. Ia dapat berkomunikasi dengan agen lain hanya melalui panggilan dan respons alat. Berikut beberapa pendekatan umum untuk berbagi konteks:

- **Pengiriman pesan**: Ketika agen Koordinator memanggil sub-agen, ia menyertakan konteks yang relevan dalam pesan. Berikut contohnya:

```
{
  "name": "agent_MAKDO_Fixer",
  "arguments": {
    "message": "Execute remediation for kind-makdo-test cluster. The
Analyzer identified 3 failing pods due to image pull
errors. Please attempt to restart the affected pods and verify they come up
successfully."
  }
}
```

- **Isolasi sesi**: Setiap agen mempertahankan sesinya sendiri dengan riwayat percakapan. Lihat https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/tree/main/ch08/ai-six/py/ai_six/agent/agent.py:

```
self.session_manager = SessionManager(config.memory_dir)
self.session = self._create_new_session(config.memory_dir)
```

- **Nilai kembalian**: Sub-agen mengembalikan respons lengkap yang kemudian diintegrasikan oleh agen Koordinator ke dalam konteksnya sendiri:

```
Coordinator Session:
- User: "Check cluster health"
- Assistant: [calls agent_MAKDO_Analyzer tool]
- Tool Result: "Analysis complete. Found 3 failing pods..."
- Assistant: "Based on analysis, we have critical issues. Let me notify the
team..."
```

Pendekatan berbasis sesi ini memastikan isolasi konteks sekaligus memungkinkan operasi terkoordinasi melalui pengiriman pesan eksplisit. Riwayat percakapan setiap agen tetap independen, mencegah polusi konteks dan memungkinkan agen untuk dipanggil beberapa kali dengan konteks yang berbeda.

Sekarang setelah kita membahas berbagi konteks dan manajemen status, mari kita lihat peran dan tanggung jawab setiap agen MAKDO.

9.3 MENDEFINISIKAN PERAN DAN TANGGUNG JAWAB UNTUK AGEN KHUSUS

Konfigurasi coordinator.yaml yang ditunjukkan sebelumnya mendefinisikan arsitektur empat agen MAKDO. Bagian ini membahas rasionalitas desain: mengapa batasan peran spesifik ini dipilih, kompromi apa yang dibuat, dan bagaimana arsitektur ini dapat diperluas.

Dari Alur Kerja DevOps Ke Arsitektur Agen

Desain agen MAKDO muncul dari analisis alur kerja respons insiden Kubernetes yang umum. Berikut yang terjadi dalam proses manual tradisional:

1. **Pemicu peringatan**: Operator menerima pemberitahuan
2. **Diagnosis**: Operator menjalankan perintah kubectl untuk menyelidiki
3. **Keputusan**: Operator menentukan perbaikan yang tepat
4. **Remediasi**: Operator menerapkan perubahan (dengan hati-hati)
5. **Komunikasi**: Operator memperbarui tim di Slack
6. **Verifikasi**: Operator mengkonfirmasi bahwa perbaikan berhasil

Ini diterjemahkan langsung ke peran agen MAKDO:

- **Deteksi dan diagnosis** dilakukan oleh agen Analyzer, yang menggunakan k8s-ai untuk menyelidiki status klaster, mengkategorikan masalah, dan merekomendasikan perbaikan
- **Remediasi** dilakukan oleh agen Fixer, yang mengeksekusi perubahan yang aman dan memerlukan persetujuan untuk operasi yang merusak
- **Komunikasi** dilakukan oleh agen Slack_Bot, yang memposting pembaruan dan menangani alur kerja persetujuan
- **Koordinasi** dilakukan oleh agen Coordinator, yang mengatur alur kerja dan membuat keputusan tingkat tinggi

Pemetaan ini mempertahankan struktur alur kerja alami sambil mendistribusikan tanggung jawab di antara agen khusus.

Mengapa Empat Agen, Bukan Satu?

Seperti yang ditunjukkan oleh sistem k8s-ai sederhana asli dari bab 3, satu agen dapat melakukan semua hal sebelumnya (kecuali komunikasi Slack). Jadi, mengapa membagi MAKDO menjadi empat agen? Ada beberapa alasan:

- **Isolasi alat**: Analyzer seharusnya tidak memiliki akses tulis ke klaster (diagnostik hanya baca). Memisahkan Analyzer/Fixer menegakkan hal ini melalui konfigurasi alat. Analyzer dapat melakukan pekerjaannya hanya menggunakan alat hanya baca, seperti get pods dan get deployments. Fixer perlu memodifikasi konfigurasi, jadi tepat untuk memberikan akses ke alat seperti apply manifest.
- **Kejelasan prompt**: Setiap agen memiliki prompt sistem yang terfokus (~50 baris) dibandingkan dengan satu prompt besar (~200 baris) yang mencoba mencakup semuanya.
- **Pengujian independen**: Ini dapat menguji akurasi diagnostik (Analyzer) secara terpisah dari keamanan remediasi (Fixer).
- **Evolusi paralel**: Ini dapat meningkatkan kemampuan diagnostik tanpa risiko perubahan pada logika remediasi.

Mari kita telusuri lebih dalam, khususnya mengenai pemisahan antara analyzer dan fixer.

Mengapa Memisahkan Penganalisis Dan Penanggung Jawab Perbaikan?

Ini adalah keputusan desain yang paling banyak diperdebatkan. Kedua agen menggunakan alat A2A k8s-ai yang sama, jadi mengapa tidak menggabungkannya? Ada tiga alasan utama:

- **Keamanan melalui pemisahan peran**:
  - Perintah penganalisis: "JANGAN PERNAH membuat perubahan, hanya mendiagnosis."
  - Memungkinkan keamanan yang kuat dengan memberikan kredensial baca-saja kepada penganalisis (selain mengandalkan perintah) di masa mendatang
  - Perintah penanggung jawab perbaikan: "Operasi yang memerlukan persetujuan: hapus, kurangi skala, mulai ulang…" Model mental yang jelas: Analisis aman, perbaikan memerlukan kehati-hatian
- **Mode kegagalan yang berbeda**:
  - **Kegagalan penganalisis**: Penganalisis dapat menghasilkan diagnosis yang tidak lengkap atau salah, tetapi tidak pernah mengubah status klaster, sehingga analisis yang buruk tidak berdampak langsung pada produksi.
  - **Kegagalan penanggung jawab perbaikan**: Penanggung jawab perbaikan dapat menjalankan perbaikan yang salah dan langsung memodifikasi sumber daya yang sedang berjalan, sehingga kesalahan di sini dapat menyebabkan pemadaman atau kehilangan data jika pengamannya lemah.
  - Agen terpisah memungkinkan Anda untuk menyesuaikan keandalan secara independen dari otoritas tindakan. Misalnya, jalankan penganalisis yang cepat tetapi akurasinya lebih rendah (recall 90%) untuk triase sambil membatasi perbaikan berisiko tinggi hingga akurasi 99,9% dengan persetujuan manusia.
- **Pola kontrol akses dunia nyata**: Banyak organisasi memberikan tim DevOps akses baca saja ke produksi untuk menghilangkan risiko kerusakan lingkungan produksi selama pengembangan. Hanya SRE atau insinyur yang siaga yang memiliki akses tulis jika perbaikan cepat atau tindakan darurat diperlukan. Pemisahan MAKDO mencerminkan pola akses ini.

Kekurangannya adalah biaya koordinasi yang lebih tinggi, tetapi itu sepadan dengan keamanan dan kejelasannya.

Mengapa Memisahkan SLACK_BOT?

Agen Koordinator dapat memposting langsung ke Slack. Tetapi logika komunikasi dapat menjadi jauh lebih rumit dengan banyak saluran komunikasi, alur kerja persetujuan, aturan pemformatan, dan sebagainya. Slack_Bot dapat berkembang secara independen dan merangkum kompleksitas ini, sementara agen Koordinator berfokus pada koordinasi.

Batasan Akses Alat

Konfigurasi alat setiap agen menyediakan alat yang tepat yang dibutuhkan untuk menjalankan tugasnya. Hal ini membuat agen lebih mudah fokus pada peran mereka dan mengurangi kemungkinan penyalahgunaan alat karena area permukaannya lebih kecil:

```
# Coordinator: Pure orchestrator, no operational tools beyond sub-agents
tools_dirs: [ "/path/to/memory" ]  # Only memory for state tracking

# Analyzer/Fixer: Kubernetes access via A2A
a2a_servers: [ { "url": "http://localhost:9999" } ]  # Same tools, different
prompts
```

```
# Slack_Bot: Communication tools via MCP
mcp_tools_dirs: [ "src/makdo/mcp_tools" ]  # Only Slack access
```

Model izin berbasis konfigurasi ini juga berarti bahwa mengubah kemampuan agen hanya memerlukan pengeditan YAML, bukan perubahan kode.

Memperluas Arsitektur

Desain MAKDO mendukung perluasan melalui agen khusus tambahan. Di sini, kita akan membahas cara menambahkan kemampuan baru. Mari kita lihat contoh penambahan agen pemindai keamanan. Untuk menambahkan pemindaian kerentanan dan penegakan kebijakan keamanan, ikuti langkah-langkah berikut:

1. Definisikan agen baru di coordinator.yaml:

```
agents:
  - name: "MAKDO_Security_Scanner"
    description: "Security vulnerability and compliance scanning agent"
    a2a_servers:
      - name: "kind-makdo-test"
        url: "http://localhost:9999"
    system_prompt: |
      You are the MAKDO Security Scanner.

      Responsibilities:
      - Scan container images for known vulnerabilities
      - Check pod security contexts
      - Verify network policies are in place
      - Report security findings to Coordinator
      Use k8s-ai tools to inspect cluster resources.
      Report findings in severity-ranked format.
```

2. Perbarui perintah agen Koordinator untuk menyertakan agen baru:

```
Available agents:
  - agent_MAKDO_Analyzer
  - agent_MAKDO_Fixer
  - agent_MAKDO_Slack_Bot
  - agent_MAKDO_Security_Scanner  # New

Workflow: After Analyzer identifies issues, also call Security_Scanner
  to check for security implications before Fixer proceeds.
```

   Perhatikan bahwa kita juga dapat menambahkan pemindai keamanan sebagai sub-agen dari Analyzer jika kita menganggap pemindaian keamanan sebagai sub-area dari analisis klaster.

3. **Tidak diperlukan perubahan kode**: AI-6 secara otomatis menemukan agen baru dan mengeksposnya sebagai alat.

Mari kita lihat beberapa pola penting yang digunakan MAKDO.

Pola Desain Utama

Arsitektur MAKDO menunjukkan beberapa pola yang berlaku untuk sistem multi-agen lainnya:

- **Dekomposisi berbasis alur kerja**: Petakan agen ke tahapan alur kerja, bukan teknologi
- **Izin melalui konfigurasi**: Akses alat menentukan apa yang dapat dilakukan agen
- **Prompt sebagai kebijakan**: Kodekan aturan operasional dalam prompt sistem agen
- **Enkapsulasi protokol**: AI-6 menangani protokol khusus (A2A dan MCP) sehingga Anda tidak perlu melakukannya
- **Ekstensibilitas berdasarkan desain**: Menambahkan kemampuan berarti menambahkan alat dan agen

Bagian selanjutnya akan membahas detail implementasi pembuatan dan penyebaran agen-agen ini menggunakan sistem konfigurasi AI-6.

9.4 MENGIMPLEMENTASIKAN AGEN

Setelah peran agen didefinisikan dan tanggung jawabnya diuraikan dengan jelas, sekarang kita beralih ke implementasi. MAKDO mengikuti pendekatan berbasis konfigurasi di mana seluruh sistem multi-agen didefinisikan melalui file konfigurasi YAML, bukan kode kustom. Pendekatan ini memanfaatkan kerangka kerja agen AI-6 yang canggih untuk secara otomatis menginstansiasi agen, menemukan alat, dan membangun saluran komunikasi.

Implementasi Agen Berbasis Konfigurasi

Sistem multi-agen tradisional seringkali memerlukan kode kustom yang ekstensif untuk menginisialisasi agen, mengkonfigurasi alat mereka, dan membangun komunikasi antar-agen. MAKDO mengambil pendekatan yang berbeda: seluruh sistem didefinisikan dalam satu file konfigurasi koordinator dengan definisi sub-agen sebaris. Arsitektur MAKDO menggunakan satu file konfigurasi utama (coordinator.yaml) yang mendefinisikan hal-hal berikut:

- **Agen koordinator**: Orkestrator tingkat atas dengan prompt sistem dan akses alatnya
- **Semua sub-agen secara inline**: Analyzer, Fixer, dan `Slack_Bot` didefinisikan dalam konfigurasi koordinator
- **Penugasan alat**: Akses alat setiap agen ditentukan melalui `tools_dirs`, `mcp_tools_dirs`, dan `a2a_servers`
- **Protokol komunikasi**: Implisit melalui pola AgentTool AI-6

Pendekatan file tunggal ini memiliki beberapa keunggulan:

- **Penyebaran atomik**: Seluruh konfigurasi sistem berada di satu tempat
- **Hierarki yang jelas**: Hubungan induk-anak eksplisit
- **Tidak ada masalah sinkronisasi file**: Tidak ada risiko file konfigurasi terpisah menjadi tidak sinkron
- **Kontrol versi yang lebih mudah**: Satu file untuk melacak perubahan sistem

Titik masuk utama MAKDO mengimplementasikan loop pemeriksaan kesehatan dan manajemen sesi. Mari kita periksa komponen-komponen kuncinya:

1. **Pemuatan konfigurasi**: `Config.from_file()` AI-6 membaca file coordinator.yaml dan secara otomatis membuat semua agen:

```
from ai_six.agent.agent import Agent
from ai_six.agent.config import Config

def create_coordinator_config() -> Config:
    """Create coordinator agent config with sub-agents."""
    return Config.from_file("src/makdo/agents/coordinator.yaml")
```

Baris tunggal ini memuat agen Koordinator dan menginstansiasi semua sub-agen sebaris sebagai alat agen.

2. **Siklus pemeriksaan kesehatan**: Agen Koordinator berjalan terus menerus, mengirimkan permintaan pemeriksaan kesehatan secara berkala:

```
def start_coordinator(coordinator: Agent, config: Dict[str, Any]):
    """Start the coordinator with health check loop."""
    logger = logging.getLogger("makdo")

    # Get check interval from environment or config (default: 60s)
    check_interval = int(os.getenv("MAKDO_CHECK_INTERVAL",
                                   config.get("monitoring",
    {}).get("check_interval", 60)))

    logger.info(f"Starting health check loop (interval: {check_interval}s)")

    try:
        while True:
            try:
                # Request health check from coordinator
                logger.info("🔍 Initiating cluster health check...")

                prompt = (
                    "Perform a comprehensive health check across all
registered clusters. "
                    "Use the Analyzer agent to identify any issues, then
use the Slack Bot agent "
                    "to report findings to the #makdo-devops channel. "
                    "If critical issues are found, use the Fixer agent to
attempt remediation."
                )

                response = coordinator.send_message(prompt)
                logger.info(f"☑ Health check completed: {response[:200]}...")
```

Loop tersebut mengirimkan perintah bahasa alami yang menjelaskan alur kerja yang diinginkan. LLM mengatur sub-agen berdasarkan perintah ini. Perhatikan bahwa MAKDO beroperasi secara otonom di sini. Tidak ada alur permintaan-respons yang dikendalikan manusia seperti pada antarmuka obrolan. Namun, manusia masih dapat memantau dan melakukan intervensi melalui Slack. Respons dari agen Koordinator hanya dicatat untuk keperluan audit.

3. **Manajemen sesi**: Setelah setiap pemeriksaan kesehatan, MAKDO menghapus riwayat percakapan untuk mencegah kelebihan konteks:

```
# CRITICAL: Clear session history to prevent context overflow
# Keep only system message - each health check cycle is independent
if len(coordinator.session.messages) > 3:
    logger.info(
        f"Clearing old messages ({len(coordinator.session.messages)}) messages"
    )
    system_message = coordinator.session.messages[0]
    if coordinator.session.messages else None
    coordinator.session.messages.clear()
    if system_message:
        coordinator.session.messages.append(system_message)
    logger.info(
        f"Session reset to {len(coordinator.session.messages)} message")
except Exception as e:
    logger.error(f"Error during health check cycle: {e}")

# Wait before next check
time.sleep(check_interval)
```

Hal ini mencegah jendela konteks terisi dengan pemeriksaan kesehatan lama. Setiap siklus diperlakukan secara independen. Jika memori jangka panjang diperlukan, hal itu dapat diimplementasikan melalui penyimpanan eksternal seperti basis data.

4. **Titik masuk utama**: Fungsi utama menghubungkan semuanya:

```
def main():
    """Main entry point - note: synchronous, not async."""load_dotenv()
    config = load_config()
    setup_logging(config)

    logger = logging.getLogger("makdo")
    logger.info("Starting MAKDO - Multi-Agent Kubernetes DevOps System")

    # Create coordinator with inline sub-agents
    coordinator = Agent(create_coordinator_config())
    logger.info("Coordinator and sub-agents created successfully")

    # Start the coordinator (runs health check loop)
    start_coordinator(coordinator, config)

if __name__ == "__main__":
    exit(main())  # Note: synchronous call, not asyncio.run()
```

Kode tersebut terdapat di `src/makdo/main.py`. Fitur utama implementasi ini meliputi:

- **Arsitektur sinkron**: Menggunakan `time.sleep()` alih-alih asyncio untuk menghindari konflik event loop dengan alat MCP
- **Manajemen sesi**: Menghapus pesan lama setelah setiap siklus pemeriksaan kesehatan untuk mencegah luapan konteks (batas token 128k) dan kesalahan pengurutan pesan
- **Loop pemeriksaan kesehatan**: Terus memantau klaster pada interval yang dapat dikonfigurasi

Perhatikan apa yang *tidak* ada dalam kode ini:

- Tidak ada instansiasi agen manual untuk sub-agen
- Tidak ada logika penemuan atau pendaftaran alat
- Tidak ada pengaturan komunikasi antar agen
- Tidak ada konfigurasi klien MCP atau A2A

Semua ini ditangani secara otomatis oleh AI-6 berdasarkan file konfigurasi koordinator. Inilah proposisi nilai dari kerangka kerja AI-6, yang secara andal dan diam-diam menangani detail-detail kompleks atas nama MAKDO.

Penemuan Dan Penugasan Alat

Manajer alat AI-6 menangani penemuan alat otomatis dari berbagai sumber. Untuk MAKDO, berikut cara alat ditugaskan ke setiap agen:

- **Agen koordinator:**

```
# Only memory tools - pure orchestrator
tools_dirs: [ "/Users/gigi/git/ai-six/py/ai_six/tools/memory" ]
mcp_tools_dirs: [ "/Users/gigi/git/ai-six/py/ai_six/mcp_tools" ]
memory_dir: "data/memory"

# NO a2a_servers - delegates to Analyzer/Fixer for k8s operations
# NO Slack tools - delegates to Slack_Bot for communications
```

- **Agen penganalisis (inline):**

```
agents:
  - name: "MAKDO_Analyzer"
    description: "Cluster health assessment agent"
    default_model_id: "gpt-4o"
    # Has A2A access to k8s-ai server
    a2a_servers:
      - name: "kind-makdo-test"
        url: "http://localhost:9999"
        timeout: 30.0
        api_key: "test-key"
```

- **Agen perbaikan (inline):**

```
- name: "MAKDO_Fixer"
  description: "Safe cluster modification agent"
  default_model_id: "gpt-4o"
  # Has A2A access to k8s-ai server
  a2a_servers:
    - name: "kind-makdo-test"
      url: "http://localhost:9999"
      timeout: 30.0
      api_key: "test-key"
```

- **Agen Slack_Bot (inline):**

```
- name: "MAKDO_Slack_Bot"
  description: "User communication agent"
```

```
default_model_id: "gpt-4o"
tools_dirs: []
# Has MCP access to local Slack tools
mcp_tools_dirs: [ "src/makdo/mcp_tools" ]
```

Agen memiliki alat mereka sendiri, tetapi mereka juga memiliki konteks independen mereka sendiri. Inilah prinsip isolasi sesi.

Isolasi Sesi

Setiap agen di MAKDO mempertahankan sesi percakapannya sendiri dengan riwayat pesan yang terisolasi. Ketika agen Koordinator memanggil `agent_MAKDO_Analyzer`, Analyzer tidak melihat riwayat percakapan koordinator. Analyzer memulai dengan konteks baru yang hanya berisi hal-hal berikut:

- Prompt sistemnya sendiri
- Pesan dari koordinator
- Katalog alatnya sendiri

Namun, isolasi sesi ini memiliki implikasi penting. Berikut adalah manfaatnya:

- **Penggunaan token yang berkurang**: Sub-agen tidak membawa konteks percakapan induk secara lengkap
- **Pemrosesan terfokus**: Setiap agen hanya mempertimbangkan tugas spesifiknya dan konteks yang relevan (panggilan dan respons alat)
- **Potensi eksekusi paralel**: Sesi independen dapat berjalan secara bersamaan, terutama untuk agen A2A
- **Pemisahan tanggung jawab yang lebih bersih**: Agen tidak dapat secara tidak sengaja membocorkan informasi antar konteks

Berikut adalah kekurangannya:

- **Penyebaran konteks diperlukan**: Induk harus secara eksplisit menyertakan konteks yang relevan dalam pesan ke sub-agen
- **Tidak ada memori bersama secara default**: Agen tidak secara otomatis berbagi status (meskipun dapat menggunakan alat memori)
- **Peningkatan verbositas pesan**: Induk mungkin perlu mengulang konteks dalam beberapa panggilan sub-agen

Bagi MAKDO, isolasi ini bermanfaat. Ketika agen Koordinator meminta Analyzer untuk menilai kesehatan klaster, Analyzer tidak perlu mengetahui tentang percakapan sebelumnya dengan Fixer atau `Slack_Bot`. Ia hanya fokus pada permintaan penilaian klaster saat ini.

Konfigurasi Di Atas Kode

Pendekatan berbasis konfigurasi berarti Anda dapat sepenuhnya mengubah perilaku MAKDO tanpa menyentuh kode Python:

- **Tambahkan agen baru**: Tambahkan entri ke daftar agen
- **Ubah akses alat**: Ubah `tools_dirs`, `mcp_tools_dirs`, `a2a_servers`, atau agen untuk agen mana pun
- **Sesuaikan perilaku agen**: Perbarui perintah sistem
- **Ganti model LLM**: Ubah `default_model_id`
- **Terhubung ke sistem AI jarak jauh yang berbeda**: Perbarui URL server A2A

Poin Penting

Implementasi MAKDO menunjukkan beberapa pola penting untuk membangun sistem multi-agen:

- **Konfigurasi mendorong arsitektur**: Definisikan struktur sistem Anda dalam YAML, bukan kode
- **Otomatisasi kerangka kerja**: Biarkan AI-6 menangani instansiasi agen, penemuan alat, dan komunikasi
- **Abstraksi alat**: Sub-agen muncul sebagai alat bagi induknya melalui kelas AgentTool
- **Isolasi sesi**: Setiap agen mempertahankan status percakapan independen
- **Prinsip hak akses minimal**: Setiap agen hanya mendapatkan alat yang dibutuhkannya
- **Ekstensibilitas melalui konfigurasi:** kemampuan baru dapat ditambahkan dengan mengedit berkas YAML, bukan dengan menulis kode

Pendekatan ini secara dramatis mengurangi kompleksitas membangun dan memelihara sistem multi-agen. Seluruh implementasi MAKDO kurang dari 150 baris kode Python, dengan sebagian besar berupa pencatatan dan penanganan kesalahan. Implementasi sebenarnya terdapat dalam file konfigurasi, sehingga sistem mudah dipahami, dimodifikasi, dan diperluas. Namun, penting untuk memiliki alat bantu dan validasi skema yang kuat untuk konfigurasi.

9.5 MENGORKESTRASI TIM DENGAN AGEN MANAJER

Agen Koordinator di MAKDO berfungsi sebagai orkestrator pusat, mengelola alur kerja di seluruh agen khusus sambil mempertahankan kesadaran akan status sistem dan prioritas operasional. Tidak seperti sistem orkestrasi tradisional yang bergantung pada definisi alur kerja eksplisit dan mesin status, Koordinator MAKDO menggunakan penalaran LLM untuk secara dinamis mengoordinasikan aktivitas agen berdasarkan situasi.

Prompt sistem agen Koordinator memberikan panduan, tetapi LLM menentukan urutan spesifik pemanggilan agen berdasarkan konteks.

Eksekusi Alur Kerja Dinamis

Mesin alur kerja tradisional memerlukan alur kerja yang telah ditentukan sebelumnya seperti berikut:

```
detect_issue -> analyze_issue -> fix_issue -> notify_humans
```

Koordinator MAKDO justru menerima panduan tingkat tinggi dalam bahasa Inggris pada prompt sistemnya:

```
Your primary responsibilities:
1. Monitor multiple Kubernetes clusters for health and issues
2. Coordinate between the Analyzer, Fixer, and Slack agents
3. Make decisions on task prioritization and escalation
4. Maintain awareness of ongoing operations across all clusters
```

LLM kemudian menentukan urutan pemanggilan agen yang sesuai berdasarkan situasi. Mari kita jelajahi beberapa contoh berbasis skenario:

```
Scenario 1: Periodic health check request
MAKDO: Invoke coordinator every minute asking to "Check the health of kind-makdo-test cluster"
Coordinator:
1. Calls the MAKDO_Analyzer agent with message "Analyze kind-makdo-test cluster health"
2. Receives detailed report
3. Calls MAKDO_Slack_Bot agent to post results

Scenario 2: Critical issue detected
User: "Production cluster has failing pods"
Coordinator:
1. Calls agent_MAKDO_Analyzer for detailed diagnostics
2. Evaluates severity from Analyzer report
3. Calls agent_MAKDO_Fixer to remediate issues
4. Calls agent_MAKDO_Slack_Bot to notify team of analysis and remediation results
```

Agen Koordinator menyesuaikan alur kerjanya berdasarkan temuan Analis, tingkat keparahan masalah, dan kebijakan operasional yang dikodekan dalam perintah sistemnya. Jika persetujuan manusia diperlukan, MAKDO akan menunggu persetujuan dari Slack_Bot sebelum memanggil agen Pemecah Masalah.

Titik Keputusan Dan Percabangan

Perintah sistem agen Koordinator mencakup pedoman pengambilan keputusan yang diterapkan oleh LLM:

```
Decision-making guidelines:
- Always assess cluster health before making changes
- Prioritize critical issues (pods failing, services down) over warnings
- Require human approval for destructive operations (delete, scale down)
- Coordinate operations to avoid conflicts between clusters
- Escalate to humans when automated fixes fail or are uncertain
- ALWAYS post full detailed results to Slack for human visibility
```

Pedoman ini menciptakan titik keputusan di mana LLM memilih jalur yang berbeda. Berikut beberapa contohnya.

Contoh 1: operasi aman versus operasi destruktif

Analyzer melaporkan “3 pod macet di CrashLoopBackOff, mungkin perlu dihapus.” Alasan LLM mencakup hal-hal berikut:

- Penghapusan bersifat destruktif (sesuai pedoman)
- Membutuhkan persetujuan manusia (sesuai pedoman)
- Harus memberi tahu manusia terlebih dahulu (sesuai pedoman)

Jalur eksekusinya adalah sebagai berikut:

```
1. agent_MAKDO_Slack_Bot: "Report analysis and request approval"
2. Wait for human response
3. If approved: agent_MAKDO_Fixer with "delete pods"
4. agent_MAKDO_Slack_Bot: "Report remediation results"
```

Contoh 2: remediasi otomatis versus eskalasi

Analyzer melaporkan "2 pod perlu di-restart, operasi risiko rendah." Alasan LLM adalah sebagai berikut:

- Restart adalah operasi yang aman (tidak termasuk dalam daftar operasi yang merusak)
- Tidak perlu persetujuan (sesuai daftar operasi aman Fixer)
- Dapat langsung dilanjutkan (sesuai pedoman)

Jalur eksekusinya adalah sebagai berikut:

```
1. agent_MAKDO_Fixer: "Restart the 2 identified pods"
2. agent_MAKDO_Slack_Bot: "Report both analysis and remediation results"
```

Pengambilan keputusan dinamis ini lebih fleksibel daripada alur kerja yang dikodekan secara kaku, namun tetap mengikuti kebijakan operasional.

Penanganan Operasi Multi-Langkah Yang Kompleks

Beberapa skenario operasional memerlukan beberapa putaran interaksi agen, pelacakan status, dan logika kondisional. Koordinator MAKDO menangani hal ini melalui konteks percakapan dan penalaran LLM.

Koordinasi Multi-Kluster

Saat mengelola beberapa kluster, agen Koordinator harus mengurutkan operasi untuk menghindari konflik:

```
User: "Check health of all production clusters and fix any issues"
Coordinator execution:
1. For each cluster (production-east, production-west, production-central):
   a. agent_MAKDO_Analyzer: "Analyze [cluster] health"
   b. Evaluate if fixes needed
   c. If needed: agent_MAKDO_Fixer: "Fix issues in [cluster]"
   d. agent_MAKDO_Slack_Bot: "Post results for [cluster]"
2. Aggregate results across clusters
3. agent_MAKDO_Slack_Bot: "Post summary of all cluster operations"
```

Agen Koordinator mempertahankan kesadaran tentang kluster mana yang telah diproses melalui konteks percakapannya. LLM melacak status secara implisit melalui riwayat pesan. Jika itu menjadi terlalu rumit, MAKDO dapat berevolusi untuk memiliki meta-koordinator sebagai agen akar yang meluncurkan beberapa agen koordinator, masing-masing dengan `Analyzer`, `Fixer`, dan `Slack_Bot`-nya sendiri.

Remediasi Bersyarat

Beberapa perbaikan bergantung pada hasil operasi sebelumnya:

```
Scenario: Image pull failures might require different remediation strategies
```

```
Coordinator logic:
1. agent_MAKDO_Analyzer: "Diagnose image pull failure"
2. If Analyzer reports: "Image not found in registry"
   -> agent_MAKDO_Slack_Bot: "Escalate - invalid image reference"
3. If Analyzer reports: "Registry authentication failed"
   -> agent_MAKDO_Fixer: "Apply image pull secrets"
   -> agent_MAKDO_Analyzer: "Verify pods now pulling successfully"
4. If Analyzer reports: "Registry temporarily unavailable"
   -> Schedule retry after delay
   -> agent_MAKDO_Slack_Bot: "Notify team of temporary issue"
```

Logika kondisional ini muncul dari penalaran LLM tentang laporan Analyzer, bukan dari pernyataan if/else eksplisit dalam kode.

Perulangan percobaan ulang dan verifikasi

Agen Koordinator dapat menerapkan logika percobaan ulang ketika operasi gagal:

```
Fixer reports: "Failed to restart pod, still in error state"

Coordinator reasoning:
- Initial fix attempt failed (per Fixer report)
- Guidelines say "escalate when automated fixes fail"
- Should verify current state before escalating

Execution:
1. agent_MAKDO_Analyzer: "Re-analyze the specific failing pod"
2. Evaluate new diagnostic information
3. Decide: Try alternative fix OR escalate to humans
4. agent_MAKDO_Slack_Bot: "Report situation and decision"
```

Upaya pengulangan dapat membantu ketika terjadi masalah sementara, tetapi terkadang sistem mungkin mengalami kegagalan yang terus-menerus, dan sistem perlu mampu menanganinya.

Penanganan Dan Pemulihan Kesalahan

Agen MAKDO dapat mengalami berbagai mode kegagalan. Prompt sistem agen Koordinator mencakup panduan penanganan kesalahan implisit yang diterapkan oleh LLM.

Kegagalan Komunikasi Agen

Jika pemanggilan sub-agen gagal (batas waktu, pengecualian, dan lain-lain.), kelas AgentTool akan mengembalikan pesan kesalahan yang diterima Koordinator sebagai hasil alat:

```
Tool: agent_MAKDO_Analyzer
Error: "Timeout connecting to k8s-ai server"

Coordinator response (guided by system prompt):
1. Recognize communication failure
2. agent_MAKDO_Slack_Bot: "Alert - cannot reach cluster analysis service"
3. Decide: Retry OR escalate OR use cached data
```

LLM menafsirkan kesalahan dan memutuskan strategi pemulihan yang tepat berdasarkan konteks.

Kegagalan Sebagian

Ketika beberapa operasi berhasil sementara yang lain gagal, agen Koordinator melacak kemajuan sebagian:

Fixer reports: "Restarted 5 of 7 failing
pods. 2 pods remain in error state due to resource constraints."

Coordinator response:
1. Recognize partial success (5/7 completed)
2. Analyze remaining issues (resource constraints)
3. agent_MAKDO_Analyzer: "Check cluster resource availability"
4. Based on analysis, decide next steps:
   - Scale up nodes (if resource exhaustion)
   - Adjust pod resource requests (if over-provisioned)
   - Escalate to humans (if capacity limits reached)
5. agent_MAKDO_Slack_Bot: "Report partial success and next steps"

Salah satu kegagalan yang paling rumit adalah kegagalan berantai.

Kegagalan Berantai

Agen Koordinator harus menangani skenario di mana memperbaiki satu masalah mengungkap atau menyebabkan masalah lain:

```
Initial issue: 3 pods failing
After fix attempt: 5 pods now failing

Coordinator reasoning:
- Situation worsened (3 -> 5 pods failing)
- Applied fix may have caused new issues
- Should halt automated remediation
- Requires human investigation

Execution:
1. agent_MAKDO_Slack_Bot: "CRITICAL: Remediation caused additional failures.
Halting automation."
2. agent_MAKDO_Analyzer: "Full cluster diagnostic to identify root cause"
3. agent_MAKDO_Slack_Bot: "Post detailed diagnostic results"
4. Wait for human guidance before proceeding
```

Pendekatan defensif ini mencegah otomatisasi memperburuk situasi.

Manajemen Dan Koordinasi Status

Koordinator MAKDO mempertahankan status operasional melalui konteks percakapan dan alat memori, memungkinkan kesadaran akan operasi yang sedang berlangsung dan konteks historis. Riwayat pesan agen Koordinator berfungsi sebagai memori kerjanya selama satu siklus pemeriksaan kesehatan.

LLM mengakses konteks ini untuk menjawab permintaan atau pertanyaan lanjutan dari pengguna di Slack dan untuk mempertahankan kesadaran akan operasi yang sedang berlangsung sementara penganalisis dan penanggung jawab beroperasi. Perhatikan bahwa koordinator dapat memanggil penganalisis dan penanggung jawab beberapa kali dalam satu siklus pemeriksaan kesehatan.

Alat Memori Untuk Status Jangka Panjang

Untuk status yang perlu dipertahankan di seluruh sesi percakapan, MAKDO menggunakan alat memori AI-6:

```
# Coordinator configuration
tools_dirs: [ "/Users/gigi/git/ai-six/py/ai_six/tools/memory" ]
memory_dir: "data/memory"
The Coordinator agent can save and retrieve structured information:
# Save cluster state after operations
Tool: save_memory
Arguments:
  key: "production-east-last-check"
  value: {
    "timestamp": "2024-10-06T14:30:00Z",
    "status": "degraded",
    "issues": 3,
    "fixed": 2,
    "pending_approval": ["pod-delete-operation-047"]
  }

# Later retrieve state
Tool: retrieve_memory
Arguments:
  key: "production-east-last-check"
Result: [saved state object]
```

Hal ini memungkinkan agen Koordinator untuk mempertahankan kesadaran di berbagai sesi percakapan dan restart agen.

Menghindari Operasi Duplikat

Koordinator dapat menggunakan memori untuk melacak operasi terbaru dan menghindari duplikasi:

```
User: "Fix production-east issues"

Coordinator checks memory:
Tool: retrieve_memory
Arguments:
  key: "recent-operations"
Result: "production-east remediation completed 5 minutes ago"

Coordinator response:
"Production-east was just remediated 5 minutes ago. Let me verify current status
instead."
```

```
Execution:
agent_MAKDO_Analyzer: "Quick health check of production-east"
```

Mari kita bahas berbagai mode orkestrasi.

Orkestrasi Proaktif Versus Reaktif

Koordinator MAKDO dapat beroperasi dalam mode reaktif (menanggapi permintaan) dan proaktif (pemantauan otonom).

Mode Reaktif – Operasi Yang Diinisiasi Pengguna

Dalam mode reaktif, agen Koordinator menanggapi permintaan pengguna secara eksplisit:

```
User: "Check cluster health"
-> Coordinator analyzes and reports

User: "Fix the failing pods"
-> Coordinator remediates and reports

User: "What's the status of production-west?"
-> Coordinator queries state and responds
```

Mode ini memberi manusia kendali langsung atas kapan dan bagaimana operasi terjadi.

Mode Proaktif – Pemantauan Otonom

Dalam mode proaktif, agen Koordinator dapat memulai operasi berdasarkan jadwal atau pemicu. Ini adalah mode operasi MAKDO saat ini:

```
# Periodic scheduled check
Every minutes:
1. agent_MAKDO_Analyzer: "Health check all registered clusters"
2. If issues found:
    a. Categorize by severity (CRITICAL, WARNING, INFO)
    b. For CRITICAL: Immediate remediation workflow
    c. For WARNING: Schedule remediation, notify humans
    d. For INFO: Log and include in next summary report
3. agent_MAKDO_Slack_Bot: "Post status update if changes detected"
```

Mode ini lebih canggih tetapi membutuhkan banyak kepercayaan karena agen Koordinator beroperasi secara otonom.

Pendekatan Hibrida – Koordinasi Berbasis Peristiwa

Pendekatan hibrida menggabungkan pola reaktif dan proaktif:

```
Normal operation:
- Proactive monitoring every 5 minutes
- Reactive response to cluster events or alerts
- Automatic remediation of routine issues
- Slack notifications for significant events

When critical issue detected:
```

```
- Notify humans immediately
- Wait for approval before major operations
- Provide detailed status updates on request
```

Sebagian besar sistem di dunia nyata akan menggunakan pendekatan hibrida. Sekarang, mari kita alihkan perhatian kita ke aspek penting yaitu menambahkan manusia ke dalam proses.

9.6 MENGINTEGRASIKAN UMPAN BALIK MANUSIA DAN SALURAN KONTROL

Meskipun agen AI dapat mengotomatiskan banyak tugas DevOps, pengawasan manusia tetap penting untuk keputusan strategis, pemecahan masalah yang kompleks, dan menjaga kendali operasional. MAKDO mengintegrasikan Slack sebagai antarmuka manusia-AI utamanya, memungkinkan pemberitahuan waktu nyata, pembaruan status, dan alur kerja persetujuan. Perhatikan bahwa karena semua komunikasi melalui agen, mudah untuk mengganti Slack dengan email atau alat komunikasi lain yang memiliki API, serta menggunakan beberapa mekanisme komunikasi secara bersamaan.

Tantangan Antarmuka Manusia-AI

Kolaborasi manusia-AI yang efektif dalam DevOps membutuhkan desain antarmuka yang cermat yang menyeimbangkan efisiensi otomatisasi dengan kendali manusia. Antarmuka harus melakukan hal-hal berikut:

- **Memberikan visibilitas**: Manusia membutuhkan transparansi penuh tentang apa yang dilakukan agen
- **Memungkinkan kontrol**: Manusia harus dapat menyetujui, menolak, atau memodifikasi tindakan agen, serta menghentikan sementara atau membatalkan operasi yang sedang berlangsung
- **Meminimalkan gangguan**: Jangan membebani manusia dengan pemberitahuan rutin
- **Mendukung komunikasi alami**: Gunakan alat dan pola komunikasi yang familiar
- **Mempertahankan konteks**: Pertahankan riwayat percakapan dan konteks operasional

MAKDO mengatasi persyaratan ini melalui agen Slack_Bot khusus yang bertindak sebagai jembatan komunikasi antara sistem AI dan operator manusia.

Arsitektur Agen `Slack_Bot`

Agen Slack_Bot adalah agen komunikasi khusus dengan akses ke Slack melalui alat MCP. Tidak seperti agen Analyzer dan Fixer yang menggunakan A2A untuk operasi Kubernetes, Slack_Bot menggunakan MCP untuk berinteraksi dengan API Slack.

Berdasarkan file coordinator.yaml yang ditunjukkan sebelumnya, konfigurasi Slack_Bot adalah sebagai berikut:

```yaml
agents:
  - name: "MAKDO_Slack_Bot"
    description: "User communication and notification interface agent"
    default_model_id: "gpt-4o"
    tools_dirs: []
    # Slack_Bot uses Local MCP server for Slack communication
    mcp_tools_dirs: [ "src/makdo/mcp_tools" ]
    system_prompt: |
```

```
        You are the MAKDO Slack Bot, the primary interface between the MAKDO
system and human users.

        Your primary responsibilities:
        1. Send alerts and notifications to the designated Slack channel (#makdo-
devops)
        2. Parse and process user commands from Slack messages
        3. Provide status updates and operation summaries
        4. Format technical information for human-readable display
        5. Handle approval requests for critical operations

        Message types and formatting:
        ⏰ CRITICAL: Immediate attention required
        ⚠ WARNING: Issue needs attention
        ℹ INFO: Status update or information
        ☑ SUCCESS: Operation completed successfully
        ✗ FAILED: Operation failed
        🕒 PENDING: Operation in progress
```

Elemen konfigurasi kunci adalah mcp_tools_dirs: ["src/makdo/mcp_tools"], yang memberi tahu AI-6 untuk menemukan server MCP di direktori tersebut.

Membangun Server MCP Kustom Dengan FASTMCP

MAKDO menggunakan server MCP kustom untuk integrasi Slack yang dibangun dengan pustaka FastMCP seperti yang dibahas sebelumnya. Pendekatan ini jauh lebih sederhana daripada mengimplementasikan protokol MCP lengkap secara manual.

Server Slack FASTMCP

Implementasi server MCP Slack menunjukkan bagaimana FastMCP menyederhanakan penanganan protokol. Mari kita periksa komponen-komponen kunci dari src/makdo/mcp_tools/slack.py:

1. **Inisialisasi server dan pemuatan token**: FastMCP hanya membutuhkan satu baris untuk membuat server MCP, dengan manajemen token yang menangani variabel lingkungan dan file .env:

```python
import os
import requests
from pathlib import Path
from mcp.server.fastmcp import FastMCP

# Initialize FastMCP server
mcp = FastMCP("Slack Tools", "")

# Get bot token from environment or .env file
bot_token = os.getenv('AI6_BOT_TOKEN')

if not bot_token:
    # Navigate from .../makdo/src/makdo/mcp_tools/slack.py to .../makdo/.env
    makdo_root = \
        Path(__file__).resolve().parent.parent.parent.parent
    env_file = makdo_root / '.env'

    if env_file.exists():
        with open(env_file) as f:
```

```
            for line in f:
                if line.startswith('AI6_BOT_TOKEN='):
                    bot_token = line.split('=', 1)[1].strip()
                    break
```

Ini adalah alat khusus MAKDO, jadi tidak masalah untuk menyematkan konfigurasi khusus MKADO seperti jalur file .env.

2. **Alat pengiriman pesan**: Dekorator `@mcp.tool()` secara otomatis mendaftarkan fungsi tersebut sebagai alat MCP. FastMCP menangani semua detail protokol:

```
@mcp.tool()
def slack_post_message(channel: str, text: str) -> str:
    """Post a message to a Slack channel"""

    # Normalize channel name (add # prefix if missing)
    if not channel.startswith('#'):
        channel = f'#{channel}'

    # Call Slack API directly with simple requests library
    url = 'https://slack.com/api/chat.postMessage'
    headers = {'Authorization': f'Bearer {bot_token}'}
    data = {'channel': channel, 'text': text}

    response = requests.post(
        url, headers=headers, json=data, timeout=10
    )
    result = response.json()

    # Return LLM-friendly success/error messages
    if result.get('ok'):
        return f"☑ Message posted to {channel}"
    else:
        return f"❌ Failed to post: {result.get('error', 'unknown_error')}"
```

Perhatikan bahwa alat ini berinteraksi dengan API Slack menggunakan panggilan HTTP langsung dengan pustaka requests. Ini berbeda dari frontend Slack AI-6 yang menggunakan SDK Slack resmi dan pustaka slack_bot. Untuk MAKDO, panggilan API langsung lebih sederhana dan memadai tanpa ketergantungan eksternal yang tidak perlu.

3. **Alat pencantuman saluran**: Alat kedua menyediakan penemuan saluran, mengikuti pola decorator yang sama:

```
@mcp.tool()
def slack_list_channels() -> str:
    """List all channels in the workspace"""

    url = 'https://slack.com/api/conversations.list'
    headers = {'Authorization': f'Bearer {bot_token}'}

    response = requests.get(url, headers=headers, timeout=10)
    result = response.json()

    if result.get('ok'):
        channels = result.get('channels', [])
```

```
        return '\n'.join([f"#{ch['name']}" for ch in channels])
    else:
        return f"❌ Failed to list channels: {result.get('error')}"
```

4. **Titik masuk server**: Menjalankan skrip akan memulai server MCP, sehingga kedua alat tersebut tersedia untuk AI-6:

```
if __name__ == '__main__':
    mcp.run()
```

Berdasarkan hal ini, kita melihat bahwa FastMCP menyederhanakan pembuatan server MCP melalui beberapa pilihan desain dan fitur utama:

- **Registrasi berbasis decorator**: Decorator `@mcp.tool()` secara otomatis mendaftarkan fungsi sebagai alat MCP. Tidak diperlukan penanganan JSON-RPC manual.
- **Inferensi tipe otomatis**: FastMCP mengekstrak tipe parameter dan deskripsi dari tanda tangan fungsi dan docstring.
- **Penanganan protokol**: FastMCP mengelola semua detail protokol MCP (inisialisasi, daftar alat, dan eksekusi alat).

Bandingkan ini dengan implementasi MCP manual, yang akan membutuhkan hal-hal berikut:

- Penguraian pesan JSON-RPC
- Negosiasi versi protokol
- Pembuatan skema alat
- Penanganan permintaan/respons
- Serialisasi kesalahan

FastMCP menangani semua ini secara otomatis, memungkinkan Anda untuk fokus pada fungsionalitas alat yang sebenarnya. Server Slack membaca token bot dari variabel lingkungan atau file .env:

```
bot_token = os.getenv('AI6_BOT_TOKEN')

if not bot_token:
    # Try loading from .env file
    env_file = makdo_root / '.env'
    if env_file.exists():
# Parse .env file for AI6_BOT_TOKEN
```

Hal ini memungkinkan penerapan variabel lingkungan yang fleksibel dalam produksi dan file .env untuk pengembangan. Kedua alat tersebut mengembalikan pesan kesalahan yang mudah dipahami oleh LLM:

```
if result.get('ok'):
    return f"☑ Message posted to {channel}"
else:
    error = result.get('error', 'unknown_error')
    return f"❌ Failed to post to {channel}: {error}"
```

Ketika agen `Slack_Bot` menerima respons ini, ia dapat menganalisis kegagalan dan mengambil tindakan yang sesuai (mencoba lagi, meningkatkan masalah, atau menggunakan saluran alternatif).

Tujuan dari alat Slack MCP adalah agar agen `Slack_Bot` dapat berkomunikasi dengan pengguna melalui Slack. Namun MAKDO memiliki beberapa pola komunikasi yang berbeda.

Pola Komunikasi

MAKDO menerapkan beberapa pola komunikasi untuk menjaga agar manusia tetap mendapat informasi sambil menghindari kelebihan informasi.

Hierarki Pemberitahuan

Perintah sistem agen Koordinator menginstruksikannya untuk selalu mengirimkan hasil terperinci ke Slack:

```
CRITICAL WORKFLOW FOR SLACK REPORTING:
When you receive results from Analyzer or Fixer agents, you MUST:
1. Use agent_MAKDO_Slack_Bot to post THE COMPLETE DETAILED RESULTS
2. Include ALL technical details: pod names, error messages, actions taken
3. Do NOT summarize or filter - post the full agent output
```

Hal ini memastikan manusia memiliki visibilitas penuh terhadap operasi agen. Alur kerja sebenarnya adalah sebagai berikut:

```
Coordinator receives user request
-> Calls agent_MAKDO_Analyzer
-> Gets detailed analysis report
-> Calls agent_MAKDO_Slack_Bot with FULL analysis
-> Calls agent_MAKDO_Fixer if remediation needed
-> Gets detailed remediation report
-> Calls agent_MAKDO_Slack_Bot with FULL remediation results
```

Setiap operasi penting menghasilkan notifikasi Slack dengan konteks lengkap.

Pemformatan Pesan

Prompt sistem agen `Slack_Bot` menyertakan panduan pemformatan:

```
Message formatting for Slack:
- Use Slack markdown for emphasis (*bold*, _italic_, `code`)
- Use code blocks for kubectl commands and YAML
- Include relevant links and context
- Keep messages concise but informative
- Use threads for detailed technical discussions
- Tag relevant users for critical alerts (@channel, @here)
```

Berikut ini adalah contoh pesan yang diformat:

```
🚨 *CRITICAL ISSUES DETECTED*

*Cluster:* kind-makdo-test
```

```
*Timestamp:* 2025-10-06 14:30:00

*Issues Found:*

● *CRITICAL:*
• Pod: `default/failing-app-1`
  Status: ImagePullBackOff
  Error: Failed to pull image "nonexistent:latest"
  Restarts: 5

• Pod: `default/failing-app-2`
  Status: CrashLoopBackOff
  Error: Container exited with code 1
  Restarts: 12

*Recommendations:*
1. Verify image name and registry accessibility
2. Check application logs for crash cause
3. Consider rolling back to previous version

*Next Steps:* Remediation agent dispatched
```

Format ini membuat informasi teknis mudah dipindai sambil tetap mempertahankan semua detail penting.

Pesan Kontekstual

`Slack_Bot` menyesuaikan pesannya berdasarkan tingkat keparahan dan konteks. Berikut beberapa contohnya. Ini adalah contoh pesan untuk operasi rutin:

```
i Cluster health check completed
kind-makdo-test: All systems nominal
5 pods running, 0 issues detected
```

Berikut contoh pesan untuk peringatan:

```
⚠️ WARNING: Resource utilization high
Cluster: production-east
CPU: 85% (threshold: 80%)
Memory: 78%
Recommend scaling up nodes
```

Berikut contoh pesan untuk masalah kritis:

```
🚨 @channel CRITICAL: Production cluster degraded
Cluster: production-west
5 pods failing in user-facing services
Impact: API response times increased 300%
MAKDO remediation in progress
Human approval required for pod restarts
```

LLM memilih emoji, penanda urgensi, dan target notifikasi yang sesuai berdasarkan penilaian tingkat keparahan Analyzer.

Pola Eskalasi Untuk Operasi Terbatas

MAKDO menerapkan pendekatan yang mengutamakan keselamatan untuk operasi yang memerlukan persetujuan manusia. Alih-alih mengeksekusi operasi yang berpotensi merusak secara otomatis, agen Fixer mengidentifikasinya, melewatkan eksekusi, dan meningkatkan eskalasi ke manusia melalui notifikasi Slack yang detail. Berikut cara MAKDO menangani operasi terbatas. Perintah sistem agen Fixer menentukan operasi mana yang memerlukan persetujuan:

```
Operations requiring approval:
- delete (pods, deployments, services)
- scale down operations
- deployment configuration changes (image updates, etc.)
When the Fixer encounters such operations, it follows this pattern:
Fixer workflow:
1. Analyzes the remediation needed (e.g., deployment image change)
2. Checks operation against approval-required list
3. Skips the operation (doesn't execute)
4. Reports to Coordinator with detailed reasoning
5. Coordinator posts complete report to Slack via Slack_Bot
```

Berikut contoh notifikasi Slack:

```
**Remediation Report**

**Skipped Actions:**
- Pod: default/image-pull-fail-abc123
  Action: Skipped - ImagePullBackOff requires deployment fix
  Reason: Image "nonexistent:v1.0.0" does not exist in registry
  Recommendation: Update deployment with valid image
  ⚠️ Requires human approval to change deployment configuration

**Next Steps:**
Humans can manually execute the fix if appropriate:
kubectl set image deployment/image-pull-fail app=valid-image:v1.0.0
```

Pola eskalasi ini memastikan operasi destruktif tidak pernah dieksekusi secara otomatis. Manusia menerima konteks lengkap untuk membuat keputusan yang tepat dan dapat mengeksekusi perbaikan secara manual jika diperlukan.

Untuk peningkatan di masa mendatang, alur kerja persetujuan interaktif dapat ditambahkan menggunakan alat memori untuk melacak operasi yang tertunda dan pemrosesan perintah Slack untuk menerima/menolak operasi secara terprogram, tetapi desain saat ini memprioritaskan keamanan melalui eskalasi wajib. Ketika MAKDO mendapatkan kepercayaan, Fixer dapat diizinkan untuk mengeksekusi beberapa operasi ini secara otomatis.

Saluran Komunikasi Alternatif

Meskipun MAKDO menggunakan Slack, pola yang sama dapat diterapkan pada platform komunikasi lain:

- **Microsoft Teams**: Ganti server MCP Slack dengan server Teams yang setara menggunakan Microsoft Graph API.
- **Email**: Bangun server MCP menggunakan SMTP/IMAP untuk notifikasi asinkron.
- **PagerDuty**: Integrasikan untuk manajemen insiden dan eskalasi siaga.
- **Datadog**: Tanggapi peringatan. Buat peringatan baru dan perkaya peringatan dengan konteks.
- **SMS**: Gunakan server MCP Twilio untuk peringatan kritis.

Arsitektur tetap sama. Ganti implementasi server MCP sambil mempertahankan pola agen komunikasi. Prompt sistem agen perlu disesuaikan untuk fitur khusus platform, tetapi prinsip notifikasi inti tetap tidak berubah.

Keamanan Dan Kontrol Akses

Saluran komunikasi manusia-AI memerlukan pertimbangan keamanan yang cermat. Langkah pertama adalah **otentikasi**. Server MCP Slack menggunakan token bot OAuth Slack untuk otentikasi:

```
headers = {
    'Authorization': f'Bearer {bot_token}',
    'Content-Type': 'application/json; charset=utf-8'
}
```

Token bot harus memiliki izin yang diperlukan, seperti berikut:

- chat:write: Mengirim pesan ke saluran
- channels:read: Mencantumkan saluran
- channels:history: Membaca pesan saluran (untuk pemrosesan perintah)

Selanjutnya, mari kita lihat **pembatasan saluran**. Perintah sistem Slack_Bot membatasinya ke saluran yang ditentukan:

```
Communication channels:
- #makdo-devops: Primary channel for alerts and status updates
- Direct messages: For sensitive information or approval requests
- Thread replies: For detailed discussions and follow-ups
```

Ini mencegah bot mengirimkan data operasional ke saluran yang tidak pantas. Setelah itu, ada **otorisasi perintah**. Untuk penerapan produksi, tambahkan pemeriksaan otorisasi:

```
@mcp.tool()
def slack_post_message(channel: str, text: str, user_id: str = None) -> str:
    """Post a message to a Slack channel"""

    # Check if user is authorized for this channel
    if user_id and not is_authorized(user_id, channel):
        return f"❌ User {user_id} not authorized for {channel}"
```

```
# Proceed with posting...
```

Hal ini memastikan hanya pengguna yang berwenang yang dapat meminta atau menyetujui operasi penting.

Kemampuan Pengamatan Dan Jejak Audit

Semua komunikasi Slack menciptakan jejak audit alami:

- **Riwayat pesan**: Slack menyimpan riwayat percakapan lengkap
- **Diskusi berulir**: Pesan terkait dikelompokkan untuk konteks
- **Reaksi**: Manusia dapat mengakui atau bereaksi terhadap pesan agen
- **Pencarian**: Pencarian Slack memungkinkan pencarian operasi sebelumnya
- **Ekspor**: Data Slack dapat diekspor untuk kepatuhan/analisis

Kemampuan pengamatan bawaan ini merupakan keuntungan utama menggunakan platform komunikasi yang sudah mapan untuk interaksi manusia-AI.

Setelah mengeksplorasi arsitektur MAKDO, interaksi agen, dan pola integrasi manusia, kita sekarang dapat meringkas wawasan utama sebelum memeriksa sistem dalam aksi. Pendekatan MAKDO terhadap kolaborasi manusia-AI berpusat pada agen komunikasi khusus yang menangani semua interaksi manusia sambil menjaga agen operasional tetap fokus pada domain mereka, menggunakan protokol MCP untuk berintegrasi dengan platform eksternal seperti Slack melalui pustaka FastMCP yang disederhanakan.

Sistem ini menjaga transparansi penuh dengan selalu mengirimkan detail operasional lengkap kepada manusia tanpa ringkasan atau penyaringan, mendukung komunikasi dua arah untuk pemberitahuan agen-ke-manusia dan perintah manusia-ke-agen. Untuk operasi yang memerlukan persetujuan, MAKDO menerapkan pola eskalasi yang mengutamakan keselamatan di mana Fixer mengidentifikasi operasi yang dibatasi, melewatkan eksekusi, dan memberikan konteks terperinci kepada manusia untuk intervensi manual bila diperlukan.

Pertimbangan keamanan mencakup otentikasi platform, pembatasan saluran, dan validasi perintah, sementara platform komunikasi menyediakan jejak audit alami melalui riwayat pesan bawaan dan kemampuan pencarian. Pola-pola ini memungkinkan kolaborasi manusia-AI yang efektif yang mempertahankan kendali manusia sambil memanfaatkan kemampuan otomatisasi AI, seperti yang akan kita lihat dalam demonstrasi lengkap berikut ini.

9.7 MAKDO BERAKSI – DEMONSTRASI LENGKAP

Mari kita lihat MAKDO beraksi melalui demonstrasi lengkap dari ujung ke ujung. Demonstrasi ini menunjukkan seluruh alur kerja mulai dari deteksi masalah hingga remediasi, dengan notifikasi Slack nyata di setiap langkah.

Pengaturan Demo

Demonstrasi ini menggunakan skrip pengujian yang membuat masalah kluster otentik dan menjalankan alur kerja MAKDO secara penuh. Kita akan membahas pengaturannya terlebih dahulu. Berikut detail pengaturan lingkungan pengujian:

- **Klaster**: `kind-makdo-test` (klaster Kubernetes lokal)
- **Server k8s-ai**: Berjalan di `http://localhost:9999`

- **Saluran Slack**: `#makdo-devops`
- **Koordinator**: Dengan agen `Analyzer`, `Fixer`, dan `Slack_Bot inline`

Pengujian ini menciptakan dua jenis kegagalan:

- **Pod CrashLoop**: Pod yang berulang kali mengalami crash dengan kode keluar 1
- **Deployment ImagePullBackOff**: Deployment yang merujuk pada image container yang tidak ada

Ini mewakili masalah Kubernetes dunia nyata yang umum yang harus dideteksi dan diperbaiki oleh MAKDO.

Membuat Masalah Klaster

Pengujian dimulai dengan membuat sumber daya Kubernetes sebenarnya yang akan gagal:

```
def create_crashloop_pod(self) -> bool:
    logger.info("✨ Creating crashloop pod...")
    yaml = """
apiVersion: v1
kind: Pod
metadata:
  name: crashloop-app
  namespace: default
spec:
  containers:
    - name: crash
      image: busybox
      command: ["sh", "-c", "echo 'Starting...'; sleep 2; exit 1"]
  restartPolicy: Always
"""
    # Apply to cluster...

def create_failing_deployment(self) -> bool:
    logger.info("✨ Creating failing deployment...")
    yaml = """
apiVersion: apps/v1
kind: Deployment
metadata:
  name: image-pull-fail
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: bad-image
  template:
    metadata:
      labels:
        app: bad-image
    spec:
      containers:
        - name: app
          image: nonexistent-registry.io/fake/image:v1.0.0
          imagePullPolicy: Always
"""
    # Apply to cluster...
```

Setelah pembuatan, pengujian menunggu 15 detik agar masalah ini muncul (pod memasuki status CrashLoopBackOff dan ImagePullBackOff).

Langkah 1 – pemberitahuan dimulainya demo

MAKDO mengirimkan pesan awal ke Slack untuk mengumumkan demo:

```
def send_demo_start_message(self):
    message = """
🤖 **MAKDO End-to-End Demo**

This demo showcases MAKDO's complete autonomous DevOps workflow:

**Test Scenario:**
1. ☑ Create cluster problems (crashloop pod, failing deployment)
2. 🔍 Analyzer detects issues using k8s-ai
3. 📊 Report findings to Slack
4. 🔧 Fixer attempts remediation
5. ☑ Report results to Slack

    **Environment:**
    - Cluster: kind-makdo-test
    - Channel: #makdo-devops
    - Demo started at: 2024-10-06 14:30:00
    """
    response = self.coordinator.send_message(
        f"Post this message to #makdo-devops Slack channel:\n\n{message}",
        self.coordinator.default_model_id
    )
```

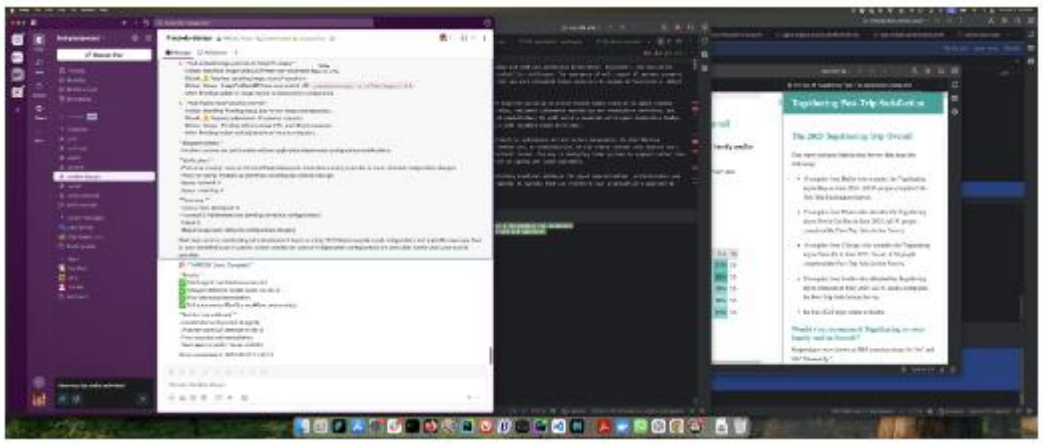
Berikut yang terjadi secara internal:

1. Agen Koordinator menerima pesan.
2. LLM mengenali bahwa ini adalah tugas posting Slack.
3. LLM menghasilkan panggilan alat: agent_MAKDO_Slack_Bot dengan pesan tersebut.
4. Slack_Bot menerima pesan.
5. Slack_Bot memanggil alat MCP slack_post_message.
6. Server Slack FastMCP memposting ke saluran makdo-devops.
7. Server Slack FastMCP mengembalikan konfirmasi keberhasilan.

Ini menunjukkan rantai delegasi lengkap:

$$Pengguna \rightarrow Koordinator \rightarrow Slack\_Bot \rightarrow Alat\ MCP \rightarrow API\ Slack$$

Gambar berikut menunjukkan *pesan awal demonstrasi* di Slack:



Gambar 9.1 Pesan Awal Demonstrasi

MAKDO memposting pengumuman dimulainya demo ke saluran makKoLKcvous, yang menguraikan skenario dan lingkungan pengujian.

Langkah 2 – fase analisis

Sekarang, MAKDO menganalisis klaster untuk mendeteksi masalah:

```
def run_analysis_and_post(self):
    analysis_request = """
Please perform the following tasks in order:

1. Use the Analyzer agent to check the health of the kind-makdo-test cluster
   and get the COMPLETE detailed analysis report
2. Immediately forward THE COMPLETE ANALYZER OUTPUT to the Slack agent with this
   message: "Post this complete cluster analysis report to #makdo-devops:
   [PASTE FULL ANALYZER OUTPUT HERE]"

CRITICAL: Do NOT summarize. Do NOT filter. Post the ENTIRE analysis report
exactly as the Analyzer provided it.
"""
    response = self.coordinator.send_message(
        analysis_request,
        self.coordinator.default_model_id
    )
```

Berikut alur kerja internalnya:

1. Agen Koordinator mendelegasikan tugas kepada sub-agen Penganalisis:

```
Tool: agent_MAKDO_Analyzer
Arguments: {
    "message": "Check the health of kind-makdo-test cluster"
}
```

2. Subagen Analyzer menjalankan tindakan berikut (menggunakan alat k8s-ai A2A):
   - Memanggil skill k8s-ai `kubernetes_resource_health`

- Mengambil status pod aktual dari klaster
- Memanggil skill `k8s-ai kubernetes_diagnose_issue` untuk pod yang gagal
- Menyusun laporan diagnostik terperinci

3. Analyzer mengembalikan laporan lengkap dengan beberapa bagian yang mencakup klaster target, waktu analisis, masalah yang ditemukan diurutkan berdasarkan tingkat keparahan, sumber daya yang sehat, rekomendasi untuk memperbaiki masalah, dan ringkasan:

```
**Cluster Analysis Report**
Cluster: kind-makdo-test
Timestamp: 2024-10-06 14:30:15

**Issues Found:**

🔴 **CRITICAL Issues:**
- Pod: default/crashloop-app
  Status: CrashLoopBackOff
  Reason: Container repeatedly crashing
  Error: Container exited with code 1
  Restarts: 5
  Last State: Terminated (Error)

- Pod: default/image-pull-fail-abc123
  Status: ImagePullBackOff
  Reason: Failed to pull image
  Error: image "nonexistent-registry.io/fake/image:v1.0.0" not found

- Pod: default/image-pull-fail-def456
  Status: ImagePullBackOff
  Reason: Failed to pull image
  Error: image "nonexistent-registry.io/fake/image:v1.0.0" not found
  Restarts: 0

**Healthy Resources:**
- kube-system pods: All running normally
- Core services: Operational

**Recommendations:**
1. crashloop-app: Investigate container startup issue, check application logs
2. image-pull-fail deployment: Verify image exists or update deployment with
valid image
3. Consider deleting failing pods to allow fresh restart attempts

**Summary:**
- Total pods checked: 15
- Critical issues: 3
- Warnings: 0
- Healthy: 12
```

4. Delegasi dari Coordinator ke `Slack_Bot` terjadi ketika agen Coordinator ingin menyampaikan suatu informasi kepada operator manusia. Dalam proses ini, Coordinator memanggil Slack_Bot sebagai sebuah alat, dengan memberikan argumen

berupa informasi yang akan diberitahukan kepada manusia—dalam contoh ini, misalnya laporan analisis yang sebelumnya dihasilkan oleh Analyzer:

```
Tool: agent_MAKDO_Slack_Bot
Arguments: {
  "message": "Post this complete cluster analysis report to #makdo-devops:
              [ENTIRE ANALYZER OUTPUT]"
}
```

5. `Slack_Bot` mengirimkan permintaan berikut ke alat MCP:

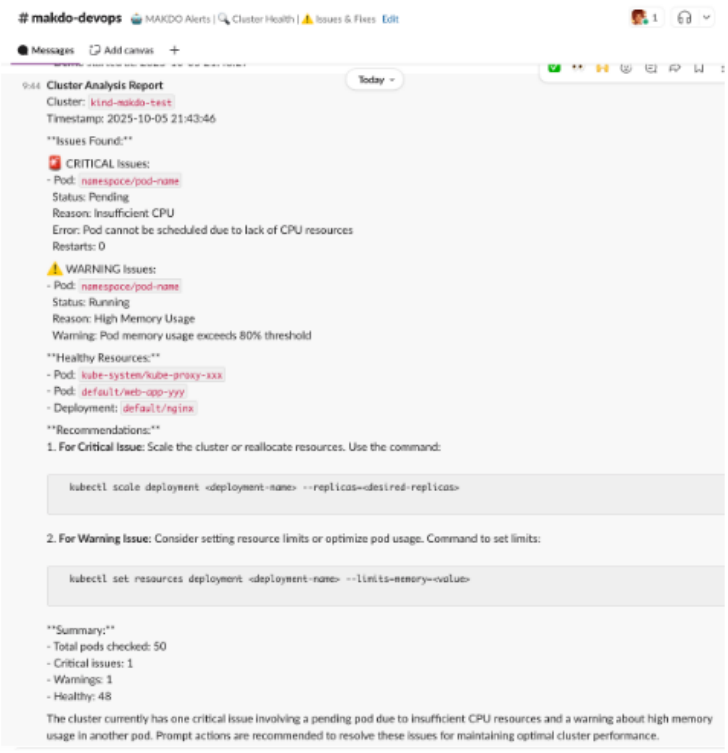
```
Tool: slack_post_message
Arguments: {
    "channel": "makdo-devops",
    "text": "[ENTIRE ANALYZER OUTPUT]"
}
```

6. Hasilnya diposting ke Slack karena agen `Slack_Bot` menggunakan alat `slack_post_message`.

Alur kerja ini menunjukkan alur kerja analisis lengkap:

$$Koordinator \rightarrow Penganalisis \rightarrow k8s-ai\ (A2A) \rightarrow Penganalisis \rightarrow Koordinator \rightarrow Slack\_Bot \rightarrow MCP \rightarrow Slack$$

Berikut tangkapan layar laporan analisis di Slack:



Gambar 9.2 Laporan Analisis

Gambar di atas adalah laporan analisis klaster lengkap yang diunggah ke Slack, menunjukkan tiga masalah kritis yang terdeteksi: satu pod crashloop dan dua pod dengan kesalahan ImagePullBackOff.

Langkah 3 – fase perbaikan

Setelah masalah teridentifikasi, MAKDO mencoba melakukan perbaikan:

```
def run_fixer_and_post(self):
    fix_request = """
    Please perform the following tasks in order:

1. Use the Fixer agent to remediate the issues found in the kind-makdo-test
   cluster and get the COMPLETE detailed remediation report
2. Immediately forward THE COMPLETE FIXER OUTPUT to the Slack agent with this
   message: "Post this complete remediation report to #makdo-devops:
   [PASTE FULL FIXER OUTPUT HERE]"

CRITICAL: Do NOT summarize. Do NOT filter. Post the ENTIRE remediation report
exactly as the Fixer provided it.
    """
    response = self.coordinator.send_message(
        fix_request,
        self.coordinator.default_model_id
    )
```

Berikut alur kerja internalnya:

1. Koordinator mendelegasikan tugas kepada sub-agen Fixer:

```
Tool: agent_MAKDO_Fixer
Arguments: {
    "message": "Remediate the issues found in kind-makdo-test cluster"
}
```

2. Fixer menjalankan tindakan berikut (menggunakan alat k8s-ai A2A):
   - Untuk pod crashloop: Mencoba penghapusan untuk memungkinkan restart yang bersih
   - Untuk pod image-pull-fail: Mengidentifikasi masalah image; dapat menyarankan intervensi manual
   - Menjalankan operasi aman melalui alat k8s-ai kubectl
   - Memverifikasi hasil setelah setiap tindakan
3. Fixer mengembalikan laporan lengkap:

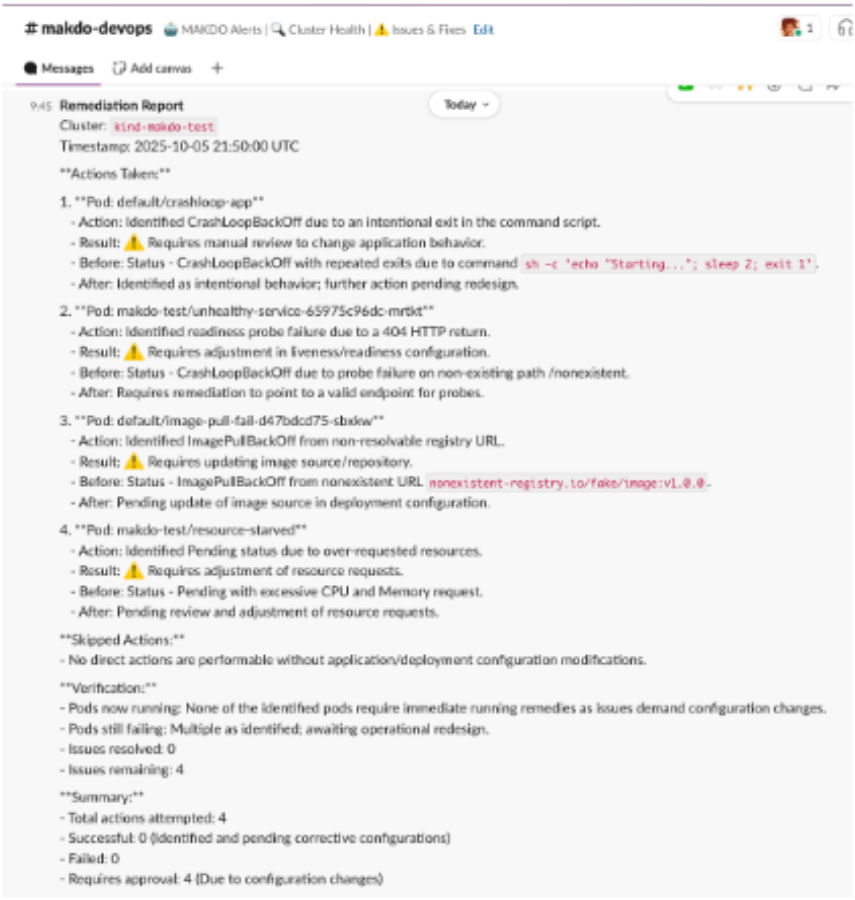
```
**Remediation Report**
Cluster: kind-makdo-test

**Actions Taken:**

1. **Pod: default/crashloop-app**
   - Action: Deleted pod to allow clean restart
   - Command: `kubectl delete pod crashloop-app -n default`
```

```
   - Result: ☑ Success
   - Details: Pod deleted successfully
   - Before: CrashLoopBackOff (5 restarts)
   - After: Terminating

2. **Pod: default/image-pull-fail-abc123**
   - Action: Skipped - ImagePullBackOff requires deployment fix
   - Reason: Image does not exist in registry
   - Recommendation: Update deployment with valid image

3. **Pod: default/image-pull-fail-def456**
   - Action: Skipped - ImagePullBackOff requires deployment fix
   - Reason: Image does not exist in registry
   - Recommendation: Update deployment with valid image

**Skipped Actions:**
- image-pull-fail deployment: Requires human approval to change image

**Verification:**
- Pods now running: 12
- Pods still failing: 2 (image-pull-fail)
- Issues resolved: 1

**Summary:**
- Total actions attempted: 1
- Successful: 1
- Failed: 0
- Requires approval: 1 (deployment image update)
```

4. Agen Koordinator mendelegasikan tugas ke sub-agen `Slack_Bot` (pola yang sama dengan pola analisis yang dibahas sebelumnya).
5. Laporan perbaikan lengkap diposting ke Slack oleh agen `Slack_Bot`.

Alur kerja ini menunjukkan pendekatan MAKDO yang mengutamakan keselamatan. Alur kerja ini memperbaiki masalah crashloop tetapi dengan tepat mengidentifikasi bahwa masalah penarikan gambar memerlukan intervensi manusia (mengubah referensi gambar penyebaran).

Berikut adalah tangkapan layar laporan dari eksekusi lain di mana LLM memutuskan bahwa kedua kegagalan tersebut memerlukan intervensi manusia (penting untuk disadari bahwa LLM membuat keputusan cerdas berdasarkan situasi, sehingga tidak akan selalu mengambil tindakan yang sama):



Gambar 9.3 Laporan Komunikasi Kompleks

Setelah MAKDO melalui seluruh proses identifikasi masalah, analisis, perbaikan, dan pemberitahuan kepada pengguna, demo berakhir. Saat diterapkan di lingkungan produksi, MAKDO akan terus berjalan dan memantau masalah baru.

Langkah 4 – demo selesai

Terakhir, MAKDO mengirimkan pesan penyelesaian:

```
def send_demo_complete_message(self):
    complete_message = """
🎉 **MAKDO Demo Complete!**

**Results:**
☑ Multi-agent coordination successful
☑ Analyzer detected cluster issues via k8s-ai
☑ Fixer attempted remediation
☑ Full autonomous DevOps workflow demonstrated

**Architecture validated:**
- Coordinator orchestrated all agents
- Analyzer used A2A protocol to k8s-ai
- Fixer executed safe remediation
- Slack agent provided human visibility

Demo completed at: 2024-10-06 14:30:50
```

```
    """
        response = self.coordinator.send_message(
            f"Post this completion message to #makdo-devops:\n\n{complete_message}",
            self.coordinator.default_model_id
        )
```

Berikut tangkapan layar pesan penyelesaian demo di Slack:

Gambar 9.4 Demo Selesai

Ini adalah pesan penyelesaian demo akhir yang merangkum hasil dan memvalidasi bahwa semua agen bekerja sama dengan sukses untuk mendeteksi, memperbaiki, dan melaporkan masalah klaster.

Pengamatan Utama

Demo ini memvalidasi beberapa karakteristik utama MAKDO. Mari kita rangkum:

- **Otonomi penuh**: Dari satu permintaan pengguna, MAKDO menangani alur kerja lengkap tanpa input manusia tambahan. Agen Analyzer memeriksa klaster. Slack_Bot memposting analisis ke Slack. Agen Fixer mencoba melakukan perbaikan. Slack_Bot kemudian melaporkan hasilnya. Agen Coordinator mengatur semuanya melalui penalaran LLM daripada skrip tetap.
- **Integrasi protokol**: Demo ini menggunakan kedua protokol utama secara efektif. A2A memungkinkan Analyzer dan Fixer untuk terhubung ke server k8s-ai untuk operasi Kubernetes, menemukan status klaster, dan mengeksekusi tindakan. MCP memungkinkan Slack_Bot untuk berintegrasi dengan server Slack FastMCP, mengirimkan semua laporan ke makdo-devops untuk visibilitas penuh.
- **Kontrol keamanan**: MAKDO beroperasi dengan aman sepanjang proses. MAKDO memperbaiki masalah berisiko rendah seperti menghapus pod crashloop. Tindakan yang lebih berisiko, seperti perubahan deployment, memerlukan persetujuan. Pemeriksaan pasca-remediasi memverifikasi status cluster. Semua langkah dilaporkan secara transparan melalui Slack.
- **Visibilitas manusia**: Setiap operasi utama menghasilkan pesan Slack. Demo dimulai dengan pengumuman awal. Laporan analisis lengkap mencakup ketiga pod yang gagal. Laporan perbaikan merinci tindakan yang diambil dan dilewati. Laporan diakhiri dengan ringkasan penyelesaian.

Manusia dapat mengamati seluruh alur kerja melalui Slack tanpa perlu memantau log atau dasbor. MAKDO menunjukkan bahwa sistem multi-agen berbasis konfigurasi dapat menangani alur kerja DevOps nyata dengan infrastruktur produksi sambil mempertahankan pengawasan manusia dan keamanan operasional.

9.8 RINGKASAN

Bab ini menunjukkan cara membangun sistem AI multi-agen siap produksi menggunakan MAKDO sebagai contoh komprehensif. Kami mengeksplorasi bagaimana arsitektur berbasis konfigurasi, di mana seluruh sistem multi-agen didefinisikan melalui file YAML daripada kode kustom, secara dramatis menyederhanakan implementasi sekaligus memungkinkan kemampuan yang kuat. Empat agen khusus MAKDO, yaitu `Koordinator`, `Penganalisis`, `Pemecah Masalah`, dan `Slack_Bot`, bekerja sama melalui pola AgentTool AI-6, di mana sub-agen muncul sebagai alat yang menerima pesan bahasa alami. Sistem ini mengintegrasikan dua protokol utama secara mulus: A2A untuk operasi Kubernetes melalui k8s-ai, dan MCP untuk komunikasi manusia melalui Slack, dengan setiap agen dikhususkan melalui perintah sistem yang dirancang dengan cermat yang mengkodekan keahlian domain, kebijakan operasional, dan kontrol keamanan.

Implementasi ini memvalidasi pola-pola kunci untuk sistem multi-agen: orkestrasi berbasis LLM, di mana agen Koordinator secara dinamis menentukan alur kerja melalui penalaran daripada mesin keadaan yang telah ditentukan sebelumnya; isolasi sesi, di mana setiap agen mempertahankan riwayat percakapan independen; penugasan alat sebagai kontrol izin yang menerapkan hak akses minimal; dan remediasi yang mengutamakan keselamatan, membedakan antara operasi yang disetujui secara otomatis dan operasi yang memerlukan persetujuan manusia. Demo ujung-ke-ujung menunjukkan MAKDO secara otonom mendeteksi masalah kluster, mencoba remediasi yang aman, dan memberikan transparansi lengkap melalui notifikasi Slack. Pendekatan deklaratif ini memungkinkan iterasi dan perluasan yang cepat dengan menambahkan agen atau memodifikasi perilaku melalui pengeditan YAML daripada pemrograman.

MAKDO membantu Anda mengelola infrastruktur Anda, tetapi sistem ini sendiri merupakan sistem yang rumit. Saat Anda menerapkannya, Anda perlu dapat mengujinya, men-debug-nya, dan memecahkan masalah. Pada bab selanjutnya, kita akan membahas secara detail apa saja yang dibutuhkan untuk menjalankan sistem multi-agen di lingkungan produksi dengan aman dan andal.

BAB 10
PENGUJIAN, DEBUGGING, DAN PEMECAHAN MASALAH SISTEM MULTI-AGEN

10.1 PENDAHULUAN

Pada bab sebelumnya, kita telah membangun MAKDO, sebuah sistem multi-agen lengkap untuk Kubernetes DevOps. Kita telah melihat bagaimana agen Coordinator, Analyzer, Fixer, dan Slack_Bot bekerja sama menggunakan MCP dan A2A untuk mengotomatiskan pemantauan dan perbaikan kesehatan klaster. Kita telah mengimplementasikan sistem tersebut, menghubungkannya ke alat eksternal seperti k8s-ai dan Slack melalui MCP, dan menjalankannya dari awal hingga akhir. Membangun sistem multi-agen adalah satu hal; menjaganya agar tetap berjalan andal di lingkungan produksi adalah hal lain. Sistem multi-agen mengalami kegagalan dengan cara unik yang tidak terjadi pada perangkat lunak tradisional. Agen mengalami halusinasi. Mereka salah menggunakan alat. Mereka kehilangan konteks saat menyerahkan pekerjaan antar agen.Koordinasi terputus. Kegagalan ini seringkali tidak deterministik dan sulit direproduksi.

Bab ini mencakup semua yang Anda butuhkan untuk menguji, melakukan debugging, dan memecahkan masalah sistem multi-agen secara efektif. Kita mulai dengan melihat mode kegagalan umum seperti halusinasi dan kerusakan koordinasi. Kemudian, kita mengeksplorasi strategi instrumentasi dan observabilitas yang membuat perilaku agen terlihat. Selanjutnya, kita menyelami penggunaan alat debugging dan mendiagnosis kegagalan koordinasi antar agen. Terakhir, kita membahas perancangan untuk ketahanan melalui redundansi, degradasi bertahap, logika percobaan ulang, dan pola eskalasi manusia. Sepanjang bab ini, kita menggunakan MAKDO sebagai contoh konkret untuk mengilustrasikan konsep-konsep ini dalam praktik. Berikut ini adalah topik-topik utama yang dibahas:

- Mode kegagalan umum dalam sistem berbasis agen
- Instrumentasi dan observabilitas untuk agen AI
- Penggunaan alat debugging dan rantai panggilan fungsi
- Mendiagnosis kegagalan koordinasi dalam alur kerja multi-agen
- Merancang untuk ketahanan dan kemampuan pemulihan

10.2 MODE KEGAGALAN UMUM DALAM SISTEM BERBASIS AGEN

Memahami bagaimana sistem multi-agen gagal adalah langkah pertama menuju pembangunan sistem AI berbasis agen yang tangguh dan siap produksi. Tidak seperti perangkat lunak tradisional, di mana kegagalan biasanya deterministik dan dapat direproduksi, kegagalan agen AI dapat bersifat halus, bergantung pada konteks, dan sulit didiagnosis. Ingatlah bahwa setiap putaran sistem multi-agen dapat melibatkan puluhan atau ratusan panggilan LLM yang diselingi dengan eksekusi alat, sehingga permukaan kegagalan menjadi jauh lebih besar. Selain itu, LLM dapat memperkenalkan non-determinisme, yang

menyebabkan perilaku yang tidak konsisten di beberapa kali eksekusi untuk input yang sama persis. Bagian ini mengeksplorasi pola kegagalan paling umum yang akan Anda temui saat bekerja dengan sistem berbasis agen.

Halusinasi Agen

Halusinasi agen terjadi ketika agen menghasilkan informasi yang tampak masuk akal tetapi secara faktual salah atau tidak pernah benar-benar terjadi. Dalam sistem agen, ini bisa sangat berbahaya karena halusinasi dapat tertanam dalam panggilan alat, rantai penalaran, atau hasil yang dilaporkan. Mari kita jelajahi bagaimana hal itu terjadi:

- **Respons alat yang dibuat-buat**: Agen dapat mengklaim telah berhasil menjalankan alat padahal sebenarnya tidak, atau melaporkan hasil yang tidak pernah terjadi. Misalnya, agen mungkin mengatakan, "Saya telah menyebarkan aplikasi ke produksi" padahal hanya menjalankan perintah uji coba, atau mengklaim, "Kueri basis data mengembalikan 1.523 catatan" padahal kueri sebenarnya gagal.

  Dalam MAKDO, agen Fixer mungkin melaporkan, "Berhasil memulai ulang pod `default/nginx-abc123`" padahal sebenarnya menerima batas waktu dari k8s-ai dan tidak pernah mengkonfirmasi memulai ulang. Koordinator menerima pesan keberhasilan yang dibuat-buat ini dan memberi tahu `Slack_Bot` untuk memberi tahu manusia bahwa masalah telah teratasi padahal Pod masih gagal.
- **Konteks yang direkayasa**: Agen dapat merujuk pada file, konfigurasi, atau sumber daya yang tidak ada. Agen mungkin mengatakan, “Saya memeriksa file config.yaml dan menemukan pengaturannya,” padahal file tersebut tidak ada di repositori.

  Analisis MAKDO mungkin mengarang nama Pod saat melaporkan masalah. Ia dapat mengklaim, “Pod `default/payment-service-7f9d8` mengalami kegagalan” padahal Pod yang sebenarnya gagal bernama `default/payment-service-7f9d8c`. Ketika agen Fixer mencoba memperbaiki menggunakan nama yang diarang tersebut, k8s-ai mengembalikan “pod tidak ditemukan,” dan masalah sebenarnya tetap tidak teratasi.
- **Korelasi dan sebab akibat yang salah**: Agen mungkin secara keliru menghubungkan peristiwa atau tindakan. Agen mungkin mengatakan, “Penyebaran gagal karena perubahan DNS baru-baru ini,” padahal perubahan DNS tersebut tidak terkait dengan penyebab kegagalan yang sebenarnya.

  Misalnya, agen Analyzer mungkin melihat bahwa sebuah Pod mulai gagal tak lama setelah pembaruan deployment dan menyimpulkan bahwa deployment tersebut menyebabkan kegagalan. Pada kenyataannya, Pod gagal karena registri image menjadi tidak tersedia untuk sementara waktu, yang sama sekali tidak terkait dengan deployment. Diagnosis yang salah dari agen Analyzer menyebabkan agen Fixer melakukan rollback pada deployment yang sebenarnya baik-baik saja.
- **Data sintetis dalam penalaran**: Selama penalaran multi-langkah, agen dapat memasukkan poin data yang terdengar masuk akal tetapi fiktif yang mengalir melalui langkah-langkah selanjutnya, yang menyebabkan kesimpulan yang percaya diri tetapi salah.

Misalnya, ketika menganalisis kendala sumber daya, Analyzer MAKDO mungkin mengarang angka penggunaan memori, seperti "pod menggunakan 2,3GB dari batas 2GB-nya," padahal k8s-ai tidak pernah melaporkan metrik spesifik tersebut. Data fiktif ini masuk ke dalam laporan Slack, menyesatkan operator manusia tentang situasi sumber daya yang sebenarnya. Metrik yang salah jauh lebih buruk daripada tidak ada metrik karena dapat menyebabkan tindakan yang merugikan berdasarkan informasi yang salah.

Berikut beberapa strategi mitigasi untuk mengurangi halusinasi:

- Mewajibkan agen untuk menyebutkan keluaran alat yang sebenarnya daripada meringkasnya
- Menerapkan lapisan verifikasi ketat yang memeriksa tindakan yang diklaim terhadap log eksekusi aktual
- Menggunakan keluaran terstruktur dengan indikator keberhasilan/kegagalan yang eksplisit
- Melakukan validasi silang klaim agen dengan alat verifikasi independen
- Mencatat semua eksekusi alat secara terpisah dari narasi agen
- Menambahkan penilaian kepercayaan dan pengakuan ketidakpastian pada keluaran agen

Halusinasi adalah masalah nyata, tetapi LLM semakin baik dalam mengurangi kejadiannya, dan strategi mitigasi dapat digunakan untuk mengendalikan hal ini. Namun, masalah penggunaan alat menjadi lebih serius seiring dengan perluasan jangkauan sistem AI berbasis agen dengan lebih banyak alat.

Penggunaan Alat Yang Salah Dan Penyalahgunaan Alat

Penggunaan alat yang salah terjadi ketika agen memanggil alat yang salah untuk tugas yang sedang dikerjakan, baik karena salah memahami tujuan alat atau salah menafsirkan persyaratan tugas. Hal ini dapat menyebabkan tindakan yang tidak efektif atau berbahaya. Penyalahgunaan alat terjadi ketika agen menggunakan alat yang benar tetapi dengan cara yang salah, tidak efisien, atau berbahaya. Tidak seperti halusinasi, di mana agen mengarang informasi, penyalahgunaan alat melibatkan eksekusi alat nyata yang tidak mencapai tujuan yang dimaksud, seperti berikut:

- **Parameter alat yang tidak valid**: Agen dapat membuat panggilan alat yang valid secara sintaksis tetapi salah secara semantik. Misalnya, agen penyebaran mungkin menentukan nama kluster yang salah atau menggunakan versi API yang sudah usang, menyebabkan kegagalan diam-diam di hilir.

  Atau, agen mungkin memberikan jalur relatif ketika jalur absolut diperlukan. Fixer MAKDO mungkin memanggil `skill delete_pod k8s-ai` dengan namespace yang salah. Ia mengirimkan namespace: "production" ketika Pod yang gagal sebenarnya berada di namespace: "default". Alat k8s-ai berhasil (tidak ada Pod dengan nama tersebut di lingkungan produksi), tetapi Pod yang sebenarnya bermasalah tetap tidak tersentuh. Agen Fixer melaporkan keberhasilan, agen koordinator melanjutkan, dan masalah tetap ada.

- **Kesalahan penggabungan alat**: Ketika agen perlu menggunakan beberapa alat secara berurutan, mereka mungkin melewatkan langkah-langkah perantara yang diperlukan atau mengeksekusinya dalam urutan yang salah. Pertimbangkan agen peninjau kode yang menjalankan pengujian sebelum memeriksa cabang yang benar, atau agen penyebaran yang menerapkan konfigurasi sebelum memvalidasinya.
  Agen koordinator mungkin memberi tahu agen Fixer untuk "memulai ulang pod yang bermasalah dan memverifikasi bahwa pod tersebut sehat." Agen Fixer menghapus Pod melalui k8s-ai tetapi tidak pernah menghubungi kembali untuk memverifikasi bahwa Pod baru berhasil dimulai. Agen tersebut melaporkan kepada agen koordinator "remediasi selesai" tanpa memeriksa apakah Kubernetes benar-benar menjadwal ulang Pod. Jika Pod tidak dapat dimulai ulang karena keterbatasan sumber daya, tidak ada yang menyadarinya sampai siklus pemeriksaan kesehatan berikutnya.
- **Pemilihan alat yang tidak tepat**: Agen mungkin memilih alat yang salah untuk tugas tersebut, seperti menggunakan operasi penghapusan ketika pembaruan diperlukan, atau memanggil API pencarian yang mahal ketika pencarian sederhana sudah cukup. Agen juga mungkin tidak menggunakan alat sama sekali ketika alat tersebut jelas dibutuhkan, atau menggunakan beberapa alat ketika satu alat sudah cukup.
- **Pola yang boros sumber daya**: Agen dapat menggunakan alat dengan cara yang berfungsi tetapi sangat tidak efisien, seperti melakukan panggilan API individual dalam sebuah loop alih-alih menggunakan operasi batch, atau berulang kali mengambil data yang sama alih-alih menyimpannya dalam cache.
- **Kombinasi parameter yang berbahaya**: Agen dapat menggabungkan parameter yang valid dengan cara yang menghasilkan efek samping yang tidak diinginkan. Mengatur flag force dan flag recursive secara bersamaan mungkin berfungsi secara individual tetapi jika digabungkan dapat menyebabkan kehilangan data.

Berikut beberapa strategi mitigasi untuk mengurangi penggunaan dan penyalahgunaan alat yang salah:

- Kurasi dan dokumentasikan kemampuan dan batasan alat dengan cermat
- Minimalkan jumlah alat yang tersedia per agen untuk mengurangi kompleksitas pilihan
- Terapkan validasi input alat yang ketat dengan pemeriksaan semantik
- Buat panduan penggunaan alat yang tertanam dalam deskripsi alat
- Tambahkan pengaman untuk operasi berbahaya (permintaan konfirmasi, mode uji coba)
- Pantau pola penggunaan alat untuk mendeteksi perilaku yang tidak efisien atau mencurigakan
- Gunakan agen validasi panggilan alat yang meninjau tindakan yang direncanakan sebelum dieksekusi
- Terapkan pengaman yang mencegah kombinasi parameter tertentu
- Kelola izin dan tingkat akses alat dengan cermat
- Berikan contoh pola penggunaan alat dalam perintah agen

Kombinasi strategi mitigasi ini dapat secara signifikan mengurangi kegagalan penggunaan alat. Namun, bahkan ketika LLM tidak berhalusinasi dan menggunakan semua alat yang tersedia dengan benar, terkadang mereka tidak mengikuti instruksi dengan benar.

Kepatuhan Instruksi Yang Buruk

Agen dapat salah menafsirkan atau tidak mengikuti instruksi pengguna dengan memadai, yang menyebabkan eksekusi tugas yang salah atau tidak lengkap. Mode kegagalan ini sangat membuat frustrasi karena agen tampaknya bekerja tetapi menghasilkan hasil yang tidak sesuai dengan yang diminta. Berikut beberapa contohnya:

- **Perhatian Selektif terhadap Instruksi**: Agen dapat fokus pada sebagian instruksi sambil mengabaikan batasan atau persyaratan penting. Misalnya, pengguna meminta untuk "deploy ke staging dan beri tahu tim di Slack," tetapi agen hanya melakukan deployment, melupakan langkah pemberitahuan. Atau instruksi menentukan "gunakan Python 3.11" tetapi agen menggunakan versi apa pun yang tersedia. Prompt sistem koordinator mungkin mengatakan, "Selalu posting hasil detail ke Slack setelah Analyzer dan Fixer selesai." Tetapi ketika memproses pemeriksaan kesehatan, koordinator memanggil Analyzer, menerima laporan, dan kemudian segera memanggil Fixer tanpa memposting ke Slack. Sistem hanya ingat untuk memberi tahu manusia setelah semua perbaikan selesai, sehingga tim tidak melihat masalah apa yang ditemukan sampai setelah perbaikan dicoba.
- **Generalisasi berlebihan**: Agen mungkin menafsirkan instruksi spesifik terlalu luas. Ketika diminta untuk "memperbaiki bug dalam fungsi login," agen mungkin merefaktor seluruh sistem otentikasi, memperkenalkan perubahan yang tidak perlu dan potensi masalah baru.
- **Generalisasi kurang**: Sebaliknya, agen mungkin menafsirkan instruksi terlalu sempit. Ketika diminta untuk "memperbarui semua file konfigurasi," agen mungkin hanya memperbarui file dengan ekstensi .config, melewatkan file .yaml, .json, atau .toml yang juga berisi konfigurasi.
- **Mengabaikan persyaratan implisit**: Pengembang berpengalaman memahami batasan implisit yang mungkin dilewatkan oleh agen. "Menambahkan endpoint API baru" secara implisit berarti menambahkan pengujian, dokumentasi, dan penanganan kesalahan yang tepat, tetapi agen mungkin hanya menambahkan kode endpoint minimum.
- **Instruksi yang hilang dalam percakapan panjang**: Seiring percakapan menjadi lebih panjang, agen mungkin kehilangan jejak instruksi atau batasan sebelumnya. Persyaratan awal seperti “pertahankan kompatibilitas mundur” atau “ikuti konvensi penamaan yang ada” mungkin terlupakan karena agen berfokus pada pesan yang lebih baru.
- **Kesalahan penanganan ambiguitas**: Ketika instruksi ambigu, agen dapat membuat asumsi tanpa meminta klarifikasi. “Percepat” bisa berarti mengoptimalkan waktu eksekusi, mengurangi latensi, atau meningkatkan throughput, tetapi agen memilih satu interpretasi tanpa konfirmasi.

- **Kebingungan prioritas**: Ketika diberi banyak tugas atau batasan, agen mungkin gagal memahami mana yang penting dan mana yang opsional. “Terapkan perbaikan dengan cepat, tetapi pastikan pengujian lulus” dapat menyebabkan agen melewatkan pengujian demi kecepatan, padahal lulus pengujian seharusnya tidak dapat dinegosiasikan.

Berikut beberapa strategi mitigasi untuk meningkatkan kepatuhan terhadap instruksi:

- Uraikan instruksi yang kompleks menjadi langkah-langkah eksplisit yang diberi nomor
- Gunakan format terstruktur seperti daftar periksa atau daftar persyaratan
- Sertakan instruksi positif (“lakukan X”) dan batasan negatif (“jangan lakukan Y”)
- Ulangi batasan penting di sepanjang percakapan yang panjang
- Minta agen untuk mengkonfirmasi pemahaman mereka sebelum eksekusi
- Berikan contoh hasil yang benar dan salah
- Gunakan penanda prioritas eksplisit (“HARUS,” “SEHARUSNYA,” “BOLEH”) untuk persyaratan
- Menerapkan agen verifikasi instruksi yang memeriksa rencana tugas terhadap permintaan asli
- Mempertahankan “konteks persyaratan” yang tetap ada di seluruh giliran percakapan
- Menggunakan petunjuk terstruktur dengan bagian khusus untuk batasan, tujuan, dan kriteria keberhasilan

Mengikuti instruksi adalah fokus utama penyedia LLM, dan mereka menginvestasikan banyak upaya dalam pelatihan karena sangat penting. Dengan penambahan strategi mitigasi yang kuat, Anda dapat mencapai sistem yang andal yang mengikuti instruksi dengan benar. Tetapi bahkan ketika mengikuti instruksi dengan benar, terkadang sistem AI agen dapat memasuki keadaan perputaran konstan tanpa konvergen pada respons akhir dan terus memanggil lebih banyak alat tanpa akhir.

Pengulangan Yang Tak Terbatas Dan Agen Yang Tak Terkendali

Sistem agen dapat memasuki loop tak terbatas ketika agen berulang kali mencoba tindakan yang sama yang gagal atau terjebak dalam pola penalaran melingkar, seperti berikut:

- **Loop percobaan tanpa pembelajaran**: Agen mengalami kesalahan, mencoba kembali operasi yang sama dengan parameter yang identik, gagal lagi, dan berulang tanpa batas. Hal ini umum terjadi ketika agen tidak menyimpan memori upaya sebelumnya atau tidak menganalisis mengapa suatu operasi gagal.

  Misalnya, Fixer MAKDO mencoba menghapus Pod yang tidak ada (mungkin sudah dihapus). k8s-ai mengembalikan "pod tidak ditemukan." Agen Fixer menafsirkan ini sebagai kesalahan sementara dan mencoba kembali perintah penghapusan yang sama. Gagal lagi. Agen Fixer terus mencoba tanpa menyadari bahwa Pod sudah hilang, menghabiskan token API hingga mencapai batas percobaan ulang.
- **Delegasi melingkar**: Dalam sistem multi-agen, agen dapat mendelegasikan tugas bolak-balik tanpa membuat kemajuan. Misalnya, Agen A meminta bantuan Agen B, Agen B menentukan Agen A lebih cocok, dan mereka berulang tanpa henti. Perhatikan bahwa

ini bukan masalah dengan delegasi hierarkis, di mana agen induk selalu mendelegasikan ke agen anak.

- **Kehabisan sumber daya**: Agen yang berjalan tak terkendali dapat mengonsumsi token API yang berlebihan, melakukan ribuan panggilan alat, atau menghasilkan sejumlah besar data log, yang menyebabkan pembengkakan biaya dan ketidakstabilan sistem.

  Sebagai contoh, bug dalam loop pemeriksaan kesehatan MAKDO menyebabkannya memulai pemeriksaan kesehatan baru setiap detik, bukan setiap 60 detik. Setiap pemeriksaan memanggil Analyzer, yang melakukan beberapa permintaan k8s-ai. Dalam hitungan menit, MAKDO melakukan ribuan panggilan LLM, menganalisis status klaster yang sama berulang kali. Tagihan API OpenAI meroket dan log server k8s-ai memenuhi disk.

Berikut beberapa strategi mitigasi untuk mencegah loop tak terbatas dan agen yang berjalan tak terkendali:

- Terapkan batasan iterasi maksimum dengan penundaan eksponensial
- Lacak riwayat tindakan agen dan deteksi pola berulang
- Gunakan pemutus sirkuit yang menghentikan eksekusi setelah kegagalan beruntun
- Pantau konsumsi sumber daya dengan peringatan dan pemutusan otomatis
- Terapkan kriteria penghentian eksplisit dalam perintah agen
- Tambahkan anggaran di mana agen hanya dapat menghabiskan hingga jumlah yang wajar

  Memanfaatkan strategi ini dapat memastikan bahwa agen tunggal tidak terjebak. Dalam sistem multi-agen, meskipun setiap agen berfungsi dengan baik sendiri-sendiri, ada masalah sistemik yang terjadi ketika komunikasi atau koordinasi antar agen gagal.

Kerusakan Komunikasi Dan Koordinasi

Sistem multi-agen bergantung pada komunikasi yang efektif antar agen. Ketika komunikasi ini rusak, seluruh sistem dapat gagal. Poin-poin berikut menjelaskan bagaimana hal itu terjadi:

- **Ketidaksesuaian format pesan**: Agen dapat menghasilkan output dalam format yang tidak dapat diuraikan oleh agen lain. Misalnya, satu agen mengeluarkan JSON sementara agen lain mengharapkan YAML, atau nama bidang tidak cocok antara pengirim dan penerima.
- **Konteks yang hilang dalam serah terima**: Ketika tugas didelegasikan antar agen, konteks penting dari interaksi sebelumnya dapat hilang. Misalnya, agen dukungan menyerahkan tugas kepada agen teknis tetapi lupa menyertakan detail akun pelanggan atau langkah-langkah pemecahan masalah sebelumnya.

  Contoh lain adalah koordinator memberi tahu Analyzer, “Periksa kesehatan klaster.” Analyzer menemukan tiga Pod yang gagal dan mengembalikan laporan terperinci dengan nama Pod, namespace, dan pesan kesalahan. Agen Koordinator kemudian memberi tahu agen Fixer, “Perbaiki masalahnya,” tanpa menyampaikan Pod spesifik mana yang perlu diperbaiki. Fixer harus memanggil k8s-ai lagi untuk menemukan kembali Pod yang gagal, menggandakan pekerjaan diagnostik dan membuang token.

- **Kondisi yang saling bertentangan**: Ketika beberapa agen bekerja secara bersamaan pada tugas-tugas terkait, mereka dapat saling mengganggu. Misalnya, dua agen penyebaran mungkin secara bersamaan memodifikasi konfigurasi yang sama, atau beberapa agen mungkin mengunci sumber daya yang dibutuhkan satu sama lain.
  Contoh lain adalah dua instance MAKDO yang berjalan secara bersamaan (mungkin dimulai secara tidak sengaja). Kedua koordinator memanggil agen Analyzer mereka untuk kluster yang sama. Kedua agen Analyzer mengidentifikasi Pod yang gagal yang sama. Kedua koordinator memberi tahu agen Fixer mereka untuk menghapusnya. Kedua agen Fixer memanggil k8s-ai untuk menghapus Pod. Yang pertama berhasil, sementara yang kedua mendapatkan "pod tidak ditemukan." Fixer kedua melaporkan kegagalan kepada koordinatornya, yang kemudian mengirimkan kesalahan ke Slack yang mengatakan "remediasi gagal," padahal sebenarnya, Pod berhasil dihapus oleh instance pertama.
- **Tujuan yang bertentangan**: Agen yang berbeda yang mengoptimalkan untuk tujuan yang berbeda dapat bekerja secara tidak tepat. Misalnya, agen optimasi biaya menurunkan skala sumber daya sementara agen kinerja meningkatkan skalanya.

Berikut beberapa strategi mitigasi untuk meningkatkan komunikasi dan koordinasi:

- Tetapkan skema pesan yang ketat dengan validasi
- Gunakan protokol serah terima terstruktur yang mencakup bidang konteks wajib
- Terapkan mekanisme koordinasi terdistribusi seperti kunci atau protokol konsensus
- Tetapkan hierarki agen yang jelas dan kebijakan penyelesaian konflik
- Pertahankan status bersama yang dapat diakses oleh semua agen

Sistem multi-agen yang kompleks seringkali perlu mengorbankan akurasi dan kualitas demi kecepatan dan biaya. Hal ini juga dapat memengaruhi agen tunggal, tetapi dalam sistem multi-agen, penundaan akan semakin parah. Jika respons terlalu lambat, mungkin sudah terlambat, atau pengalaman pengguna tidak menyenangkan. Mari kita lihat bagaimana kita dapat mengatasi masalah ini.

Masalah Waktu Habis Dan Latensi

Sistem agen di dunia nyata beroperasi di bawah batasan waktu, dan masalah latensi dapat menyebar ke seluruh sistem:

- **Waktu habis eksekusi alat**: Alat eksternal mungkin membutuhkan waktu lebih lama dari yang diharapkan, menyebabkan agen kehabisan waktu atau membuat keputusan berdasarkan informasi yang tidak lengkap. Misalnya, kueri basis data berjalan lambat, penyebaran kehabisan waktu, atau panggilan API macet.
- **Penundaan berantai**: Ketika agen saling bergantung, penundaan akan bertambah. Jika Agen A menunggu 30 detik untuk Agen B, yang menunggu 30 detik untuk Agen C, yang membutuhkan waktu 30 detik untuk merespons, pengguna akan mengalami waktu respons yang tidak dapat diterima, yaitu lebih dari 90 detik.
- **Agen yang macet**: Sebuah agen mungkin menunggu respons yang tidak akan pernah datang—webhook yang tidak pernah aktif, file yang tidak pernah dibuat, atau pesan yang hilang dalam perjalanan.

Berikut beberapa strategi mitigasi untuk menangani masalah timeout dan latensi:

- Tetapkan timeout yang realistis untuk setiap interaksi alat dan agen
- Jalankan tugas independen secara paralel untuk mengurangi latensi keseluruhan
- Terapkan pola asinkron di mana agen tidak terblokir pada operasi yang lambat
- Tambahkan indikator kemajuan dan streaming hasil parsial
- Rancang untuk penurunan kinerja yang lebih baik ketika operasi mengalami timeout
- Gunakan logika percobaan ulang yang sadar akan timeout dengan backoff
- Pertimbangkan untuk memberi stempel pada data yang diteruskan antar agen dengan TTL sehingga data yang usang diperbarui.

Oke, kita telah mengurangi masalah timeout dan latensi, tetapi ceritanya belum berakhir. LLM tidak memiliki status, tetapi sistem AI berbasis agen memiliki status. Mengelola status ini bukanlah hal yang sepele.

Manajemen Status Dan Kesalahan Konsistensi

Mempertahankan status yang konsisten di berbagai agen dan eksekusi alat merupakan tantangan:

- **Status usang**: Agen membuat keputusan berdasarkan informasi yang sudah ketinggalan zaman. Misalnya, agen memeriksa apakah penyebaran sudah siap. Awalnya berhasil, tetapi pada saat agen mengambil tindakan selanjutnya, penyebaran telah mengalami masalah.
- **Kegagalan sinkronisasi status**: Dalam sistem terdistribusi, agen yang berbeda mungkin memiliki pandangan yang berbeda tentang status saat ini. Misalnya, Agen A percaya bahwa suatu sumber daya ada sementara Agen B melihatnya sebagai telah dihapus.
- **Kegagalan sebagian**: Ketika operasi multi-langkah gagal di tengah jalan, sistem mungkin berada dalam keadaan yang tidak konsisten. Misalnya, setengah dari infrastruktur telah disediakan, atau beberapa konfigurasi telah diterapkan, sementara yang lain belum.
- **Kebocoran memori dan pembengkakan konteks**: Seiring percakapan menjadi lebih panjang, agen dapat mengakumulasi terlalu banyak konteks, yang menyebabkan penurunan kinerja atau melebihi batas token.

Berikut beberapa strategi mitigasi untuk meningkatkan manajemen dan konsistensi status:

- Terapkan pola seperti transaksi dengan kemampuan rollback
- Gunakan nomor versi atau stempel waktu untuk mendeteksi status usang
- Gunakan sistem manajemen status terdistribusi
- Terapkan rekonsiliasi status secara berkala
- Tetapkan batasan jendela konteks dan terapkan pemangkasan konteks yang cerdas
- Uraikan tugas-tugas yang rumit menjadi sub-tugas yang lebih kecil dan mudah dikelola yang dapat ditangani oleh agen terpisah

Kegagalan Keamanan Dan Izin

Sistem agen sering beroperasi dengan hak akses yang tinggi, sehingga kegagalan keamanan menjadi sangat berbahaya, misalnya dalam hal-hal berikut:

- **Peningkatan izin**: Agen dapat mencoba operasi yang seharusnya tidak dapat mereka akses, baik melalui kesalahan konfigurasi atau dengan mempelajari cara mengeksploitasi alat dengan cara yang tidak disengaja
- **Paparan kredensial**: Agen mungkin secara tidak sengaja mencatat, mengirimkan, atau mengekspos kredensial sensitif dalam pesan kesalahan atau keluaran alat
- **Eksploitasi input berbahaya**: Input pengguna yang dirancang dengan cermat dapat memanipulasi perilaku agen, menyebabkan mereka menjalankan operasi yang tidak disengaja atau melewati pemeriksaan keamanan
- **Kesenjangan jejak audit**: Tanpa pencatatan yang tepat, mungkin tidak mungkin untuk menentukan tindakan apa yang dilakukan agen, sehingga investigasi keamanan menjadi sulit

Berikut adalah beberapa strategi mitigasi untuk meningkatkan keamanan dan izin. Sebagian besar merupakan praktik terbaik umum untuk sistem apa pun, tetapi sangat penting dalam sistem berbasis agen:

- Terapkan prinsip hak akses minimal untuk semua akses alat
- Tambahkan pemeriksaan izin eksplisit sebelum operasi berisiko tinggi
- Bersihkan dan validasi semua input ke agen
- Jangan pernah menyertakan kredensial dalam perintah agen atau parameter alat
- Pertahankan log audit komprehensif dari semua tindakan agen
- Gunakan persetujuan manusia untuk operasi sensitif

Seperti yang Anda lihat, ada banyak cara sistem multi-agen dapat gagal. Dengan memahami mode kegagalan umum ini, Anda dapat merancang agen yang lebih tangguh, menerapkan pemantauan yang efektif, dan membangun strategi pemecahan masalah untuk mendiagnosis dan menyelesaikan masalah dengan cepat ketika muncul. Aspek penting lainnya adalah observabilitas.

10.3 INSTRUMENTASI DAN OBSERVABILITAS UNTUK AGEN AI

Observabilitas sangat penting untuk memahami apa yang terjadi di dalam sistem AI berbasis agen Anda. Tidak seperti perangkat lunak tradisional, di mana Anda dapat menelusuri kode dengan debugger, agen AI beroperasi melalui serangkaian panggilan LLM, eksekusi alat, dan transisi status yang sulit dilacak. Instrumentasi yang efektif mengubah sistem "kotak hitam" ini menjadi sistem "kotak kaca" di mana Anda dapat melihat dengan tepat apa yang terjadi di setiap langkahnya.

Sebagaimana dibahas dalam *From Black Box to Glass Box: LLM Observability in Action*, observabilitas yang komprehensif memerlukan pencatatan berbagai lapisan informasi. Lapisan-lapisan tersebut mencakup prompt yang dikirim ke LLM, respons yang diterima, alat yang dipanggil, langkah-langkah penalaran perantara, serta hasil akhir.

Pada bagian ini, kita akan mengeksplorasi praktik terbaik untuk menginstrumentasi sistem agen, termasuk primitif observabilitas inti, format pencatatan terstruktur, alat observabilitas LLM khusus, metrik kunci yang perlu dilacak, strategi pemantauan waktu nyata, teknik debugging, dan pertimbangan privasi.

Primitif Observabilitas Inti

Membangun sistem agen yang dapat diamati dimulai dengan menginstrumentasi operasi fundamental. Mari kita uraikan primitif observabilitas kunci yang harus Anda tangkap.

Pelacakan Panggilan LLM

Setiap interaksi dengan LLM harus dicatat dengan konteks lengkap, termasuk perintah lengkap, parameter model (suhu, token maksimum, dan lain-lain.), respons lengkap, penggunaan token, latensi, dan kesalahan apa pun. Ini menciptakan jejak audit dari semua keputusan AI.

Untuk MAKDO, Anda ingin menangkap setiap interaksi LLM di keempat agen. Ketika koordinator memutuskan untuk memanggil Analyzer, catat prompt koordinator (termasuk pesan sistem dan riwayat percakapan), model yang digunakan (GPT-4o), panggilan alat yang dihasilkannya (`agent_MAKDO_Analyzer` dengan pesan tertentu), jumlah token, dan latensi. Lakukan hal yang sama untuk interaksi Analyzer dengan LLM-nya ketika memproses permintaan. Ini menciptakan jejak keputusan lengkap yang menunjukkan mengapa MAKDO mengambil setiap tindakan.

Log Eksekusi Alat

Catat setiap pemanggilan alat dengan parameter input, durasi eksekusi, hasil output, dan status sukses/gagal. Sertakan jejak tumpukan untuk kesalahan dan konteks tentang mengapa alat tersebut dipanggil.

Di MAKDO, catat setiap panggilan A2A dari Analyzer dan Fixer ke k8s-ai. Catat nama skill (`kubernetes_resource_health`), parameter (`nama cluster, namespace`), respons lengkap dari k8s-ai, dan Catat juga waktu. Selain itu, catat juga panggilan alat MCP `Slack_Bot` (`slack_post_message` dengan saluran dan konten pesan). Saat melakukan debugging, Anda dapat merekonstruksi dengan tepat data apa yang dikirim setiap agen ke sistem eksternal dan respons apa yang mereka terima.

Cuplikan Status Agen

Secara berkala, tangkap status internal agen, termasuk memori kerja, riwayat percakapan, konteks yang terakumulasi, dan alasan pengambilan keputusan. Ini membantu memahami bagaimana status tersebut berkembang dari waktu ke waktu. Dalam kasus penggunaan komputer atau otomatisasi browser, tangkap tangkapan layar setelah setiap tindakan agen.

Hierarki Rentang Dan Jejak

Atur log ke dalam jejak hierarkis di mana permintaan pengguna membuat rentang induk, dan setiap tindakan agen (panggilan LLM, eksekusi alat, dan delegasi sub-agen) membuat rentang anak. Ini mencerminkan pelacakan terdistribusi dalam layanan mikro. Pemeriksaan kesehatan MAKDO menciptakan hierarki rentang ini:

> Pemeriksaan kesehatan (akar) → Panggilan LLM Koordinator → Panggilan alat agent_MAKDO_Analyzer → Panggilan LLM Analyzer → Panggilan A2A kubernetes_resource_health → Panggilan LLM Analyzer (memproses hasil) → Panggilan alat agent_MAKDO_Slack_Bot → Panggilan MCP slack_post_message

Setiap rentang memiliki waktu dan status. Jika notifikasi Slack gagal, Anda dapat menelusuri kembali melalui hierarki untuk melihat dengan tepat panggilan mana yang berhasil dan di mana kegagalan terjadi.

Pengayaan Metadata

Beri tag setiap operasi dengan metadata kontekstual, seperti ID pengguna, ID sesi, jenis agen, kategori tugas, dan lingkungan (dev/staging/prod). Ini memungkinkan pemfilteran dan analisis di berbagai dimensi.

Beri tag pada operasi MAKDO dengan cluster_name: "`kind-makdo-test`", `health_check_cycle: 47`, `agent_type`: "analyzer", dan `operation_type`: "`diagnostics`". Saat Anda perlu men-debug mengapa Analyzer terus salah mengidentifikasi Pod, lakukan kueri untuk semua operasi diagnostik Analyzer pada klaster spesifik tersebut dan periksa pola di beberapa siklus pemeriksaan kesehatan.

Format Pencatatan Dan Pelacakan Terstruktur

Log mentah sulit untuk dikueri dan dianalisis. Pencatatan terstruktur membuat data observabilitas dapat dikueri dan ditindaklanjuti. Ini adalah praktik terbaik untuk sistem tingkat produksi apa pun, tetapi sangat penting untuk sistem AI agenik yang kompleks. Sebagian besar bahasa memiliki pustaka yang matang untuk pencatatan terstruktur (misalnya, structlog untuk Python, Winston untuk Node.js, dan Logrus untuk Go). Manfaatkan pustaka ini untuk mengeluarkan log terstruktur secara konsisten. Berikut beberapa praktik yang direkomendasikan untuk diikuti:

- **Peristiwa terstruktur JSON**: Kirim log sebagai JSON terstruktur dengan skema yang konsisten, termasuk stempel waktu, ID jejak, ID rentang, ID rentang induk, jenis peristiwa, dan bidang khusus peristiwa. Ini memungkinkan kueri dan agregasi yang efisien.
- **Tipe peristiwa terstandarisasi**: Definisikan taksonomi peristiwa, seperti `agent.start`, `agent.complete`, `llm.request`, `llm.response`, `tool.invoke`, `tool.complete`, dan `error.occurred`. Tipe peristiwa yang konsisten memungkinkan analisis otomatis.
- **ID Korelasi**: Gunakan ID korelasi untuk menghubungkan operasi terkait di seluruh agen, layanan, dan sistem. Satu permintaan pengguna harus memiliki satu ID korelasi yang mengalir melalui seluruh sistem.
- **Pembuatan versi semantik untuk skema**: Saat skema observabilitas Anda berkembang, buat versinya sehingga Anda dapat mengurai log historis dan mempertahankan kompatibilitas mundur dalam alat analisis.

Alat Observabilitas Khusus LLM

Beberapa alat khusus telah muncul untuk mengamati sistem berbasis LLM:

- **Datadog LLM Observability**: Datadog menyediakan kemampuan observabilitas LLM khusus yang secara otomatis menangkap dan memvisualisasikan interaksi LLM. Ini melacak templat prompt, pemanggilan model, penggunaan token, metrik latensi, dan analisis biaya di beberapa penyedia LLM. Datadog LLM Observability terintegrasi dengan kerangka kerja populer seperti LangChain dan LlamaIndex, menyediakan instrumentasi siap pakai untuk aplikasi berbasis agen. Ini memungkinkan pelacakan

percakapan multi-giliran, alur kerja agen, dan pola penggunaan alat dengan perubahan kode minimal. Saya mengimplementasikan solusi observabilitas LLM menggunakan OpenTelemetry dan Datadog LLM Observability di Invisible Platforms, yang diakuisisi oleh Perplexity. Ini sangat berharga untuk mendiagnosis masalah rumit dan hambatan kinerja.

Fitur-fitur utama meliputi:

- Pengambilan prompt dan penyelesaian otomatis
- Pelacakan penggunaan token dan biaya per permintaan
- Persentil latensi dan analisis kinerja
- Pemantauan dan peringatan tingkat kesalahan
- Integrasi dengan APM dan pemantauan infrastruktur yang ada

- **Langsmith**: Platform observabilitas LangChain menyediakan jejak detail aplikasi LangChain, termasuk rantai penalaran agen, penggunaan alat, dan langkah-langkah pengambilan. Platform ini menawarkan antarmuka visual untuk menjelajahi jejak dan men-debug perilaku agen.
- **Weights h Biases (WB)**: Menawarkan pelacakan prompt, pemantauan kinerja model, dan perbandingan eksperimen untuk aplikasi LLM. Platform ini sangat berguna untuk melacak iterasi rekayasa prompt dan pengujian A/B konfigurasi agen yang berbeda.
- **Helicone**: Menyediakan pemantauan, caching, dan analitik real-time untuk API LLM. Melacak biaya, latensi, dan tingkat keberhasilan di berbagai penyedia LLM dengan antarmuka terpadu.
- **Kerangka kerja instrumentasi khusus**: Banyak tim membangun lapisan observabilitas khusus menggunakan OpenTelemetry sebagai fondasi, menambahkan konteks dan metadata khusus LLM ke pelacakan terdistribusi standar.

Metrik Dan Kpi Untuk Kinerja Agen

Anda tidak dapat mengelola apa yang tidak Anda ukur. Tetapkan metrik kuantitatif untuk melacak kesehatan dan kinerja agen, menetapkan garis dasar untuk operasi normal dan mendeteksi ketika terjadi kesalahan. Metrik ini memiliki banyak tujuan: membantu Anda memahami perilaku sistem, membenarkan investasi infrastruktur, mengoptimalkan biaya, dan membuktikan nilai kepada pemangku kepentingan.

Metrik tingkat keberhasilan memberi tahu Anda apakah agen Anda benar-benar menyelesaikan tugas yang diberikan. Penurunan tingkat penyelesaian tugas mungkin menunjukkan penyimpangan prompt, perubahan lingkungan, atau penurunan keandalan alat. Berikut adalah contoh metrik tingkat keberhasilan:

- Tingkat penyelesaian tugas (persentase tugas yang berhasil diselesaikan)
- Tingkat keberhasilan percobaan pertama (diselesaikan tanpa percobaan ulang)
- Tingkat keberhasilan panggilan alat (persentase pemanggilan alat yang berhasil)

Metrik kinerja mengungkapkan seberapa cepat sistem Anda merespons dan seberapa efisien agen beroperasi. Jumlah panggilan LLM yang tinggi per tugas mungkin menunjukkan pola penalaran yang tidak efisien atau percobaan ulang yang tidak perlu yang meningkatkan biaya dan latensi. Berikut adalah contoh metrik kinerja:

- Latensi ujung-ke-ujung (permintaan pengguna hingga respons akhir)
- Latensi panggilan LLM (persentil p50, p95, p99)
- Latensi eksekusi alat
- Jumlah panggilan LLM per tugas
- Jumlah panggilan alat per tugas

Metrik biaya sangat penting untuk sistem produksi di mana penggunaan token secara langsung diterjemahkan ke tagihan cloud. Lonjakan tiba-tiba dalam biaya per permintaan dapat menunjukkan agen yang tidak terkendali, perintah yang tidak efisien, atau agen yang menggunakan model yang lebih mahal daripada yang diperlukan. Berikut adalah contoh metrik biaya:

- Penggunaan token per permintaan (token perintah + token penyelesaian)
- Biaya per permintaan (berdasarkan harga model)
- Biaya per penyelesaian tugas yang berhasil
- Efisiensi biaya (penyelesaian yang berhasil / total biaya)

Metrik kualitas menilai apakah agen menghasilkan output yang benar dan bermanfaat. Metrik ini seringkali lebih sulit diukur secara otomatis, tetapi sangat penting untuk memahami kinerja dunia nyata di luar sekadar keberhasilan/kegagalan. Berikut adalah contoh metrik kualitas:

- Tingkat halusinasi (informasi palsu yang terdeteksi)
- Akurasi mengikuti instruksi
- Kepatuhan format output
- Skor kepuasan pengguna (jika tersedia)

Metrik kesehatan sistem memberikan sinyal peringatan dini tentang degradasi. Tingkat waktu habis yang meningkat mungkin mendahului kegagalan sistem total, memberi Anda waktu untuk menyelidiki dan memperbaiki sebelum pengguna sangat terpengaruh. Berikut adalah contoh metrik kesehatan sistem:

- Tingkat kesalahan berdasarkan jenis kesalahan
- Tingkat waktu habis
- Tingkat percobaan ulang
- Aktivasi pemutus sirkuit

Pemantauan Dan Peringatan Waktu Nyata

Pengamatan pasif tidak cukup, karena mengumpulkan metrik yang hanya Anda lihat setelah sesuatu rusak seperti memasang kamera keamanan dan hanya menonton rekamannya setelah perampokan. Anda membutuhkan pemantauan dan peringatan aktif yang menangkap masalah sebelum berdampak pada pengguna. Tujuannya adalah untuk beralih dari penanganan masalah reaktif ke deteksi masalah proaktif. Mari kita bahas beberapa cara terbaik untuk mendeteksi masalah ini:

- **Deteksi anomali** membantu Anda menangkap masalah yang tidak Anda antisipasi. Deteksi pola yang tidak biasa secara otomatis, seperti lonjakan tiba-tiba dalam tingkat kesalahan, peningkatan latensi yang tidak normal, atau pola penggunaan alat yang tidak terduga. Gunakan garis dasar statistik (seperti deviasi standar dari rata-rata) atau

model pembelajaran mesin untuk mengidentifikasi anomali. Ini sangat berharga untuk sistem agen di mana mode kegagalan dapat tidak dapat diprediksi dan bergantung pada konteks.

- **Peringatan berbasis ambang batas** adalah jaring pengaman Anda untuk kondisi kritis yang diketahui. Tetapkan peringatan untuk ambang batas kritis seperti tingkat kesalahan melebihi 5%, latensi rata-rata melebihi 10 detik, atau biaya per jam melebihi batas anggaran. Ini harus disetel dengan cermat untuk meminimalkan positif palsu sambil memastikan tidak ada insiden nyata yang lolos. Terlalu banyak peringatan yang berisik menyebabkan kelelahan peringatan, di mana tim mengabaikan pemberitahuan.
- **Analisis tren** mengungkapkan masalah yang bergerak lambat sebelum menjadi krisis. Lacak metrik dari waktu ke waktu untuk mengidentifikasi degradasi bertahap. Latensi rata-rata yang meningkat perlahan atau tingkat keberhasilan yang menurun mungkin mengindikasikan masalah mendasar sebelum menjadi kritis. Skenario "katak yang direbus" ini sangat berbahaya karena tidak ada satu titik data pun yang memicu peringatan, tetapi efek kumulatifnya secara signifikan menurunkan pengalaman pengguna.
- **Deteksi pembengkakan biaya** sangat penting untuk sistem berbasis LLM di mana biaya dapat meningkat tanpa terkendali. Pantau penggunaan token dan biaya secara real-time untuk mendeteksi agen yang membengkak sebelum menghabiskan seluruh anggaran Anda. Beri peringatan ketika pengeluaran per jam melebihi pola yang diharapkan, terutama selama jam-jam di luar jam sibuk ketika Anda mengharapkan penggunaan yang lebih rendah. Insiden produksi pada pukul 2 pagi yang menyebabkan perulangan tak terbatas dapat menghabiskan ribuan dolar sebelum ada yang menyadarinya tanpa pemantauan biaya yang tepat.
- **Integrasi dengan manajemen insiden** memastikan peringatan mencapai orang yang tepat pada waktu yang tepat. Hubungkan alat observabilitas ke sistem manajemen insiden seperti PagerDuty atau Opsgenie untuk peringatan dan eskalasi saat siaga. Tetapkan kebijakan eskalasi agar peringatan kritis tidak hilang di saluran Slack, dan pastikan selalu ada seseorang yang bertanggung jawab untuk menanggapi masalah produksi.

Sekarang, berbekal data observabilitas yang solid, mari kita lihat bagaimana cara men-debug sistem AI agen kita.

Debugging Dengan Data Observabilitas

Data observabilitas paling berharga ketika memungkinkan debugging yang efektif. Perbedaan antara sistem yang mengumpulkan data dan sistem yang memungkinkan debugging adalah apakah para insinyur dapat dengan cepat menjawab pertanyaan seperti, “Mengapa permintaan spesifik ini gagal?” atau “Apa yang berubah antara kemarin dan hari ini?” Data observabilitas yang baik seharusnya membuat analisis akar penyebab terasa seperti pekerjaan detektif dengan semua petunjuk yang telah disiapkan, bukan seperti mencari jarum di tumpukan jerami.

Pemutaran ulang jejak adalah mesin waktu Anda untuk memahami perilaku agen. Tangkap jejak lengkap sehingga Anda dapat memutar ulang eksekusi agen untuk memahami dengan tepat apa yang terjadi. Sertakan semua perintah, respons, panggilan alat, dan status perantara untuk merekonstruksi urutan peristiwa yang tepat. Ini sangat berharga saat men-debug kegagalan non-deterministik karena Anda dapat melihat perintah dan respons aktual yang menyebabkan kegagalan, bukan hanya asumsi Anda tentang apa yang seharusnya terjadi.

Pencarian kontekstual mengubah log mentah menjadi pengetahuan yang dapat dikueri. Aktifkan pencarian log berdasarkan permintaan pengguna, pesan kesalahan, jenis agen, atau bidang metadata apa pun. "Temukan semua penyebaran yang gagal dari Agen X dalam satu jam terakhir" seharusnya merupakan kueri sederhana, bukan latihan penguraian log yang kompleks. Pencatatan log terstruktur dengan metadata yang konsisten memungkinkan hal ini; investasikan waktu untuk melakukan instrumentasi dengan benar dan Anda akan menghemat waktu secara eksponensial lebih banyak selama proses debugging.

Analisis korelasi membantu Anda menemukan hubungan tersembunyi dalam data Anda. Analisis hubungan antar metrik: apakah peningkatan panjang prompt berkorelasi dengan tingkat kesalahan yang lebih tinggi? Apakah alat-alat tertentu memiliki latensi yang lebih tinggi daripada yang lain? Apakah waktu dalam sehari memengaruhi tingkat keberhasilan? Wawasan ini sering kali mengungkapkan peluang optimasi atau bug tersembunyi yang tidak terlihat jelas hanya dengan melihat metrik individual secara terpisah.

Debugging komparatif sangat ampuh untuk masalah yang terjadi sesekali. Bandingkan eksekusi yang berhasil dan gagal secara berdampingan untuk mengidentifikasi perbedaannya. Apa yang dilakukan agen yang berhasil yang tidak dilakukan agen yang gagal? Apakah mereka menggunakan alat yang berbeda, melakukan lebih sedikit panggilan LLM, atau memproses format input yang berbeda? Teknik ini sering mengungkapkan perbedaan kritis yang menentukan keberhasilan atau kegagalan.

Visualisasi deret waktu membuat pola yang tidak terlihat dalam data tabular menjadi terlihat. Visualisasikan metrik dari waktu ke waktu dengan alat seperti dasbor Grafana atau Datadog. Temukan pola, anomali, dan tren secara visual—manusia jauh lebih baik dalam mengenali pola visual daripada memindai kolom angka. Lonjakan latensi yang tiba-tiba atau peningkatan bertahap dalam tingkat kesalahan akan segera terlihat jelas dalam dasbor yang dirancang dengan baik.

Pertimbangan Privasi Dan Keamanan

Observabilitas komprehensif menimbulkan risiko kebocoran informasi sensitif jika tidak ditangani dengan benar. Pelanggaran data pada sistem observabilitas Anda dapat mengekspos data pelanggan, rahasia bisnis, atau kerentanan keamanan. Ketegangan antara visibilitas dan keamanan ini harus diseimbangkan dengan cermat melalui hal-hal berikut:

- **Penghapusan informasi identitas pribadi** (PII) harus otomatis dan diberlakukan pada lapisan instrumentasi, bukan diserahkan pada upaya manual. Hapus PII dari log secara otomatis. Nama pengguna, alamat, alamat email, nomor kartu kredit, nomor jaminan sosial, dan data sensitif lainnya tidak boleh muncul dalam log. Gunakan ekspresi reguler atau deteksi berbasis pembelajaran mesin untuk mengidentifikasi dan

menghapus PII sebelum log ditulis. Lebih baik lagi, rancang instrumentasi Anda agar tidak pernah menangkap PII sejak awal dengan hanya mencatat pengidentifikasi seperti ID pengguna yang dapat dikorelasikan dengan data pelanggan melalui pencarian yang diotorisasi dan diaudit.

- **Sanitasi prompt** memerlukan pertimbangan cermat tentang apa yang Anda catat. Berhati-hatilah dalam mencatat prompt lengkap jika berisi data pengguna atau informasi bisnis yang sensitif. Pertimbangkan untuk hanya mencatat versi yang telah disanitasi atau dianonimkan, atau mencatat prompt secara terpisah dengan kontrol akses yang lebih ketat. Untuk aplikasi keamanan tinggi, Anda dapat melakukan hashing pada prompt dan hanya menyimpan hash-nya, sehingga Anda dapat mendeteksi masalah duplikat tanpa mengekspos konten sebenarnya.
- **Kontrol akses** harus mengikuti prinsip hak akses minimal. Batasi akses ke data observabilitas berdasarkan peran, karena tidak semua orang boleh melihat log produksi yang berisi data pelanggan. Teknisi mungkin memerlukan akses untuk men-debug masalah, tetapi akses tersebut harus dicatat dan dapat diaudit. Pertimbangkan untuk menerapkan prosedur darurat untuk akses dengan pemberitahuan otomatis kepada tim keamanan. Beberapa organisasi memelihara sistem observabilitas terpisah untuk tingkatan keamanan yang berbeda.
- **Kebijakan retensi** harus menyeimbangkan kebutuhan yang saling bertentangan. Tentukan berapa lama data observabilitas akan disimpan, dengan menyeimbangkan kebutuhan debugging dengan biaya penyimpanan dan persyaratan kepatuhan. Data terbaru harus segera tersedia, tetapi data yang lebih lama dapat diarsipkan di penyimpanan yang lebih murah atau diagregasi untuk mengurangi volume. Banyak peraturan mensyaratkan periode retensi tertentu atau batas retensi maksimum, jadi pastikan kebijakan Anda sesuai dengan aturan yang berlaku.
- **Kepatuhan** bukanlah pilihan. Pastikan praktik observabilitas mematuhi peraturan seperti GDPR, HIPAA, SOC2, atau standar lainnya, tergantung pada domain dan yurisdiksi Anda. Ini termasuk persyaratan residensi data (di mana data dapat disimpan), perjanjian pemrosesan data dengan vendor observabilitas, dan hak penghapusan untuk data pelanggan. Dokumentasikan praktik observabilitas Anda sebagai bagian dari audit kepatuhan, dan tinjau secara berkala seiring perkembangan peraturan.

Dengan infrastruktur observabilitas yang kuat, Anda memiliki fondasi untuk debugging yang efektif. Sekarang mari kita bahas tantangan khusus dalam debugging penggunaan alat dan rantai panggilan fungsi, yang merupakan salah satu skenario debugging yang paling umum dan membuat frustrasi dalam sistem agen.

10.4 DEBUGGING PENGGUNAAN ALAT DAN RANTAI PANGGILAN FUNGSI

Penggunaan alat adalah tempat agen berinteraksi dengan dunia nyata, dan juga tempat di mana kesalahan paling sering terjadi. Agen mungkin memanggil alat yang salah, memberikan parameter yang tidak valid, salah menafsirkan hasil alat, atau menggabungkan alat secara tidak benar. Tidak seperti kesalahan penalaran murni yang tetap berada di dalam

LLM, kegagalan penggunaan alat memiliki konsekuensi nyata: penyebaran yang gagal, pembaruan basis data yang salah, atau integrasi yang rusak. Debugging masalah ini membutuhkan pemahaman tentang apa yang dimaksudkan agen dan apa yang sebenarnya terjadi.

Memahami Siklus Hidup Panggilan Alat

Mari kita tinjau kembali bagaimana sistem AI, kerangka kerja AI, agen, dan LLM berinteraksi dengan alat dan mempertimbangkan mode kegagalan. Setiap pemanggilan alat melalui siklus hidup yang dapat diprediksi, dan kegagalan dapat terjadi di setiap tahap:

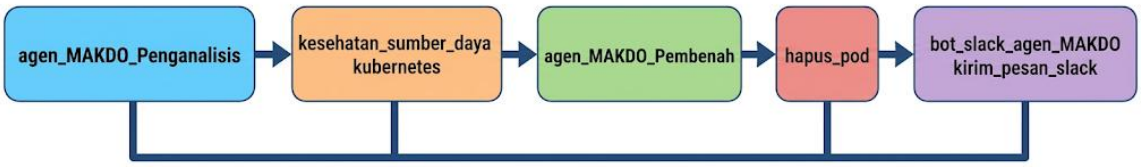
- Agen memutuskan alat mana yang akan dipanggil berdasarkan pemahamannya tentang tugas dan alat yang tersedia. Keputusan ini dapat salah jika deskripsi alat ambigu, jika agen salah memahami tugas, atau jika beberapa alat tampak sama-sama berlaku.
- Agen membangun parameter untuk panggilan alat, menerjemahkan maksud abstraknya menjadi argumen konkret. Konstruksi parameter gagal ketika agen kekurangan informasi yang diperlukan, salah memahami semantik parameter, atau menghasilkan nilai yang valid secara sintaksis tetapi salah secara semantik.
- Sistem AI host memvalidasi dan memanggil alat, memeriksa parameter terhadap skema dan mengeksekusi operasi sebenarnya. Validasi dapat menangkap kesalahan yang jelas, tetapi masalah semantik dapat lolos. Eksekusi alat itu sendiri mungkin gagal karena faktor eksternal: batas waktu API, kesalahan izin, atau ketidaktersediaan sumber daya.
- Alat mengembalikan hasil yang harus diinterpretasikan oleh agen. Bahkan panggilan alat yang berhasil pun dapat menyebabkan masalah jika agen salah menafsirkan hasil, mengabaikan peringatan penting, atau gagal memverifikasi bahwa operasi tersebut mencapai efek yang diinginkan.

Memahami siklus hidup ini membantu Anda menentukan di mana kegagalan terjadi. Panggilan alat yang tidak pernah terjadi menunjukkan masalah pengambilan keputusan. Panggilan alat dengan parameter yang tidak valid menunjukkan masalah konstruksi. Panggilan alat yang dieksekusi tetapi tidak mencapai tujuannya menunjukkan masalah interpretasi.

Melacak Rantai Panggilan Fungsi

Alur kerja agen multi-langkah menciptakan rantai panggilan alat di mana setiap agen memanggil agen berikutnya sebagai panggilan alat yang bergantung pada hasil sebelumnya. Rantai ini dapat dengan cepat menjadi kompleks:

- **Rantai linier** adalah yang paling sederhana: Alat A → Alat B → Alat C. Debugging rantai linier berarti memverifikasi setiap langkah dan memastikan konteks mengalir dengan benar antar langkah. Kerusakan di mana pun dalam rantai menyebabkan kegagalan di hilir.
- Alur kerja khas MAKDO adalah rantai linier.



Gambar 10.1 Alur Kerja Tipikal MAKDO

Jika notifikasi Slack tidak pernah sampai, lacak mundur: Apakah Slack_Bot dipanggil? Apakah ia menerima pesan yang benar? Apakah alat MCP dieksekusi? Apakah token API Slack valid? Setiap tautan dalam rantai harus berhasil agar alur kerja lengkap selesai.

- **Rantai kondisional** bercabang berdasarkan hasil sementara: Alat A → (jika berhasil: Alat B, jika gagal: Alat C). Agen harus menafsirkan hasil dengan benar dan memilih jalur yang tepat. Debugging membutuhkan pemahaman tentang logika percabangan dan apakah kondisi dievaluasi dengan benar. Ingat bahwa semua keputusan dibuat oleh LLM, jadi desain yang cepat sangat penting untuk menghasilkan logika yang benar.
  Koordinator membuat keputusan kondisional berdasarkan hasil Analyzer. Jika Analyzer melaporkan `CrashLoopBackOff`, panggil `Fixer`. Jika melaporkan ImagePullBackOff, lewati Fixer dan eskalasi ke manusia melalui Slack_Bot. Jika koordinator secara konsisten memilih cabang yang salah (memanggil Fixer untuk `ImagePullBackOff`), masalahnya terletak pada perintah sistemnya atau bagaimana ia menafsirkan struktur laporan Analyzer.
- **Rantai paralel** mengeksekusi beberapa alat secara bersamaan: (Alat A + Alat B + Alat C) → Alat D. Ini lebih efisien tetapi menimbulkan tantangan koordinasi. Apakah semua operasi paralel selesai? Apakah ada yang gagal? Bagaimana agen menggabungkan hasil sebelum melanjutkan ke Alat D?
- **Rantai ekrusif** melibatkan agen yang memanggil diri mereka sendiri sebagai panggilan alat (misalnya, sub-agen AI-6) atau membuat siklus: Alat A → Alat B → Alat A (dengan parameter yang berbeda). Ini bisa sangat ampuh tetapi rentan terhadap loop tak terbatas jika kondisi penghentian tidak dikelola dengan hati-hati.
  Saat melakukan debugging rantai, lacak mundur dari titik kegagalan. Panggilan alat mana yang gagal? Apa inputnya? Dari mana input tersebut berasal? Pendekatan arkeologis ini sering mengungkapkan bahwa akar penyebabnya terjadi beberapa langkah sebelumnya dalam rantai.

Mari kita lihat beberapa anti-pola penggunaan alat yang umum dan strategi debugging.

Anti-Pola Penggunaan Alat Yang Umum

Berikut adalah beberapa pola penyalahgunaan alat yang umum dalam sistem agen. Mengenali anti-pola ini membantu Anda dengan cepat mengidentifikasi kemungkinan akar penyebabnya:

- **Parameter zombie**: Agen menyertakan parameter yang diabaikan atau tidak dikenali oleh alat. Ini sering terjadi ketika API alat berubah tetapi perintah agen tidak diperbarui, atau ketika agen mengarang parameter berdasarkan alat serupa. Parameter

zombie berbahaya karena panggilan alat "berhasil" tetapi tidak berperilaku seperti yang diharapkan.

Misalnya, Fixer memanggil `delete_pod` k8s-ai dengan parameter tambahan: force: true. k8s-ai tidak mendukung parameter force (tidak diekspos melalui antarmuka A2A). k8s-ai mengabaikannya dan melakukan penghapusan normal. Fixer mengira telah melakukan penghapusan paksa tetapi sebenarnya telah melakukan penghapusan yang wajar. Saat melakukan debugging mengapa Pod membutuhkan waktu 30 detik untuk dihentikan, Anda menemukan bahwa flag force tidak berfungsi karena k8s-ai secara diam-diam mengabaikannya.

- **Optimasi prematur**: Agen mencoba melakukan terlalu banyak hal dalam satu panggilan alat, meneruskan struktur bersarang yang kompleks atau mencoba melakukan beberapa operasi sekaligus. Hal ini membuat debugging lebih sulit dan seringkali gagal karena alat tersebut tidak mendukung kasus penggunaan yang kompleks. Memecah operasi menjadi panggilan alat yang lebih kecil dan lebih sederhana biasanya lebih andal. Ini umum terjadi pada alat CLI canggih dengan sejumlah besar perintah dan opsi, seperti kubectl, git, atau bash (yang dapat menjalankan perintah lain yang diinstal pada host, termasuk menginstal perintah baru).

  Misalnya, Analyzer mencoba mendiagnosis semua Pod di semua namespace dengan satu panggilan k8s-ai, dengan meneruskan struktur kueri yang kompleks. k8s-ai mengembalikan hasil parsial atau mengalami timeout. Pendekatan yang lebih baik adalah dengan memanggil `kubernetes_resource_health` sekali per namespace dan menggabungkan hasilnya di Analyzer. Ini menghasilkan panggilan alat yang lebih sederhana, eksekusi yang lebih andal, dan debugging yang lebih mudah ketika satu namespace gagal.
- **Asumsi status**: Agen mengasumsikan sumber daya ada atau berada dalam status tertentu tanpa memverifikasinya. Agen mencoba memperbarui file yang tidak ada, melakukan deployment ke cluster yang sedang offline, atau melakukan query pada tabel database yang belum dibuat. Selalu verifikasi prasyarat sebelum mengeksekusi operasi jika memungkinkan.
- **Kesalahan yang diabaikan**: Agen menerima respons kesalahan dari suatu alat tetapi melanjutkan seolah-olah operasi berhasil. Ini terjadi ketika agen terlalu dioptimalkan untuk skenario sukses dan tidak menangani respons kesalahan dengan benar. Kegagalan akan berantai karena alat selanjutnya beroperasi pada data yang tidak ada atau berada dalam status tidak valid.
- **Kegilaan percobaan ulang**: Agen mencoba ulang operasi yang gagal berulang kali tanpa mengubah apa pun. Ini terkadang benar ketika menunggu operasi sebelumnya selesai. Namun, jika panggilan alat gagal karena parameter yang tidak valid, mencoba ulang dengan parameter yang sama akan gagal lagi. Percobaan ulang yang efektif memerlukan perubahan parameter, menunggu kondisi eksternal berubah, atau menggunakan pendekatan yang berbeda sama sekali.

Mari kita lihat strategi untuk men-debug dan mencegah masalah ini.

Strategi Debugging Untuk Penggunaan Alat

Ketika panggilan alat salah, kita membutuhkan pendekatan sistematis untuk men-debugnya. Strategi berikut efektif:

- **Putar ulang dengan inspeksi**: Tangkap panggilan alat yang tepat (nama, parameter, dan konteks) dan putar ulang secara terpisah. Apakah berhasil di luar agen? Jika ya, masalahnya adalah bagaimana agen memanggilnya. Jika tidak, masalahnya ada pada alat itu sendiri atau dependensinya.
- **Validasi parameter**: Verifikasi secara independen bahwa setiap parameter masuk akal. Jangan hanya memeriksa tipe—verifikasi semantik. Apakah jalur file ini valid? Apakah nama cluster ini ada? Apakah ID ini diformat dengan benar? Buat alat validasi yang dapat dipanggil agen sebelum mengeksekusi operasi berisiko atau tulis pembungkus dengan logika validasi dan ekspos pembungkus tersebut sebagai alat.
- **Verifikasi hasil**: Setelah panggilan alat berhasil, verifikasi bahwa operasi tersebut mencapai tujuannya. API mungkin mengembalikan "200 OK" tetapi sebenarnya tidak melakukan operasi karena kesalahan logika bisnis. Minta agen untuk secara eksplisit memeriksa bahwa sumber daya telah dibuat, file telah ditulis, atau konfigurasi telah diterapkan. Sekali lagi, ini dapat dilakukan dalam alat pembungkus yang melakukan verifikasi setelah memanggil alat yang sebenarnya.
- **Analisis perbedaan**: Bandingkan panggilan alat yang gagal dengan yang berhasil. Apa perbedaannya? Parameter? Waktu? Konteks? Faktor lingkungan? Seringkali, perbedaan tersebut mengungkapkan akar penyebabnya.
- **Pemantauan alat**: Terapkan alat pemantauan yang mencatat panggilan tanpa mengeksekusinya. Gunakan ini selama pengembangan untuk memverifikasi bahwa agen memanggil alat dengan benar sebelum mengizinkan operasi nyata. Ini sangat berharga untuk operasi destruktif seperti penghapusan atau penyebaran.

Untuk mengurangi kejadian kesalahan penggunaan alat, pertimbangkan untuk membangun pengaman ke dalam sistem AI agen Anda.

Membangun Pengaman Penggunaan Alat

Pencegahan lebih baik daripada debugging. Anda harus membangun pengaman yang menangkap kesalahan penggunaan alat sebelum menyebabkan masalah. Berikut beberapa pengaman yang efektif:

- **Validasi skema**: Terapkan skema parameter yang ketat dan tolak panggilan alat yang tidak valid sebelum eksekusi. Gunakan JSON Schema atau validasi serupa untuk menangkap kesalahan tipe, bidang wajib yang hilang, atau nilai di luar rentang yang diizinkan.
- **Mode uji coba**: Dukung mode uji coba atau pratinjau untuk semua operasi destruktif. Agen dapat memanggil alat dengan dry_run=true untuk melihat apa yang akan terjadi tanpa benar-benar melakukannya. Ini menangkap banyak kesalahan sebelum berdampak pada produksi.

- **Permintaan konfirmasi**: Untuk operasi berisiko tinggi, perlukan konfirmasi eksplisit. Agen menjelaskan apa yang akan dilakukannya dan harus mengkonfirmasi sebelum eksekusi. Ini menciptakan titik henti alami untuk tinjauan oleh manusia.
- **Pemeriksaan kompatibilitas alat**: Ketika agen menggabungkan beberapa alat, verifikasi bahwa format output sesuai dengan persyaratan input. Jika Alat A mengembalikan JSON tetapi Alat B mengharapkan YAML, tangkap ketidakkompatibilitas ini sebelum eksekusi.
- **Pembatasan waktu dan pemutus sirkuit**: Cegah penggunaan alat yang berlebihan dengan membatasi berapa kali alat dapat dipanggil dalam jangka waktu tertentu. Jika agen memanggil alat yang sama 50 kali dalam satu menit, ada yang salah. Ini harus menghentikan eksekusi dan memberi tahu manusia yang akan menyelidiki.

Kami telah membahas secara mendalam cara mendiagnosis, men-debug, dan membangun pengaman untuk mengatasi masalah penggunaan alat. Mari kita pertimbangkan bagaimana mendiagnosis kegagalan koordinasi, yang sering terjadi pada sistem multi-agen.

10.5 MENDIAGNOSIS KEGAGALAN KOORDINASI DALAM ALUR KERJA MULTI-AGEN

Sistem multi-agen memperoleh kekuatannya dari koordinasi. Beberapa agen bekerja sama untuk menyelesaikan tugas-tugas kompleks yang tidak dapat ditangani oleh satu agen saja. Namun, koordinasi ini memperkenalkan mode kegagalan baru yang sulit didiagnosis. Tidak seperti kegagalan agen tunggal yang memiliki akar penyebab yang jelas, kegagalan koordinasi sering muncul dari interaksi antar agen, sehingga lebih sulit untuk diisolasi dan di-debug.

Ketika agen tidak berkoordinasi dengan benar, gejalanya bisa samar: tugas yang sebagian selesai, pekerjaan yang duplikat, tindakan yang bertentangan, atau kebuntuan misterius di mana sistem tampak macet. Kegagalan ini tidak sesuai dengan kategori debugging tradisional karena masalahnya bukan pada komponen individual mana pun; melainkan pada bagaimana komponen berinteraksi.

Bagian ini memberikan pendekatan sistematis untuk mendiagnosis kegagalan koordinasi dalam alur kerja multi-agen, memahami di mana kerusakan terjadi, dan menerapkan strategi untuk mencegahnya.

Memahami Pola Kegagalan Koordinasi

Langkah pertama dalam mendiagnosis kegagalan koordinasi adalah mengenali kapan hal itu terjadi dan memahami pola interaksi sistem Anda. Kegagalan koordinasi bermanifestasi dalam pola karakteristik yang berbeda dari kegagalan agen tunggal. Mari kita bahas beberapa gejala kegagalan koordinasi yang umum.

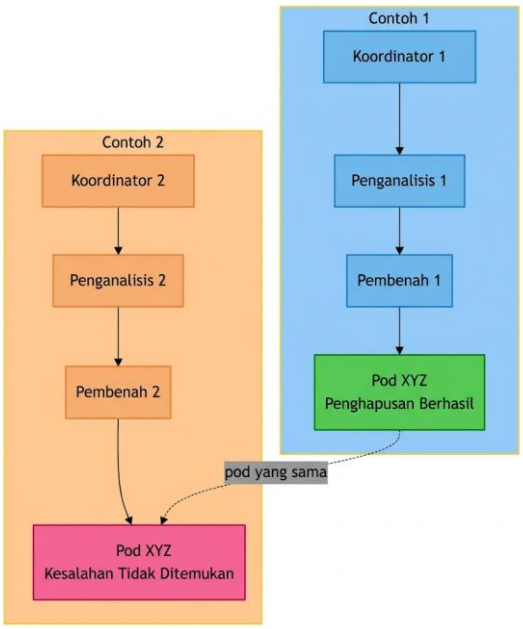
Pertama adalah **penyelesaian tugas sebagian**. Ini adalah salah satu sinyal kegagalan koordinasi yang paling jelas. Tugas secara keseluruhan tampaknya "selesai," tetapi langkah-langkah penting hilang. Katakanlah Agen A menyelesaikan bagiannya dan menyerahkannya kepada Agen B, tetapi Agen B tidak pernah menerima pekerjaan tersebut atau tidak mengerti apa yang harus dilakukan dengannya. Hasilnya adalah tugas yang 80% selesai tetapi terhenti. Pengguna melaporkan bahwa "ada sesuatu yang tidak berfungsi," tetapi tidak dapat

menjelaskan secara tepat apa itu. Agen individu berhasil dalam tugas-tugas sempit mereka sementara tujuan keseluruhan gagal. Gambar berikut mengilustrasikan konsep ini:

Gambar 10.2 Penyelesaian Tugas Sebagian Karena Alur Kerja Berhenti Pada 80% Ketika Agen D Tidak Pernah Memulai

Dalam MAKDO, agen koordinator memanggil agen Analyzer, mendapatkan laporan diagnostik, dan kemudian memanggil agen Fixer untuk melakukan remediasi. Fixer berhasil menghapus Pod yang gagal. Tetapi koordinator tidak pernah memanggil `Slack_Bot` untuk memberi tahu tim. Remediasi terjadi, klaster diperbaiki, tetapi manusia tidak memiliki visibilitas. Mereka memeriksa Slack beberapa jam kemudian dan tidak melihat pembaruan, tidak tahu apakah masalah telah teratasi, dan mungkin secara manual melakukan intervensi pada masalah yang sudah diperbaiki.

Pola kegagalan lainnya adalah **duplikasi pekerjaan**, yang terjadi ketika beberapa agen melakukan pekerjaan yang sama tanpa menyadari bahwa agen lain telah menanganinya. Misalnya, dua agen penyebaran sama-sama menyebarkan layanan yang sama, dua agen pemrosesan data sama-sama mengubah kumpulan data yang sama, atau dua agen layanan pelanggan sama-sama menanggapi tiket yang sama. Ini membuang sumber daya dan dapat menyebabkan kondisi persaingan atau keadaan yang tidak konsisten. Gejalanya sering terlihat dalam metrik: jumlah panggilan alat yang luar biasa tinggi, entri log duplikat, atau pengguna menerima beberapa tanggapan untuk satu permintaan. Lihat gambar berikut:



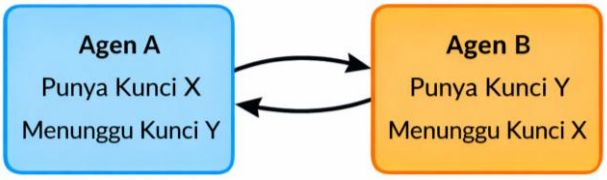
Gambar 10.3 Duplikasi Pekerjaan Karena Dua Instance MAKDO Memproses Pod Yang Sama Secara Bersamaan

Dua siklus pemeriksaan kesehatan MAKDO dimulai hampir bersamaan karena bug waktu. Kedua koordinator secara independen memanggil agen Analyzer mereka untuk memeriksa kluster yang sama. Kedua agen Analyzer mengidentifikasi tiga Pod yang gagal dan melaporkannya. Kedua koordinator kemudian memanggil agen Fixer mereka. Kedua agen Fixer mencoba menghapus Pod yang sama melalui k8s-ai. Yang pertama berhasil, sementara yang kedua mendapatkan kesalahan "pod tidak ditemukan". Keduanya mengirimkan laporan terpisah ke Slack, membingungkan tim dengan notifikasi duplikat dan hasil yang bertentangan.

Pola kegagalan utama lainnya adalah **tindakan yang bertentangan**, yang terjadi ketika agen bekerja dengan tujuan yang berbeda karena mereka tidak berbagi konteks atau tujuan. Satu agen meningkatkan infrastruktur sementara yang lain menurunkannya. Satu agen menutup tiket sementara yang lain meningkatkannya.

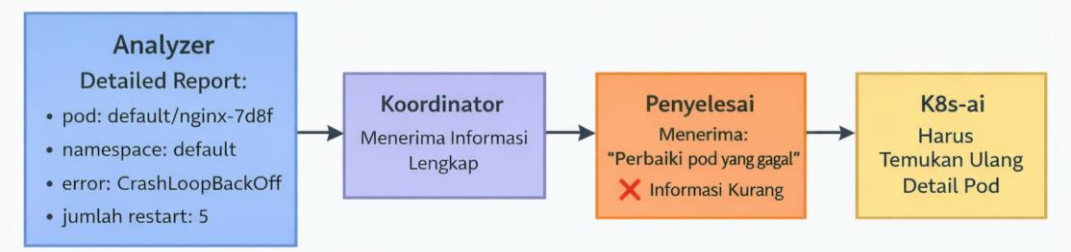
Satu agen menandai data untuk dihapus sementara yang lain mengarsipkannya. Sistem berputar-putar di antara keadaan yang bertentangan, berpotensi tidak pernah mencapai hasil yang diinginkan. Konflik-konflik ini sering muncul sebagai perubahan status yang sering terjadi dalam log audit atau sumber daya yang berosilasi antara konfigurasi.

Kebuntuan dan kemacetan adalah kegagalan koordinasi yang paling membuat frustrasi untuk didiagnosis. Sistem berhenti membuat kemajuan tanpa kesalahan yang jelas. Misalnya, Agen A sedang menunggu Agen B, sementara Agen B sedang menunggu Agen A, atau agen-agen menunggu sumber daya yang dikunci oleh agen lain, menciptakan ketergantungan siklik. Gejalanya adalah peningkatan metrik latensi dan tugas yang tidak pernah selesai, akhirnya habis waktu. Pengguna mengalami hal ini sebagai sistem yang "membeku" atau menjadi tidak responsif:



Gambar 10.4 Kebuntuan Dan Penantian Melingkar Di Mana Agen A Dan Agen B Saling Menunggu

Kehilangan konteks pada saat perpindahan antaragen muncul dalam bentuk agen yang meminta kembali informasi yang sebenarnya sudah diberikan, mengulangi pertanyaan, atau mengambil keputusan yang mengabaikan konteks penting dari tahap sebelumnya dalam alur kerja. Sebagai contoh, seorang pelanggan menjelaskan masalahnya kepada Agen A, kemudian Agen A meneruskan tugas tersebut kepada Agen B, tetapi Agen B meminta pelanggan untuk menjelaskan masalahnya kembali. Contoh lain, Agen A menemukan masalah konfigurasi, namun Agen B tidak mempertimbangkannya ketika memberikan rekomendasi. Pengalaman yang menurun seperti ini menimbulkan frustrasi bagi pengguna dan mengurangi tingkat keberhasilan penyelesaian tugas. Lihat gambar berikut.



Gambar 10.5 Hilangnya Konteks Dalam Serah Terima Karena Informasi Detail Dari Analyzer Hilang Ketika Koordinator Mendelegasikan Ke Fixer

Analyzer mengidentifikasi tiga Pod yang gagal dan memberikan informasi detail: nama Pod, namespace, pesan kesalahan, dan jumlah restart. Koordinator menerima laporan lengkap ini. Saat mendelegasikan ke Fixer, koordinator hanya mengatakan, "Perbaiki Pod yang gagal di klaster" tanpa memberikan nama Pod atau detail spesifik. Fixer harus memanggil k8s-ai lagi untuk menemukan kembali Pod mana yang gagal, membuang waktu dan token. Pekerjaan diagnostik terjadi dua kali karena konteks tidak dipertahankan dalam serah terima.

Terakhir, kita memiliki **masalah pengurutan pesan** yang terjadi ketika agen menerima pesan di luar urutan, yang menyebabkan keadaan bingung atau keputusan yang salah. Misalnya, Agen B memproses pesan "batalkan" sebelum memproses pesan "buat" asli, yang menyebabkan upaya untuk membatalkan sesuatu yang belum ada. Atau agen menerima hasil parsial sebelum hasil akhir, menyimpan data parsial dalam cache, dan membuat keputusan berdasarkan informasi yang tidak lengkap. Gejalanya meliputi kesalahan yang tidak dapat dijelaskan tentang sumber daya yang hilang atau inkonsistensi status yang tidak sesuai dengan log eksekusi.

Pola Interaksi Agen Dan Mode Kegagalannya

Untuk mendiagnosis kegagalan koordinasi, Anda perlu memahami bagaimana agen berinteraksi dalam sistem Anda. Pola interaksi yang berbeda memiliki mode kegagalan yang berbeda dan memerlukan pendekatan debugging yang berbeda.

Alur kerja sekuensial melibatkan agen yang meneruskan pekerjaan dari satu ke yang berikutnya dalam sebuah rantai: Agen A → Agen B → Agen C → Agen D. Setiap agen harus menyelesaikan pekerjaannya dan berhasil menyerahkan tugas ke agen berikutnya. Kegagalan biasanya terjadi pada batas penyerahan tugas. Misalnya, Agen B tidak menerima pekerjaan dari Agen A, atau Agen C menerima data yang tidak lengkap dari Agen B. Debugging memerlukan pemeriksaan protokol penyerahan tugas: data apa yang ditransfer, bagaimana formatnya, dan bagaimana agen penerima mengetahui bahwa pekerjaan telah tiba? Lacak eksekusi secara linier, identifikasi penyerahan tugas mana yang gagal dan mengapa.

Delegasi hierarkis melibatkan agen induk yang mendelegasikan sub-tugas ke agen anak yang terspesialisasi, kemudian menggabungkan hasilnya. Sebagai contoh, agen koordinator memecah "menerapkan aplikasi" menjadi sub-tugas seperti "menyediakan infrastruktur" (didelegasikan kepada Agen A), "mengkonfigurasi layanan" (didelegasikan kepada Agen B), dan "menjalankan uji coba fungsional" (didelegasikan kepada Agen C).

Agen induk harus menguraikan tugas dengan benar, menetapkan sub-tugas kepada anak-anak yang sesuai, melacak kemajuan mereka, dan mensintesis hasilnya. Kegagalan terjadi

ketika tugas ditugaskan kepada agen anak yang salah, ketika ketergantungan antar sub-tugas tidak dihormati, atau ketika agen induk salah menggabungkan hasil (memperlakukan kegagalan parsial sebagai keberhasilan).

MAKDO menggunakan delegasi hierarkis. Koordinator adalah induk, sedangkan Analyzer, Fixer, dan Slack_Bot adalah anak-anaknya. Koordinator harus mengurutkan pemanggilan mereka dengan benar: panggil Analyzer terlebih dahulu untuk mendiagnosis, kemudian Fixer untuk memperbaiki berdasarkan diagnosis, kemudian Slack_Bot untuk memberi tahu. Jika koordinator memanggil Fixer sebelum Analyzer, perbaikan terjadi tanpa diagnosis. Jika dipanggil secara paralel, Fixer mungkin mulai sebelum Analyzer selesai, sehingga kehilangan konteks penting tentang apa yang perlu diperbaiki.

Kolaborasi antar-agen melibatkan agen dengan status yang sama yang bekerja bersama, seringkali menegosiasikan siapa yang melakukan apa. Beberapa agen layanan pelanggan mungkin semuanya mampu menangani tiket, jadi mereka membutuhkan protokol untuk mencegah beberapa agen menanggapi permintaan yang sama. Debugging koordinasi peer-to-peer membutuhkan pemahaman tentang mekanisme konsensus: Bagaimana agen memutuskan siapa yang menangani apa? Pola umum meliputi pemilihan pemimpin (satu agen menjadi pemimpin dan menetapkan pekerjaan), koordinasi berbasis token (memegang token memberikan izin untuk bekerja), dan konkurensi optimis (agen mencoba pekerjaan dan menyelesaikan konflik jika terjadi). Telusuri protokol koordinasi untuk menemukan di mana kesepakatan gagal.

Koordinasi berbasis peristiwa melibatkan agen yang menanggapi peristiwa yang dipublikasikan ke bus peristiwa bersama atau antrian pesan. Misalnya, Agen A mempublikasikan peristiwa "pengguna terdaftar", Agen B berlangganan peristiwa tersebut dan mengirim email selamat datang, dan Agen C berlangganan dan membuat profil pelanggan. Pola ini terhubung secara longgar. Agen tidak saling mengetahui secara langsung, hanya tentang peristiwa. Kegagalan terjadi ketika peristiwa hilang, dikirim tidak berurutan, atau diproses beberapa kali. Debugging membutuhkan visibilitas ke dalam bus peristiwa: Apakah peristiwa tersebut dipublikasikan? Agen mana yang menerimanya? Apakah mereka memprosesnya dengan sukses? Kegagalan koordinasi berbasis peristiwa seringkali tidak meninggalkan jejak langsung karena agen penerbit tidak mengetahui apakah pelanggan telah memproses peristiwa tersebut. Observabilitas mendalam sangat penting.

Koordinasi status bersama melibatkan agen yang berkoordinasi melalui struktur data bersama seperti basis data, cache, atau penyimpanan status terdistribusi. Misalnya, Agen A menulis data ke basis data, dan Agen B membacanya dan membuat keputusan berdasarkan data tersebut. Pola ini membutuhkan perhatian cermat terhadap konsistensi, penguncian, dan batasan transaksi. Kegagalan terjadi karena kondisi persaingan (dua agen membaca sebelum salah satu menulis), pembacaan usang (Agen B membaca data lama yang telah diperbarui oleh Agen A), atau perebutan kunci (agen menunggu tanpa batas waktu untuk memperoleh kunci). Debugging membutuhkan pemeriksaan waktu pembacaan dan penulisan serta pemahaman tentang jaminan konsistensi sistem penyimpanan Anda.

Pastikan untuk menggambar diagram alur kerja multi-agen Anda yang menunjukkan pola mana yang Anda gunakan. Banyak sistem nyata menggabungkan pola, seperti delegasi hierarkis untuk dekomposisi tugas, koordinasi berbasis peristiwa untuk kopling longgar, dan status bersama untuk melacak kemajuan alur kerja secara keseluruhan. Memahami pola interaksi Anda sangat penting untuk debugging karena setiap pola memiliki mode kegagalan spesifik dan membutuhkan teknik diagnostik spesifik.

Melacak Dependensi Dan Aliran Data

Kegagalan koordinasi sering kali berasal dari dependensi yang rusak antara agen atau aliran data yang rusak. Bagian ini membahas teknik untuk melacak apa yang menjadi ketergantungan agen dan bagaimana informasi mengalir di antara mereka. Berikut adalah jenis dependensi antar-agen:

- **Dependensi eksplisit** adalah dependensi yang didefinisikan dalam kode atau konfigurasi. Misalnya, definisi tugas Agen B mencakup bidang `depends_on: [Agent_A]`, atau skema input Agen C membutuhkan output dari Agen B. Ini relatif mudah untuk di-debug karena didokumentasikan. Periksa apakah grafik dependensi secara akurat mencerminkan dependensi aktual dalam alur kerja Anda.

  Apakah ada dependensi yang hilang? Ketergantungan melingkar yang menyebabkan kebuntuan? Ketergantungan yang merujuk pada agen atau sumber daya yang tidak ada?
- **Ketergantungan implisit** lebih berbahaya. Ini adalah ketergantungan yang ada dalam praktik tetapi tidak tercatat dalam sistem orkestrasi Anda. Misalnya, Agen B mengasumsikan tabel basis data tertentu ada, yang seharusnya dibuat oleh Agen A, tetapi tidak ada hubungan ketergantungan formal. Agen C mengasumsikan kunci API telah disediakan, yang ditangani oleh Agen D, tetapi Agen D berjalan secara asinkron dan mungkin belum selesai ketika Agen C dimulai.

  Ketergantungan implisit ini menyebabkan kegagalan yang sulit didiagnosis karena pemahaman sistem tentang ketergantungan tidak sesuai dengan kenyataan. Perbaikannya biasanya adalah membuat ketergantungan implisit menjadi eksplisit dengan menambahkan deklarasi ketergantungan yang tepat atau menyusun ulang alur kerja untuk menghilangkan asumsi tersebut.
- **Ketergantungan data** melibatkan agen yang membutuhkan data spesifik dari tahap sebelumnya. Agen B membutuhkan ID pelanggan yang dicari oleh Agen A, dan Agen C membutuhkan URL penyebaran yang dibuat oleh Agen B. Lacak aliran data melalui sistem Anda: Apakah setiap agen menerima data yang dibutuhkannya? Apakah data diubah atau difilter dengan cara yang menghilangkan informasi penting? Apakah ada bidang opsional yang dianggap wajib oleh agen, menyebabkan kegagalan ketika bidang tersebut tidak ada? Cari ketidaksesuaian skema, penanganan kesalahan yang hilang ketika data yang diharapkan tidak ada, atau validasi yang terlalu ketat yang menolak input yang valid.

  Di MAKDO, Fixer bergantung pada Analyzer yang menyediakan pengidentifikasi Pod tertentu. Jika Analyzer mengembalikan ringkasan yang tidak jelas (“beberapa pod

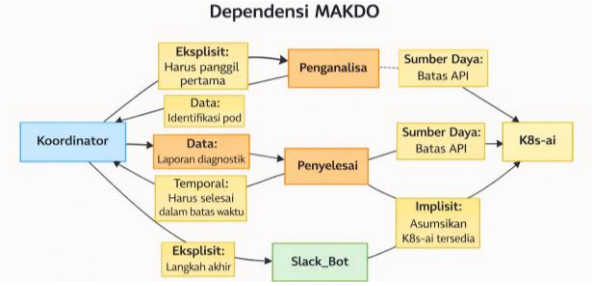
gagal") alih-alih nama yang tepat, Fixer tidak dapat bertindak. Ketergantungan data terputus.

Fixer mungkin mencoba memperbaiki berdasarkan informasi yang tidak lengkap, melewatkan beberapa Pod, atau harus memanggil k8s-ai lagi untuk mendapatkan detail yang dibutuhkannya.

- **Ketergantungan temporal** melibatkan batasan waktu. Misalnya, Agen B harus berjalan setelah Agen A selesai, tetapi juga sebelum jendela lima menit berakhir. Atau Agen C bergantung pada sistem eksternal yang berada dalam keadaan tertentu, yang berubah seiring waktu. Hal ini sangat rumit karena bergantung pada waktu eksekusi, sehingga kegagalan menjadi tidak deterministik. Alur kerja mungkin berhasil ketika agen dieksekusi dengan cepat, tetapi gagal ketika ada latensi jaringan atau perebutan sumber daya. Diagnosa ketergantungan temporal dengan menambahkan stempel waktu ke semua peristiwa koordinasi dan mencari pola: apakah kegagalan berkorelasi dengan peningkatan latensi? Apakah terjadi selama periode beban tinggi? Tambahkan penanganan batas waktu eksplisit dan pertimbangkan pola asinkron yang tidak mengasumsikan waktu yang ketat.
- **Ketergantungan sumber daya** melibatkan agen yang membutuhkan akses ke sumber daya bersama seperti batas laju API, koneksi basis data, instance komputasi, atau kunci pada status bersama. Jika beberapa agen bersaing untuk sumber daya yang terbatas, beberapa mungkin gagal atau mengalami penundaan. Lacak alokasi sumber daya: Apakah agen mendapatkan sumber daya yang mereka butuhkan? Apakah sumber daya dilepaskan dengan benar setelah digunakan? Apakah ada kebocoran sumber daya di mana agen menahan sumber daya tanpa batas waktu? Pantau metrik pemanfaatan sumber daya bersamaan dengan jejak eksekusi agen untuk mengkorelasikan kehabisan sumber daya dengan kegagalan koordinasi.

  Alat-alat seperti sistem pelacakan terdistribusi (OpenTelemetry, Jaeger, dan Zipkin) sangat berharga untuk memvisualisasikan ketergantungan. Alat-alat ini menunjukkan hubungan induk-anak antar rentang, memungkinkan Anda untuk melihat agen mana yang bergantung pada agen mana, berapa lama setiap langkah berlangsung, dan di mana kegagalan terjadi dalam grafik ketergantungan. Tanpa pelacakan terdistribusi, merekonstruksi ketergantungan dari log sangat melelahkan dan rawan kesalahan.

Berikut adalah gambar yang mengilustrasikan semua ketergantungan agen:



Gambar 10.6 Tipe Ketergantungan MAKDO Yang Menunjukkan Ketergantungan Eksplisit, Implisit, Data, Temporal, Dan Sumber Daya Antar Agen

Seperti yang telah kita bahas sebelumnya dalam bab ini, manajemen status dalam sistem multi-agen sangat menantang. Mari kita lihat bagaimana cara men-debug status bersama dan masalah sinkronisasi.

Mengelola Status Bersama Dan Mendeteksi Kebuntuan

Ketika beberapa agen berbagi status atau bersaing untuk mendapatkan sumber daya, koordinasi menjadi sangat menantang. Bagian ini membahas teknik untuk men-debug masalah sinkronisasi status dan mendeteksi kebuntuan.

Tantangan Sinkronisasi Status

Banyak sistem multi-agen mempertahankan status bersama yang dibaca dan diperbarui oleh banyak agen. Kegagalan terjadi ketika agen memiliki pandangan yang tidak konsisten tentang status ini atau ketika pembaruan bersamaan menyebabkan kerusakan. Berikut adalah teknik dan prinsip yang berguna yang dapat mengurangi dan terkadang mencegah kegagalan ini:

- **Model konsistensi** menentukan jaminan apa yang diberikan oleh penyimpanan status Anda. Konsistensi kuat berarti semua agen melihat status yang sama pada waktu yang sama. Misalnya, jika Agen A menulis suatu nilai, Agen B segera melihatnya. Konsistensi bertahap berarti penulisan menyebar secara asinkron. Agen B mungkin melihat status yang usang untuk jangka waktu tertentu. Sebagian besar basis data terdistribusi menawarkan konsistensi bertahap secara default untuk kinerja dan ketersediaan. Memahami model konsistensi Anda sangat penting untuk debugging: jika Anda mengasumsikan konsistensi kuat tetapi memiliki konsistensi bertahap, Anda akan melihat kegagalan misterius di mana agen membuat keputusan berdasarkan data yang usang.
- **Konsistensi baca-tulis-Anda** memastikan agen melihat pembaruan mereka sendiri segera, bahkan jika agen lain mungkin belum melihat pembaruan tersebut. Ini mencegah skenario yang membingungkan di mana agen menulis suatu nilai, lalu segera membaca kembali nilai yang berbeda. Verifikasi bahwa penyimpanan status Anda menyediakan konsistensi baca-tulis atau implementasikan pada lapisan aplikasi dengan melakukan caching pada penulisan terbaru.
- **Kontrol konkurensi optimis** mengasumsikan konflik jarang terjadi dan mendeteksinya ketika terjadi. Misalnya, agen membaca data dengan nomor versi, melakukan pembaruan, lalu menulis kembali dengan nomor versi tersebut. Jika agen lain memodifikasi data sementara itu (nomor versi berubah), penulisan gagal, dan agen harus mencoba lagi. Ini bekerja dengan baik untuk skenario persaingan rendah tetapi gagal dalam kondisi konkurensi tinggi. Diagnosa hal ini dengan memeriksa kegagalan penulisan berulang dan konflik versi dalam log. Perhatikan bahwa beberapa data, seperti penghitung atau akumulator, secara otomatis kebal terhadap konflik karena pembaruannya bersifat komutatif.
- **Penguncian pesimis** mencegah konflik dengan membuat agen memperoleh kunci sebelum mengakses status bersama. Misalnya, Agen A mengunci sebuah catatan, memperbaruinya, lalu melepaskan kunci tersebut. Ini mencegah konflik tetapi dapat

menyebabkan kebuntuan jika agen mengunci beberapa sumber daya dalam urutan yang berbeda, atau jika agen mengalami crash saat memegang kunci. Diagnosa masalah penguncian dengan melacak akuisisi dan pelepasan kunci, mencari kunci yang dipegang untuk jangka waktu yang tidak terduga atau tidak pernah dilepaskan. Manfaatkan kunci berbasis kedaluwarsa yang secara otomatis dilepaskan setelah waktu habis untuk mencegah penguncian permanen.

Pola sinkronisasi status mencakup beberapa pendekatan umum:

- **Pemilihan pemimpin**: Satu agen menjadi pemimpin dan mengelola status bersama, sementara agen lain membaca dari pemimpin
- **Konsensus terdistribusi**: Agen menggunakan protokol seperti Raft atau Paxos untuk menyepakati perubahan status
- **Pemisahan tanggung jawab perintah** (C&S): Agen terpisah untuk penulisan (perintah) dan pembacaan (kueri), dengan sinkronisasi asinkron
- **Sumber peristiwa**: Simpan status sebagai urutan peristiwa, agen membangun status dengan memutar ulang peristiwa

Setiap pola memiliki kompromi dalam kompleksitas, kinerja, dan jaminan konsistensi. Pilih berdasarkan kebutuhan Anda dan pastikan semua agen mengimplementasikan pola yang dipilih dengan benar.

Mendiagnosis kerusakan status memerlukan perbandingan antara status yang diharapkan (berdasarkan urutan operasi) dengan status aktual. Sistem dapat merekonstruksi apa yang seharusnya terjadi dengan memutar ulang log operasi, kemudian membandingkannya dengan status penyimpanan data aktual. Perbedaan menunjukkan adanya bug dalam logika pembaruan status atau kondisi persaingan di mana pembaruan bersamaan saling mengganggu. Gunakan log transaksi basis data atau log audit untuk melihat urutan penulisan yang tepat dan mengidentifikasi di mana kerusakan terjadi.

Mendeteksi Dan Menyelesaikan Kebuntuan

Kebuntuan adalah kegagalan koordinasi yang sangat buruk di mana agen menunggu satu sama lain tanpa batas waktu, tanpa membuat kemajuan. Sistem tampak macet tanpa kesalahan yang jelas. Berikut adalah jenis-jenis kebuntuan:

- **Kebuntuan klasik** terjadi ketika agen memegang sumber daya sambil menunggu sumber daya lain:
  - Agen A memegang kunci X dan menunggu kunci Y
  - Agen B memegang kunci Y dan menunggu kunci X
  - Keduanya tidak dapat melanjutkan

  Deteksi ini dengan melacak akuisisi kunci dan mencari pola tunggu melingkar. Atasi hal ini dengan memberlakukan urutan penguncian (semua agen memperoleh kunci dalam urutan yang sama) atau menggunakan batas waktu (jika kunci tidak diperoleh dalam N detik, lepaskan semua kunci dan coba lagi).
- Kebuntuan pesan terjadi dalam pola permintaan-respons:
  - Agen A mengirimkan permintaan ke Agen B dan menunggu respons
  - Agen B mengirimkan permintaan ke Agen A dan menunggu respons

- Keduanya tidak dapat merespons karena keduanya sedang menunggu

Cegah hal ini dengan menggunakan pola pesan asinkron alih-alih permintaan-respons sinkron, atau dengan memastikan agen dapat memproses permintaan masuk sambil menunggu respons (penangan non-pemblokiran).

Perhatikan bahwa desain interaksi sekuensial atau hierarkis (seperti MAKDO) kurang rentan terhadap kebuntuan pesan karena agen tidak saling menunggu dalam siklus. Pola peer-to-peer dan berbasis peristiwa lebih rentan.

- **Kebuntuan ketergantungan** terjadi dalam penjadwalan tugas:
  - Tugas A bergantung pada penyelesaian Tugas B
  - Tugas B bergantung pada penyelesaian Tugas A
  - Keduanya tidak dapat dimulai

  Anda dapat mendeteksi masalah ini dengan menganalisis grafik ketergantungan untuk siklus sebelum eksekusi. Atasi dengan menghilangkan ketergantungan siklik melalui desain ulang alur kerja.
- **Kebuntuan sumber daya** terjadi dengan sumber daya terbatas:
  - Agen A memegang sumber daya X dan menunggu sumber daya Y
  - Agen B memegang sumber daya Y dan menunggu sumber daya X
  - Keduanya menunggu tanpa batas waktu

  Anda dapat mendeteksi ini dengan melacak alokasi sumber daya dan mencari agen yang menunggu sumber daya sambil memegang sumber daya lain. Anda dapat menyelesaikan ini dengan akuisisi sumber daya berbasis waktu habis (jika Anda tidak dapat memperoleh semua sumber daya yang dibutuhkan dalam waktu habis, lepaskan apa yang Anda miliki dan coba lagi) atau dengan mengalokasikan terlebih dahulu semua sumber daya yang dibutuhkan sebelum memulai pekerjaan.

  Perhatikan bahwa ini berbeda dari kehabisan sumber daya, di mana agen gagal karena sumber daya tidak tersedia. Deadlock melibatkan penantian melingkar, sedangkan kehabisan melibatkan kurangnya ketersediaan.

Mendebug deadlock memerlukan pengambilan status semua agen ketika deadlock terjadi: Apa yang dilakukan setiap agen? Sumber daya apa yang dipegangnya? Apa yang ditunggunya? Thread dump (untuk agen berbasis thread) atau snapshot status (untuk agen berbasis event) memberikan informasi ini. Ketika Anda memvisualisasikan hubungan tunggu sebagai grafik terarah, deadlock muncul sebagai siklus.

Mendebug sistem multi-agen membutuhkan banyak upaya baik di awal dalam mengumpulkan data yang diperlukan untuk membantu Anda memecahkan masalah dan ketika masalah benar-benar terjadi. Jika memungkinkan, lebih baik berinvestasi dalam pencegahan sehingga Anda dapat menghilangkan seluruh kategori masalah dengan segera. Mari kita lihat bagaimana membangun pengaman koordinasi yang membantu Anda dengan pendekatan ini.

Membangun Pengaman Koordinasi

Pencegahan lebih baik daripada debugging. Bangun pengaman berikut ke dalam sistem multi-agen Anda untuk menangkap kesalahan koordinasi sebelum menyebabkan kegagalan:

- **Validasi ketergantungan** memeriksa grafik ketergantungan untuk masalah sebelum eksekusi. Apakah ada ketergantungan melingkar? Ketergantungan pada agen yang tidak ada? Ketergantungan yang dibutuhkan hilang? Validasi pada saat penerapan dan tolak definisi alur kerja yang tidak valid.
- **Penegakan batas waktu** memastikan tidak ada agen yang menunggu selamanya. Tetapkan batas waktu maksimum untuk semua komunikasi antar-agen, eksekusi alat, dan akuisisi sumber daya. Ketika batas waktu terjadi, gagal dengan baik dengan pesan kesalahan yang jelas daripada menggantung tanpa batas waktu. Protokol seperti percobaan ulang dengan penundaan eksponensial dan jittering membantu menghindari masalah "thundering herd" selama percobaan ulang.
- **Persyaratan idempotensi** memastikan agen dapat dengan aman mencoba kembali operasi. Tandai operasi mana yang idempoten dan mana yang tidak. Untuk operasi non-idempoten, terapkan mekanisme untuk mendeteksi dan melewati eksekusi duplikat.
- **Validasi status** secara berkala memeriksa apakah status bersama konsisten dan valid. Jalankan pekerjaan latar belakang yang memverifikasi invarian: “jumlah semua saldo akun sama dengan total setoran dikurangi penarikan” atau “setiap tugas yang ditugaskan ke agen merujuk pada ID agen yang ada.” Beri peringatan ketika invarian dilanggar. Bangun kemampuan untuk memperbaiki inkonsistensi secara otomatis jika memungkinkan.
- **Pemutus sirkuit** mencegah kegagalan berantai. Misalnya, jika Agen B terus-menerus gagal, Agen A harus berhenti memanggilnya daripada menghasilkan lebih banyak kegagalan. Terapkan pola pemutus sirkuit yang mendeteksi tingkat kegagalan dan menghentikan sementara komunikasi antar agen ketika satu agen tidak sehat. Jika Anda bertujuan untuk ketersediaan tinggi, miliki agen cadangan yang dapat mengambil alih ketika agen utama gagal.
- **Pengujian koordinasi** memvalidasi bahwa agen berinteraksi dengan benar. Tulis pengujian integrasi yang menguji alur kerja multi-agen, termasuk kasus-kasus ekstrem seperti berikut:
  - Apa yang terjadi jika Agen B lambat atau tidak responsif?
  - Apa yang terjadi jika Agen A mengirimkan data yang salah format ke Agen B?
  - Apa yang terjadi jika pesan dikirim tidak berurutan?
  - Apa yang terjadi jika dua agen mencoba memperbarui sumber daya yang sama secara bersamaan?

Pengamanan ini menangkap kesalahan koordinasi sebelum mencapai produksi. Bahkan dengan pengamanan terbaik sekalipun, beberapa kegagalan tidak dapat dihindari. Tetapi sistem secara keseluruhan harus mampu pulih dan terus beroperasi. Ini membutuhkan pilihan desain yang eksplisit. Mari kita lihat apa yang dibutuhkan.

10.6 MERANCANG UNTUK KETAHANAN DAN KEMAMPUAN PEMULIHAN

Sistem AI multi-agen skala besar harus dirancang untuk ketahanan dan kemampuan pemulihan. Kegagalan tidak dapat dihindari, jadi sistem harus menangani kesalahan dengan baik, pulih dari kegagalan, dan terus beroperasi dengan gangguan minimal. Selain tantangan sistem terdistribusi standar, yang diketahui jauh dari mudah dipecahkan, sistem multi-agen memperkenalkan kompleksitas baru karena interaksi agen, status bersama, dan ketergantungan alat. Seluruh ekosistem harus tangguh, bukan hanya komponen individual. Mari kita lihat strategi utama untuk membangun sistem AI multi-agen yang tangguh. Kami telah menyebutkan beberapa di antaranya sebelumnya, tetapi sekarang kami akan membahasnya lebih mendalam.

Redundansi Dan Mekanisme Failover

Redundansi adalah prinsip inti dari ketahanan. Komponen kritis harus memiliki instance redundan yang dapat mengambil alih jika salah satunya gagal. Dalam konteks sistem AI multi-agen, salah satu aspek terpenting adalah penyedia LLM. Mengandalkan satu penyedia LLM menciptakan satu titik kegagalan. Jika penyedia tersebut mengalami downtime, pembatasan laju, atau penurunan kinerja, seluruh sistem Anda akan menderita. Untuk mengurangi risiko ini, terapkan strategi multi-penyedia.

Rancang agen Anda agar dapat beralih antara beberapa penyedia LLM (misalnya, OpenAI, Anthropic, dan Google Gemini) berdasarkan ketersediaan, biaya, atau kinerja. Abstraksikan antarmuka LLM sehingga agen dapat menggunakan penyedia mana pun secara bergantian.

Dimensi redundansi lainnya adalah model yang digunakan agen Anda. Jika agen dibangun di sekitar model tertentu dengan prompt yang sangat disetel dan mengandalkan alat bawaan, maka agen tersebut mungkin kesulitan untuk beralih model. Rancang agen agar sebisa mungkin tidak bergantung pada model tertentu, sehingga memungkinkan mereka untuk menggunakan model yang berbeda dengan perubahan minimal. Ini mungkin melibatkan standarisasi perintah, antarmuka alat, dan penguraian respons. Jika Anda benar-benar perlu mengoptimalkan ke model tertentu, sediakan model cadangan yang dapat digunakan dalam keadaan darurat, meskipun model tersebut tidak seoptimal model utama.

Berikan perhatian khusus pada alat yang tersedia untuk agen Anda. Beberapa model, seperti GPT-5, dilengkapi dengan alat bawaan (misalnya, penerjemah kode atau peramban web) yang terintegrasi erat. Jika agen Anda bergantung pada alat-alat ini, pastikan alat alternatif tersedia saat beralih model. Misalnya, jika agen menggunakan alat penerjemah kode yang tidak tersedia, sediakan mekanisme cadangan untuk menggunakan layanan eksekusi kode eksternal.

Saat berinteraksi dengan layanan eksternal melalui alat (misalnya, basis data atau API), terapkan strategi failover. Jika satu titik akhir layanan tidak dapat dijangkau, sediakan titik akhir cadangan atau layanan alternatif yang dapat digunakan. Gunakan penyeimbangan beban berbasis DNS atau penemuan layanan untuk mengarahkan permintaan ke instance yang sehat. Idealnya, alat itu sendiri dapat menerapkan logika percobaan ulang dan failover secara implisit, sehingga agen tidak perlu menanganinya secara langsung. Tetapi jika tidak memungkinkan

(misalnya, saat menggunakan alat MCP pihak ketiga), Anda perlu membuat alat pembungkus yang menerapkan failover atau meminta agen menanganinya dengan mengekspos ini sebagai lapisan alat alternatif dan menginstruksikan agen untuk mengelola failover.

Strategi Degradasi Yang Anggun

Ketika terjadi kegagalan, sistem harus mengalami degradasi secara anggun daripada gagal total. Degradasi yang anggun berarti mengurangi fungsionalitas sambil mempertahankan operasi inti. Sistem multi-agen dapat menonaktifkan fitur yang tidak penting, mengurangi kualitas, atau memperlambat pemrosesan daripada mematikan sepenuhnya. Kuncinya adalah memprioritaskan alur kerja kritis dan mengorbankan alur kerja yang kurang penting ketika sumber daya terbatas atau komponen gagal.

Tetapkan tingkatan degradasi untuk sistem multi-agen Anda. Misalnya, Tingkat 1 adalah fungsionalitas penuh dengan semua fitur yang berfungsi. Tingkat 2 mungkin menonaktifkan analitik atau pencatatan untuk mengurangi beban. Tingkat 3 dapat beralih ke model yang lebih sederhana dan lebih cepat yang kurang akurat tetapi lebih andal. Tingkat 4 mungkin menonaktifkan remediasi otomatis dan hanya melakukan diagnostik. Setiap tingkatan mempertahankan nilai tertentu sambil mengonsumsi lebih sedikit sumber daya atau bergantung pada lebih sedikit komponen. Rancang agen untuk mendeteksi tingkatan mana yang sedang mereka operasikan dan sesuaikan perilakunya. Untuk MAKDO, model degradasi mungkin terlihat seperti berikut:

- Tingkat 1 menjalankan pemeriksaan kesehatan penuh dengan analisis terperinci dan perbaikan otomatis
- Tingkat 2 melewati perbaikan otomatis dan hanya melaporkan masalah ke Slack untuk intervensi manusia
- Tingkat 3 melakukan pemeriksaan kesehatan dasar tanpa diagnostik terperinci
- Tingkat 4 menghentikan pemeriksaan kesehatan sepenuhnya dan hanya menanggapi permintaan pengguna yang eksplisit

Implementasikan fitur flag atau kill switch yang memungkinkan operator untuk menonaktifkan fungsionalitas tertentu selama insiden. Jika agen Fixer mulai membuat keputusan yang salah, saklar pemutus (kill switch) dapat menonaktifkan remediasi otomatis sambil tetap menjalankan diagnostik dan notifikasi. Jika k8s-ai menjadi lambat atau tidak andal, kembali ke perintah kubectl yang lebih sederhana untuk operasi dasar. Jika Slack sedang down, simpan notifikasi sementara dan kirimkan saat layanan dipulihkan, atau kirimkan peringatan melalui email sebagai saluran cadangan. Sistem beradaptasi dengan apa yang tersedia daripada gagal karena satu komponen tidak tersedia.

Gambar 10.7 Tingkat Degradasi MAKDO Dari Fungsionalitas Penuh Ke Mode Manual

Terkadang, kegagalan bersifat sementara, dan alih-alih mengalami degradasi secara bertahap, Anda hanya perlu mencoba lagi. Tetapi ada seni dan ilmu dalam mengklasifikasikan dan menangani kegagalan.

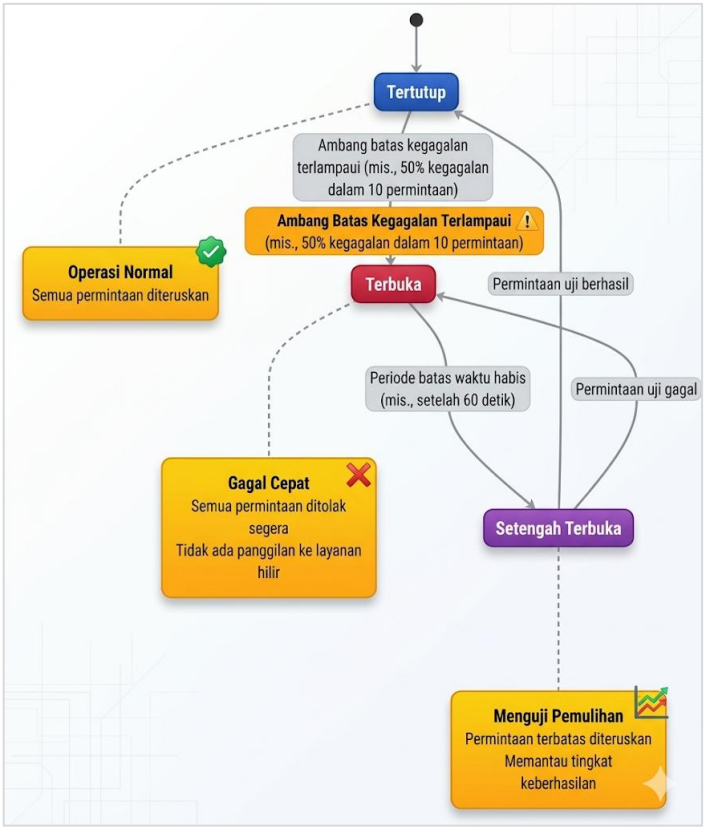
Logika Percobaan Ulang Dan Pemutus Sirkuit

Kegagalan sementara sering terjadi pada sistem terdistribusi. Gangguan jaringan, batasan laju API sementara, atau kelebihan beban layanan singkat dapat menyebabkan operasi gagal bahkan ketika sistem yang mendasarinya sehat. Logika percobaan ulang menangani kegagalan sementara ini dengan secara otomatis mencoba kembali operasi yang gagal setelah penundaan. Kuncinya adalah membedakan antara kegagalan sementara yang layak dicoba kembali dan kegagalan permanen yang tidak akan pernah berhasil. Mencoba kembali timeout jaringan masuk akal. Mencoba kembali kunci API yang tidak valid tidak. Rancang logika percobaan ulang untuk mengidentifikasi kesalahan yang dapat dicoba kembali dan segera menyerah pada kegagalan permanen.

Terapkan penundaan eksponensial dengan jitter untuk percobaan ulang. Mulailah dengan penundaan singkat (100 ms), lalu gandakan dengan setiap percobaan ulang (200 ms, 400 ms, 800 ms). Tambahkan jitter acak untuk mencegah masalah "thundering herd" di mana banyak agen mencoba kembali secara bersamaan dan membanjiri layanan yang gagal. Batasi penundaan maksimum (misalnya, 30 detik) dan batasi total upaya percobaan ulang (misalnya, lima kali percobaan). Catat setiap upaya percobaan ulang dengan konteks tentang mengapa upaya tersebut gagal sehingga Anda dapat mengidentifikasi pola. Untuk MAKDO, jika k8s-ai mengembalikan *503 Service Unavailable error*, coba lagi dengan penundaan eksponensial. Jika mengembalikan *401 Unauthorized error*, segera gagal karena mencoba lagi tidak akan memperbaiki kredensial yang tidak valid. Jika Fixer mencoba menghapus Pod dan mengalami batas waktu, coba lagi beberapa kali sebelum menyerah dan meningkatkan masalah ke manusia melalui Slack.

Pemutus sirkuit mencegah kegagalan berantai dengan menghentikan permintaan ke layanan yang tidak sehat. Pemutus sirkuit memantau tingkat kegagalan operasi ke layanan hilir. Ketika kegagalan melebihi ambang batas (misalnya, tingkat kegagalan 50% dari 10 permintaan), pemutus sirkuit terbuka dan segera menggagalkan semua permintaan tanpa mencoba memprosesnya. Ini memberi layanan yang gagal waktu untuk pulih tanpa dibombardir dengan permintaan. Setelah periode waktu tunggu (misalnya, 60 detik), pemutus sirkuit memasuki keadaan setengah terbuka dan memungkinkan beberapa permintaan uji coba untuk lewat. Jika berhasil, sirkuit akan tertutup dan operasi normal dilanjutkan. Jika gagal, sirkuit tetap terbuka. Pemutus sirkuit sangat penting untuk sistem multi-agen karena satu layanan yang lambat atau gagal dapat memblokir banyak agen yang menunggu respons.

Dalam MAKDO, jika k8s-ai menjadi tidak responsif dan setiap permintaan Analyzer mengalami waktu tunggu, pemutus sirkuit akan terbuka setelah beberapa kegagalan. Pemeriksaan kesehatan selanjutnya akan melewati pemanggilan k8s-ai sepenuhnya dan segera melaporkan status yang menurun ke Slack, daripada membuat setiap koordinator menunggu 30 detik untuk waktu tunggu. Ini mencegah seluruh sistem MAKDO berhenti total ketika satu dependensi gagal.



Gambar 10.8 Status Pemutus Sirkuit Beralih Antara Tertutup, Terbuka, Dan Setengah Terbuka Berdasarkan Tingkat Kegagalan Dan Waktu Habis

Ketika terjadi masalah, Anda mencoba lagi atau hanya berhenti sejenak dan melanjutkan nanti; Anda tidak ingin kehilangan pekerjaan penting yang terjadi sebelum kegagalan. Di sinilah checkpointing sangat berguna.

Checkpointing Dan Pemulihan Status

Alur kerja multi-agen dapat berjalan lama dan melibatkan banyak langkah mahal di berbagai agen. Jika agen mengalami crash di tengah alur kerja, sistem perlu pulih tanpa memulai dari awal. Perhatikan bahwa ini tidak diperlukan untuk alur kerja cepat yang selesai dalam hitungan detik, tetapi untuk alur kerja yang memakan waktu beberapa menit atau lebih, checkpointing seringkali sangat penting.

Checkpointing menyimpan status alur kerja pada titik-titik penting sehingga dapat dilanjutkan setelah kegagalan. Setiap checkpoint mencatat pekerjaan apa yang telah diselesaikan, data apa yang telah dihasilkan, dan apa yang masih harus dilakukan. Ketika agen pulih dari crash, ia memuat checkpoint terbaru dan melanjutkan dari sana daripada mengulang pekerjaan yang sudah berhasil.

Rancang titik pemeriksaan pada batas alami dalam alur kerja Anda. Setelah setiap agen menyelesaikan pekerjaannya, dan sebelum mendelegasikan ke agen berikutnya, simpan titik pemeriksaan. Simpan titik pemeriksaan dalam penyimpanan yang tahan lama, seperti basis data atau penyimpanan key-value terdistribusi, bukan di memori agen. Sertakan konteks yang cukup dalam titik pemeriksaan untuk melanjutkan pekerjaan: parameter input, hasil sementara, agen mana yang telah berjalan, dan langkah mana yang tersisa. Tambahkan metadata seperti stempel waktu, ID alur kerja, dan ID korelasi untuk melacak titik pemeriksaan di seluruh sistem terdistribusi. Untuk MAKDO, titik pemeriksaan berada setelah Analyzer selesai dan mengembalikan laporan diagnostiknya. Jika koordinator mengalami crash sebelum memanggil Fixer, koordinator pengganti dapat memuat titik pemeriksaan, melihat bahwa analisis telah selesai, dan langsung melanjutkan ke remediasi. Tanpa titik pemeriksaan, seluruh pemeriksaan kesehatan akan dimulai ulang, membuang pekerjaan Analyzer dan menggandakan waktu untuk memperbaiki masalah. Fixer juga harus membuat titik pemeriksaan setelah menyelesaikan setiap langkah remediasi, sehingga jika mengalami crash dan dimulai ulang, ia dapat melanjutkan dari tempat ia berhenti. `Slack_Bot` tidak memerlukan checkpointing karena skenario terburuknya adalah ia akan mengirim ulang notifikasi.

Implementasikan pemulihan idempoten untuk menangani kegagalan parsial dengan aman. Saat melanjutkan dari checkpoint, sistem mungkin mengeksekusi ulang beberapa operasi yang sebagian telah selesai sebelum crash. Rancang operasi agar idempoten sehingga eksekusi ulang menghasilkan hasil yang sama. Jika Fixer menghapus Pod tetapi mengalami crash sebelum memperbarui checkpoint, pemulihan mungkin mencoba menghapus Pod yang sama lagi. k8s-ai harus menangani ini dengan baik dengan mengembalikan keberhasilan jika Pod sudah hilang. Simpan ID operasi atau token transaksi di checkpoint untuk mendeteksi dan melewati operasi duplikat. Bersihkan checkpoint lama secara berkala untuk menghindari pertumbuhan penyimpanan yang tidak terbatas. Setelah alur kerja berhasil diselesaikan, hapus checkpoint-nya. Simpan checkpoint alur kerja yang gagal lebih lama untuk debugging. Seimbangkan frekuensi checkpoint dengan overhead. Checkpointing setelah setiap panggilan alat menambah latensi dan biaya penyimpanan. Checkpointing hanya di awal dan akhir alur

kerja berarti lebih banyak pekerjaan yang hilang karena kegagalan. Temukan tingkat detail yang tepat untuk kebutuhan keandalan sistem dan biaya operasional Anda.

Pola Eskalasi Manusia

Tidak semua kegagalan dapat ditangani secara otomatis. Beberapa situasi memerlukan intervensi manusia dalam hal penilaian atau persetujuan. Sistem multi-agen membutuhkan pola eskalasi yang jelas yang melibatkan manusia pada waktu yang tepat. Kuncinya adalah melakukan eskalasi cukup dini sehingga manusia dapat mencegah kerusakan, tetapi tidak terlalu agresif sehingga mereka kewalahan. Ini berarti bahwa sistem harus cukup cerdas untuk meminta umpan balik manusia hanya untuk keputusan kritis, dan menghindari pemberitahuan atau permintaan izin untuk setiap tindakan. Misalnya, jika sistem harus menganalisis 100 dokumen dan meminta izin untuk membaca setiap dokumen, maka pemberi persetujuan manusia akan merasa ini membosankan dan proses keseluruhan akan lebih lambat (misalnya, jika pemberi persetujuan manusia pergi makan siang).

Rancang aturan eskalasi berdasarkan tingkat kepercayaan, penilaian risiko, dan tingkat keparahan kegagalan. Operasi dengan kepercayaan tinggi dan risiko rendah dapat berjalan secara otomatis. Operasi dengan kepercayaan rendah atau risiko tinggi harus menunggu persetujuan manusia. Tentukan pemicu eksplisit yang melibatkan manusia. Kesalahan yang tidak diketahui yang tidak dapat diklasifikasikan oleh agen memerlukan penilaian manusia. Skor kepercayaan rendah di bawah ambang batas menunjukkan ketidakpastian yang perlu ditinjau oleh manusia. Operasi yang memengaruhi sistem produksi atau sumber daya kritis memerlukan persetujuan sebelum dilanjutkan. Beberapa upaya percobaan ulang sering kali menunjukkan kegagalan permanen yang perlu diselidiki. Operasi mode terdegradasi berarti fungsionalitas yang berkurang yang perlu diketahui oleh tim. Untuk MAKDO, Analyzer mungkin mengidentifikasi masalah yang tidak diketahui cara memperbaikinya oleh Fixer, dan Fixer mungkin gagal dalam upaya memperbaiki beberapa masalah karena izin yang tidak mencukupi. Ketidaktersediaan k8s-ai berarti sistem tidak dapat melakukan diagnostik, dan manusia perlu mengetahuinya. Setiap pemberitahuan harus mencakup konteks yang kaya: apa yang gagal, mengapa gagal, apa yang dicoba, dan opsi apa yang tersedia. Koordinator harus mengumpulkan informasi dan memberi tahu agen `Slack_Bot` untuk memberi tahu manusia dengan item tindakan yang jelas sehingga tim tahu persis keputusan apa yang diperlukan.

Berikan manusia opsi yang dapat ditindaklanjuti, bukan hanya pemberitahuan. Alih-alih "Pemeriksaan kesehatan gagal," katakan, "3 pod di namespace default mengalami crash-looping. Opsi 1: Hapus dan mulai ulang (otomatis). Opsi 2: Selidiki log terlebih dahulu (manual). Opsi 3: Abaikan dan tunggu pemeriksaan kesehatan berikutnya." Sertakan tombol atau perintah yang memungkinkan manusia menyetujui tindakan langsung dari Slack tanpa beralih konteks. Eskalasi yang dibatasi waktu dengan tindakan default. Jika manusia tidak merespons dalam 15 menit, ambil tindakan default yang paling aman atau eskalasikan lebih lanjut ke insinyur yang siaga. Lacak pola eskalasi untuk mengidentifikasi masalah yang berulang. Jika jenis kegagalan yang sama selalu membutuhkan intervensi manusia, itu adalah sinyal untuk meningkatkan kemampuan agen atau menambahkan otomatisasi. Jika manusia selalu memilih opsi yang sama, pertimbangkan untuk menjadikannya default otomatis.

Tujuannya adalah peningkatan berkelanjutan, di mana lebih sedikit eskalasi yang dibutuhkan dari waktu ke waktu karena agen belajar menangani lebih banyak situasi secara otonom. Dengan demikian, kita sampai pada akhir pembahasan mendalam kita tentang pengujian, debugging, dan pemecahan masalah sistem AI multi-agen.

10.7 RINGKASAN

Dalam bab ini, kita telah mengeksplorasi tantangan unik dalam pengujian, debugging, dan pemecahan masalah sistem AI multi-agen. Kita telah membahas strategi untuk merancang pengujian efektif yang memvalidasi koordinasi antar-agen, melacak ketergantungan dan aliran data untuk mengidentifikasi kegagalan, mengelola status bersama dan mendeteksi kebuntuan, serta membangun pengaman untuk mencegah kesalahan koordinasi. Kita juga membahas perancangan untuk ketahanan dan pemulihan melalui redundansi, degradasi bertahap, logika percobaan ulang, checkpointing, dan pola eskalasi manusia.

Sistem multi-agen memperkenalkan kompleksitas baru di luar sistem perangkat lunak tradisional, tetapi dengan desain yang cermat dan alat yang kuat, tantangan ini dapat dikelola secara efektif untuk membangun sistem AI multi-agen yang andal dan mudah dipelihara. Jangan lupa bahwa Anda dapat menambahkan agen AI introspektif ke sistem Anda yang akan membantu memantau, melakukan debugging, dan bahkan memperbaiki masalah secara otomatis saat muncul. Pada bab selanjutnya, kita akan mengeksplorasi cara menerapkan dan mengoperasikan sistem AI multi-agen di lapangan.

BAB 11
MENERAPKAN SISTEM MULTI-AGEN

11.1 PENDAHULUAN

Pada bab sebelumnya, kita telah membahas pengujian, debugging, dan pemecahan masalah sistem multi-agen. Kita telah mengeksplorasi mode kegagalan umum seperti halusinasi dan penyalahgunaan alat, strategi instrumentasi dan observabilitas, teknik debugging untuk kegagalan koordinasi, dan perancangan ketahanan melalui redundansi dan degradasi bertahap. Kita menggunakan MAKDO sebagai contoh konkret untuk mengilustrasikan konsep-konsep ini.

Sekarang, saatnya untuk menerapkan MAKDO secara nyata. Namun, menjalankan sistem multi-agen di lingkungan produksi menghadirkan tantangan baru seputar arsitektur penerapan, penemuan layanan, jaringan, dan keamanan. Bagaimana Anda menerapkan agen di berbagai lingkungan? Bagaimana mereka berkomunikasi dengan aman? Bagaimana Anda mengelola kredensial dan kontrol akses?

Pada bab ini, kita akan menerapkan MAKDO dalam pengaturan yang realistis seperti di lingkungan produksi. Kita akan membuat dua klaster Kubernetes menggunakan **Kubernetes di Docker** (KinD). Klaster kontrol menjalankan MAKDO itu sendiri. Klaster pekerja adalah klaster target yang dipantau dan dikelola oleh MAKDO. K8s-ai (server agen-ke-agen) berjalan di kluster pekerja, memberikan MAKDO kemampuan untuk mendiagnosis dan memperbaiki masalah melalui protokol **Agen-ke-Agen** (A2A). Pengaturan ini mencerminkan skenario produksi nyata di mana sistem manajemen Anda berjalan terpisah dari beban kerja yang dikelolanya.

Kita akan membahas cara menyiapkan kedua kluster, menyebarkan komponen MAKDO ke kluster kontrol, menyebarkan k8s-ai ke kluster pekerja, mengkonfigurasi komunikasi lintas kluster yang aman, dan mengelola penemuan layanan dan kontrol akses. Pada akhirnya, Anda akan memiliki sistem multi-agen yang berfungsi penuh yang mengelola kluster Kubernetes terpisah, siap untuk mendeteksi dan memperbaiki masalah secara otomatis. Berikut poin-poin penting yang dibahas:

- Menyiapkan kluster KinD ganda untuk penyebaran multi-agen
- Menyebarkan MAKDO ke kluster kontrol
- Menyebarkan k8s-ai ke kluster pekerja
- Mengaktifkan komunikasi aman antar kluster

11.2 MENYIAPKAN KLUSTER KinD GANDA UNTUK PENYEBARAN MULTI-AGEN

Mari kita bangun penyebaran seperti produksi di mana MAKDO berjalan pada satu kluster dan mengelola kluster lain. Pemisahan ini sangat penting di lingkungan nyata untuk mengisolasi infrastruktur manajemen dari beban kerja produksi. Ini mengurangi dampak kerusakan dan memungkinkan sistem produksi untuk tetap beroperasi bahkan jika MAKDO tidak beroperasi. Selain itu, jika kluster produksi mengalami kehabisan sumber daya atau

masalah serupa, MAKDO akan dapat menganalisis dan berpotensi memperbaikinya karena tidak mengalami masalah yang sama.

Menginstal KinD Dan Prasyarat

Sebelum membuat klaster, Anda perlu menginstal Docker, kubectl, dan KinD. Daripada menjelaskan langkah-langkah instalasi untuk setiap alat (yang bervariasi tergantung platform dan berubah seiring waktu), kami menyediakan skrip verifikasi yang memeriksa apakah semuanya sudah siap. Pertama, klon repositori buku jika Anda belum melakukannya:

```
git clone git@github.com:PacktPublishing/Design-Multi-Agent-AI-Systems-using-MCP-and-A2A.git
```

Instal alat-alat yang diperlukan menggunakan dokumentasi resminya:

- Docker: https://docs.docker.com/get-docker/
- kubectl: https://kubernetes.io/docs/tasks/tools/
- KinD: https://kind.sigs.k8s.io/docs/user/quick-start/#installation

Setelah terinstal, verifikasi bahwa semuanya berfungsi dengan skrip pengecekan prasyarat kami:

```
cd ch11
chmod +x check-prerequisites.sh
./check-prerequisites.sh
```

Skrip ini memeriksa apakah Docker sudah terpasang dan berjalan, kubectl tersedia, KinD sudah terpasang (versi 0.20.0 atau yang lebih baru disarankan), dan ruang disk tersedia (membutuhkan ~10 GB untuk kedua klaster), alokasi memori Docker (membutuhkan setidaknya 4 GB), dan klaster KinD yang ada yang mungkin bertentangan dengan pengaturan kita.

Jika skrip melaporkan prasyarat yang hilang, instal prasyarat tersebut menggunakan tautan sebelumnya dan jalankan skrip lagi. Saat Anda melihat Semua prasyarat terpenuhi!, Anda siap untuk membuat klaster.

Membuat Konfigurasi Klaster Kontrol

Klaster kontrol menampung agen MAKDO. Klaster ini perlu dikonfigurasi untuk memungkinkan akses eksternal ke titik akhir API MAKDO. KinD menggunakan file konfigurasi YAML untuk mendefinisikan topologi klaster dan jaringan. Buat `control-klaster.yaml`:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: makdo-control
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "cluster-role=control"
```

```
extraPortMappings:
# Port for accessing MAKDO API/dashboard if needed
 - containerPort: 30080
   hostPort: 8080
   protocol: TCP
# Port for MAKDO health check endpoint
- containerPort: 30090
  hostPort: 9090
  protocol: TCP
```

Berikut adalah poin konfigurasi utama:

- `name: makdo-control` mengidentifikasi kluster ini. KinD menggunakan nama ini untuk kontainer Docker dan konteks kubectl.
- `node-labels`: Label membantu mengidentifikasi kluster tempat pod berjalan. Berguna untuk debugging dan pemantauan.
- `extraPortMappings`: Mengekspos port dari kluster ke mesin host Anda. Port `8080` dipetakan ke port kontainer `30080` (rentang NodePort di Kubernetes). Port `9090` dipetakan ke `30090` untuk pemeriksaan kesehatan. Pemetaan ini memungkinkan Anda mengakses layanan MAKDO dari mesin lokal Anda atau memungkinkan kluster pekerja untuk mencapai MAKDO, jika diperlukan.

Selanjutnya, buat kluster kontrol:

```
kind create cluster --config control-cluster.yaml
```

Proses ini memakan waktu 1–2 menit. KinD mengunduh image node Kubernetes (jika belum di-cache), membuat kontainer Docker, dan menginisialisasi Kubernetes. Setelah selesai, Anda akan melihat tampilan berikut:

```
Creating klaster "makdo-control" ...
 ✓ Ensuring node image (kindest/node:v1.33.1) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-makdo-control"
```

Verifikasi klaster:

```
kubectl cluster-info --context kind-makdo-control
kubectl get nodes --context kind-makdo-control
```

Berikut adalah hasil yang diharapkan:

```
Kubernetes control plane is running at https://127.0.0.1:60905
CoreDNS is running at https://127.0.0.1:60905/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy

NAME                          STATUS   ROLES           AGE   VERSION
makdo-control-control-plane   Ready    control-plane   1m    v1.33.1
```

Satu node berada dalam status Siap. Klaster sedang berjalan tetapi kosong.

Membuat Konfigurasi Klaster Pekerja

Klaster pekerja adalah tempat beban kerja Anda yang sebenarnya berjalan. MAKDO memantau klaster ini dan memperbaiki masalah ketika muncul. Klaster ini perlu mengekspos server MCP k8s-ai agar MAKDO dapat berkomunikasi dengannya. Buat `worker-klaster.yaml`:

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: makdo-worker
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "cluster-role=worker"
  extraPortMappings:
  # Port for k8s-ai MCP server
  - containerPort: 30100
  hostPort: 8100
  protocol: TCP
  # Port for exposing problematic workloads (for testing)
  - containerPort: 30200
  hostPort: 8200
  protocol: TCP
```

Berikut perbedaan utama dari kluster kontrol:

- `nama`: `makdo-worker` membedakan ini dari kluster kontrol.
- `label node`: `klaster-role=worker` mengidentifikasi ini sebagai kluster terkelola.
- `Port 8100`: Dipetakan ke port kontainer 30100, tempat server A2A k8s-ai akan berjalan. Agen Analyzer dan Fixer MAKDO memanggil k8s-ai melalui port ini menggunakan protokol A2A.
- `Port 8200`: Untuk beban kerja pengujian. Kita akan menyebarkan pod yang rusak di sini yang dapat dideteksi dan diperbaiki oleh MAKDO.

Buat kluster pekerja:

```
kind create cluster --config worker-cluster.yaml
```

Sekarang Anda memiliki dua klaster Kubernetes independen yang berjalan sebagai kontainer Docker:

```
kind get clusters
```

Berikut adalah output yang diharapkan (Anda mungkin juga melihat klaster lain):

```
makdo-control
makdo-worker
```

Periksa kedua konteks klaster:

```
kubectl config get-contexts | grep makdo
```

Berikut hasilnya:

```
          kind-makdo-control   kind-makdo-control   kind-makdo-control
 *        kind-makdo-worker    kind-makdo-worker    kind-makdo-worker
```

Simbol * menunjukkan konteks mana yang saat ini aktif. Beralihlah di antara keduanya dengan perintah berikut:

```
# Use control cluster
kubectl config use-context kind-makdo-control

# Use worker cluster
kubectl config use-context kind-makdo-worker
```

Kedua kluster tersebut terisolasi. Pod di satu kluster tidak dapat langsung menjangkau pod di kluster lain tanpa konfigurasi jaringan eksplisit, yang akan kita siapkan selanjutnya.

Menyiapkan Jaringan Kluster Dan Penerusan Port

Kluster KindD berjalan sebagai kontainer Docker di mesin host Anda. Mereka berada di jaringan Docker yang sama, yang berarti mereka dapat saling menjangkau menggunakan jaringan Docker. Tetapi layanan di dalam kluster memerlukan konfigurasi khusus agar dapat diakses. Pertama, temukan jaringan Docker yang digunakan oleh kedua kluster:

```
docker network ls | grep kind
```

Berikut hasilnya:

```
aadf3e99800d   kind           bridge    local
```

Kedua kontainer klaster terhubung ke jaringan jenis ini:

```
docker network inspect kind | grep -A 3 "makdo"
```

Ini menunjukkan alamat IP dari kedua kontainer kluster. Namun, kita tidak ingin memasukkan IP secara langsung. Sebagai gantinya, kita akan menggunakan pemetaan port yang telah kita konfigurasi sebelumnya.

Untuk komunikasi antar kluster, MAKDO (pada kluster kontrol) perlu mencapai k8s-ai (pada kluster pekerja). Karena kedua kluster berada di host yang sama, MAKDO dapat mencapai `k8s-ai di host.docker.internal:8100`. Nama DNS khusus ini mengarah ke mesin host dari dalam kontainer Docker. Mari kita verifikasi bahwa pemetaan port berfungsi. Periksa apa yang sedang mendengarkan pada port yang dipetakan:

```
# Control cluster ports
lsof -i :8080 2>/dev/null || echo "Port 8080: not yet in use"
lsof -i :9090 2>/dev/null || echo "Port 9090: not yet in use"

# Worker cluster ports
lsof -i :8100 2>/dev/null || echo "Port 8100: not yet in use"
lsof -i :8200 2>/dev/null || echo "Port 8200: not yet in use"
```

Saat ini, port-port ini belum digunakan karena kami belum menerapkan layanan apa pun. Pemetaan port telah dikonfigurasi tetapi tidak aktif sampai layanan terhubung ke port internal (`30080, 30090, 30100,` dan `30200`). Periksa apakah kontainer Docker itu sendiri berjalan dan memiliki pemetaan port:

```
docker ps --format "table {{.Names}}\t{{.Ports}}" | grep makdo
```

Berikut adalah outputnya:

```
makdo-worker-control-plane   127.0.0.1:61462->6443/tcp, 0.0.0.0:8100->30100/tcp,
0.0.0.0:8200->30200/tcp
makdo-control-control-plane  127.0.0.1:60905->6443/tcp, 0.0.0.0:8080->30080/tcp,
0.0.0.0:9090->30090/tcp
```

Anda dapat melihat kedua kontainer dengan pemetaan port-nya. Port 6443 adalah server API Kubernetes. Pemetaan khusus kami (`8080->30080, 8100->30100`, dan lain-lain.) siap digunakan oleh layanan.

Memverifikasi Konektivitas Lintas Klaster

Sebelum menerapkan MAKDO dan k8s-ai, verifikasi bahwa kedua klaster dalam keadaan sehat dan berpotensi dapat berkomunikasi. Periksa kesehatan klaster:

```
kubectl cluster-info --context kind-makdo-control
kubectl get nodes --context kind-makdo-control
```

Berikut hasilnya:

```
Kubernetes control plane is running at https://127.0.0.1:60905
CoreDNS is running at https://127.0.0.1:60905/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy

NAME                          STATUS   ROLES           AGE   VERSION
makdo-control-control-plane   Ready    control-plane   15m   v1.33.1
```

Sekarang, periksa klaster pekerja:

```
kubectl cluster-info --context kind-makdo-worker
kubectl get nodes --context kind-makdo-worker
```

Berikut hasilnya:

```
Kubernetes control plane is running at https://127.0.0.1:61462
CoreDNS is running at https://127.0.0.1:61462/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy

NAME                          STATUS   ROLES           AGE   VERSION
makdo-worker-control-plane    Ready    control-plane   12m   v1.33.1
```

Kedua klaster menunjukkan bahwa control plane mereka berjalan, dan satu node berada dalam status Siap. Uji jaringan dasar di dalam setiap klaster:

```
# Test DNS in control cluster
kubectl run dns-test --image=busybox:latest --restart=Never --rm -i --context
kind-makdo-control --command -- \
    nslookup kubernetes.default.svc.cluster.local

# Test DNS in worker cluster
kubectl run dns-test --image=busybox:latest --restart=Never --rm -i --context
kind-makdo-worker --command -- \
    nslookup kubernetes.default.svc.cluster.local
```

Keduanya harus berhasil menyelesaikan nama DNS layanan Kubernetes. Untuk komunikasi lintas kluster, MAKDO tidak akan langsung menjangkau pod di kluster pekerja. Sebaliknya, MAKDO akan memanggil server A2A k8s-ai melalui port yang diekspos. Kita akan menguji ini setelah k8s-ai di-deploy. Kluster sudah siap untuk langkah selanjutnya.

Menerapkan Beban Kerja Uji Ke Kluster Pekerja

Untuk memverifikasi bahwa MAKDO berfungsi dengan benar, kita membutuhkan beban kerja yang bermasalah agar dapat dideteksi dan diperbaiki. Mari kita terapkan namespace uji dengan tiga pod: satu yang mengalami crash, satu dengan kesalahan penarikan image, dan satu yang sehat. Buat `test-workload.yaml`:

```
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: test-workload
---
# A pod that will crashloop - missing required environment variable
apiVersion: v1
kind: Pod
metadata:
  name: crashloop-pod
  namespace: test-workload
  labels:
    app: crashloop-test
spec:
  containers:
  - name: app
    image: busybox:latest
    command: ["sh", "-c"]
    args:
      - |
        if [ -z "$REQUIRED_VAR" ]; then
          echo "ERROR: REQUIRED_VAR not set"
          exit 1
        fi
        echo "Running successfully"
        sleep 3600
---
# A pod with an image that doesn't exist
apiVersion: v1
kind: Pod
metadata:
  name: imagepull-pod
  namespace: test-workload
  labels:
    app: imagepull-test
spec:
  containers:
  - name: app
    image: nonexistent-registry.example.com/fake-image:v1.0.0
    command: ["sleep", "3600"]
---
# A healthy pod for comparison
apiVersion: v1
kind: Pod
metadata:
  name: healthy-pod
  namespace: test-workload
  labels:
    app: healthy-test
spec:
  containers:
  - name: app
    image: nginx:alpine
    ports:
      - containerPort: 80
```

Lakukan deployment ke klaster worker:

```
kubectl apply -f test-workload.yaml --context kind-makdo-worker
```

Periksa status pod:

```
kubectl get pods -n test-workload --context kind-makdo-worker
```

Berikut adalah outputnya:

```
NAME             READY   STATUS             RESTARTS      AGE
crashloop-pod    0/1     CrashLoopBackOff   7 (27s ago)   11m
healthy-pod      1/1     Running            0             11m
imagepull-pod    0/1     ImagePullBackOff   0             11m
```

Sempurna! Kita memiliki hal-hal berikut:

- `crashloop-pod`: `CrashLoopBackOff` (keluar segera karena variabel lingkungan hilang)
- `healthy-pod`: Running (nginx berhasil dimulai)
- `imagepull-pod`: `ImagePullBackOff` (gambar tidak ada)

Perhatikan crashloop-pod yang berulang kali memulai ulang:

```
kubectl get pods -n test-workload --context kind-makdo-worker --watch
```

Tekan *ctrl* + *c* untuk berhenti menonton. Pod yang gagal ini adalah target yang sempurna untuk MAKDO. Analyzer akan mendeteksinya, mengklasifikasikan jenis kegagalan, dan Fixer dapat mencoba melakukan perbaikan. Periksa log pod untuk melihat alasan kegagalan:

```
kubectl logs crashloop-pod -n test-workload --context kind-makdo-worker
```

Berikut hasilnya:

```
ERROR: REQUIRED_VAR not set
```

Klaster pekerja sekarang memiliki beban kerja yang bermasalah. Selanjutnya, kita akan menerapkan MAKDO ke klaster kontrol dan k8s-ai ke klaster pekerja, lalu mengamati MAKDO secara otomatis mendeteksi dan menangani masalah ini.

11.3 MENERAPKAN MAKDO KE KLASTER KONTROL

Sekarang setelah kita memiliki kedua klaster yang berjalan dan beban kerja pengujian gagal di klaster pekerja, saatnya untuk menerapkan MAKDO. MAKDO akan berjalan sebagai Deployment di klaster kontrol dan memantau klaster pekerja melalui server A2A k8s-ai.

Mempersiapkan Konfigurasi Makdo

MAKDO membutuhkan file konfigurasi untuk mengetahui klaster mana yang akan dipantau, bagaimana cara terhubung ke k8s-ai, dan bagaimana cara berkomunikasi melalui Slack. Kita akan menyesuaikan konfigurasi contoh untuk pengaturan dua klaster kita. Lihat https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/tree/main/ch11/makdo untuk detailnya. Berikut adalah poin-poin konfigurasi utama:

- **Klaster**: Hanya klaster pekerja yang tercantum. MAKDO tidak perlu memantau klaster kontrolnya sendiri. Konteks kind-makdo-worker cocok dengan konteks kubectl yang dibuat oleh KinD.
- `kubeconfig_path`: Diatur ke `/root/.kube/config` karena di sinilah kubeconfig akan dipasang di dalam kontainer MAKDO. Kita akan memasang kubeconfig sebagai volume.
- `k8s_ai.base_url`: Menggunakan host.docker.internal:8100 untuk mencapai k8s-ai pada klaster pekerja. Nama DNS khusus ini mengarah ke mesin host dari dalam kontainer Docker. Port 8100 adalah port yang dipetakan untuk server A2A k8s-ai.
- **Variabel lingkungan**: Kunci API dan token menggunakan sintaks ${VAR}. MAKDO memuat ini dari variabel lingkungan, yang akan kita berikan melalui rahasia Kubernetes.
- `monitoring.check_interval`: Atur ke 120 detik (2 menit). MAKDO melakukan polling klaster pekerja setiap dua menit untuk mendeteksi masalah dengan cepat.

Selanjutnya, kita membutuhkan file konfigurasi agen. MAKDO menggunakan file YAML terpisah untuk setiap prompt dan konfigurasi alat agen. Buat `makdo-config/coordinator.yaml`:

```
name: "MAKDO Coordinator"
description: |

  You are the MAKDO Coordinator, the central orchestrator for multi-cluster
Kubernetes operations.

  Your responsibilities:
  1. Periodically check cluster health across all registered clusters
  2. Coordinate between Analyzer, Fixer, and Slack Bot agents
  3. Escalate critical issues to appropriate agents
  4. Maintain operational awareness of all clusters

When asked to perform health checks:
  1. Use the Analyzer agent to assess cluster health
  2. If issues are found, use the Slack Bot to notify users
  3. For critical issues, use the Fixer agent if remediation is needed
  4. Always report status back to the Slack channel

  Be proactive, clear, and focused on cluster reliability.

model_id: "gpt-4o"

sub_agents:
  - name: "MAKDO_Analyzer"
    config_path: "src/makdo/agents/analyzer.yaml"

  - name: "MAKDO_Fixer"
    config_path: "src/makdo/agents/fixer.yaml"

  - name: "MAKDO_Slack_Bot"
    config_path: "src/makdo/agents/slack.yaml"

history:
```

```
  max_messages: 10
  summarize_after: 5
Create makdo-config/analyzer.yaml:
name: "MAKDO Analyzer"
description: |
  You are the MAKDO Analyzer agent. Your role is to diagnose Kubernetes cluster
health issues.

  Your responsibilities:
  1. Check cluster resource health (pods, deployments, services, nodes)
  2. Identify problems like CrashLoopBackOff, ImagePullBackOff, resource
exhaustion
  3. Analyze logs and events to understand root causes
  4. Provide clear diagnostic reports to the Coordinator

  When analyzing issues:
  1. Always check ALL namespaces, not just default
  2. Look at pod status, events, and logs
  3. Identify patterns across multiple failing pods
  4. Classify issues by severity (critical, warning, info)
  5. Provide actionable recommendations

  Use the k8s diagnostic tools available to you through the A2A protocol.

model_id: "gpt-4o"

tools:
  - name: "kubernetes_resource_health"
    type: "a2a"
    endpoint: "http://host.docker.internal:8100"
    description: "Check health of Kubernetes resources (pods, deployments,
services, nodes)"

  - name: "kubernetes_diagnose_issue"
    type: "a2a"
    endpoint: "http://host.docker.internal:8100"
    description: "Diagnose specific Kubernetes issues by analyzing events, logs,
and resource state"


history:
  max_messages: 8
  summarize_after: 4
Create makdo-config/fixer.yaml:
name: "MAKDO Fixer"
description: |

   You are the MAKDO Fixer agent. Your role is to remediate Kubernetes cluster
issues safely.

Your responsibilities:
  1. Apply fixes for diagnosed issues from the Analyzer
  2. Always prioritize safety and avoid destructive actions
  3. Request approval for critical operations
  4. Verify fixes were successful after applying

Common remediation actions:
  - Restart failing pods
  - Update pod configurations to fix missing env vars
```

```
  - Scale deployments
  - Apply configuration changes

CRITICAL SAFETY RULES:
  1. Never delete resources without explicit approval
  2. Always verify before applying changes
  3. If unsure, ask for human approval
  4. Test fixes on non-production resources first when possible

  Use the k8s remediation tools available through the A2A protocol.


model_id: "gpt-4o"

tools:
  - name: "kubernetes_apply_fix"
    type: "a2a"
    endpoint: "http://host.docker.internal:8100"
    description: "Apply fixes to Kubernetes resources (restart pods, update
configs, scale)"

  - name: "kubernetes_verify"
    type: "a2a"
    endpoint: "http://host.docker.internal:8100"
    description: "Verify that applied fixes resolved the issue"

history:
  max_messages: 6
  summarize_after: 3
Create makdo-config/slack.yaml:
name: "MAKDO Slack Bot"
description: |

  You are the MAKDO Slack Bot agent. Your role is to communicate with users via
Slack.

Your responsibilities:
  1. Post cluster health reports to the #makdo-devops channel
  2. Alert users to critical issues
  3. Provide status updates on ongoing operations
  4. Respond to user questions about cluster health

Message formatting:
  - Use clear, concise language
  - Highlight critical issues with appropriate urgency
  - Include relevant details (pod names, namespaces, error messages)
  - Provide actionable next steps when possible

  Always post to #makdo-devops channel unless told otherwise.

model_id: "gpt-4o"
mcp_servers:
  slack:
    command: "npx"
    args: ["@korotovsky/slack-mcp-server"]
    env:
      SLACK_BOT_TOKEN: "${AI6_BOT_TOKEN}"
      SLACK_TEAM_ID: "${SLACK_TEAM_ID}"
```

```
history:
  max_messages: 5
  summarize_after: 3
```

File konfigurasi ini mendefinisikan perilaku, alat, dan cara komunikasi setiap agen. Agen Analyzer dan Fixer menggunakan alat A2A yang mengarah ke k8s-ai. Agen `Slack_Bot` menggunakan server MCP untuk integrasi Slack.

Membuat Rahasia Kubernetes Untuk Kunci API

MAKDO membutuhkan kunci API untuk OpenAI (untuk menjalankan agen berbasis LLM) dan Slack (untuk mengirim pesan). Kami akan menyimpan ini sebagai rahasia Kubernetes di klaster kontrol. Pertama, buat file .env dengan kunci API Anda:

```
cat > .env <<EOF
OPENAI_API_KEY=sk-...your-key-here...
AI6_BOT_TOKEN=xoxb-...your-slack-bot-token...
AI6_APP_TOKEN=xapp-...your-slack-app-token...
SLACK_TEAM_ID=T...your-team-id...
EOF
```

Ganti placeholder dengan kunci Anda yang sebenarnya. Jika Anda belum memiliki kunci-kunci ini, berikut cara mendapatkannya:

- **Kunci API OpenAI**: Dapatkan dari https://platform.openai.com/api-keys
- **Token Slack**: Buat aplikasi Slack di https://api.slack.com/apps dengan cakupan token bot (chat:write dan channels:read) dan token tingkat aplikasi untuk mode soket
- **ID Tim Slack**: Temukan di pengaturan ruang kerja Slack Anda di bawah ID Ruang Kerja

Buat rahasia Kubernetes di kluster kontrol:

```
# Switch to control cluster
kubectl config use-context kind-makdo-control

# Load environment variables
set -a
source .env
set +a

# Create secret
kubectl create secret generic makdo-secrets \
  --from-literal=openai-api-key="$OPENAI_API_KEY" \
  --from-literal=slack-bot-token="$AI6_BOT_TOKEN" \
  --from-literal=slack-app-token="$AI6_APP_TOKEN" \
  --from-literal=slack-team-id="$SLACK_TEAM_ID" \
  -n default
```

Verifikasi bahwa rahasia tersebut telah dibuat:

```
kubectl get secret makdo-secrets -n default
```

Berikut adalah hasilnya:

```
NAME            TYPE      DATA   AGE
makdo-secrets   Opaque    4      5s
```

Rahasia telah dibuat. Jangan khawatir nilai-nilainya disembunyikan. Itulah tujuan dari rahasia. Anda dapat memverifikasi bahwa kunci tersebut ada:

```
kubectl describe secret makdo-secrets -n default
```

Berikut hasilnya:

```
Name:         makdo-secrets
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
openai-api-key:   51 bytes
slack-app-token:  53 bytes
slack-bot-token:  56 bytes
slack-team-id:    11 bytes
```

Rahasia sudah siap. Kita akan memasangnya sebagai variabel lingkungan di deployment MAKDO.

Membuat Configmap Kubeconfig

MAKDO membutuhkan akses ke file kubeconfig kluster pekerja untuk mengelolanya. Kita akan membuat sumber daya ConfigMap yang berisi file kubeconfig dan memasangnya ke dalam pod MAKDO. Perhatikan bahwa di lingkungan produksi, lebih baik menyimpannya dalam sebuah secret untuk tujuan keamanan. Ekstrak kubeconfig kluster pekerja dan buat ConfigMap:

```
# Create kubeconfig ConfigMap from your local kubeconfig
kubectl create configmap makdo-kubeconfig \
  --from-file=config=$HOME/.kube/config \
  --context kind-makdo-control
```

ConfigMap ini akan dipasang di `/root/.kube` di dalam kontainer MAKDO, sehingga memungkinkan akses ke klaster kontrol dan klaster pekerja. Verifikasi bahwa ConfigMap telah dibuat:

```
kubectl get configmap makdo-kubeconfig --context kind-makdo-control
```

Berikut hasilnya:

```
cd ch11
chmod +x check-prerequisites.sh
./check-prerequisites.sh
```

Kubeconfig kini tersedia untuk digunakan MAKDO saat terhubung ke klaster pekerja.

Menerapkan Makdo Sebagai Deployment Kubernetes

MAKDO akan berjalan sebagai deployment dengan satu replika. Deployment menyediakan restart otomatis jika pod mengalami crash dan memudahkan pembaruan konfigurasi dengan meluncurkan versi baru. Berikut adalah manifest deployment: https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/blob/main/ch11/makdo/deployment.yaml.

Berikut adalah konfigurasi deployment utama:

- `ConfigMap`: Berisi file konfigurasi makdo.yaml. Ini memungkinkan kita untuk memperbarui konfigurasi tanpa membangun ulang image Docker.
- `replika`: 1: Satu instance MAKDO. Beberapa replika akan menyebabkan pemeriksaan kesehatan ganda dan konflik koordinasi. Dalam produksi, disarankan untuk menggunakan pemilihan pemimpin agar instance siaga siap mengambil alih.
- `imagePullPolicy`: Never: Kami akan membangun image MAKDO secara lokal dan memuatnya ke KinD. Jangan pernah menarik dari registry untuk mendapatkan kode terbaru.
- **Bagian** env: Variabel lingkungan dimuat dari secret makdo-secrets. Ini menimpa placeholder ${VAR} di config.
- `volumeMounts`: Dua volume dipasang:
  - `/app/config`: Konfigurasi MAKDO dari ConfigMap
  - `/root/.kube`: File kubeconfig dari ConfigMap sehingga MAKDO dapat mengakses kedua klaster
- `resources`: Batasan memori dan CPU mencegah MAKDO mengonsumsi terlalu banyak. 512 Mi hingga 1 Gi wajar untuk proses Python dan panggilan LLM.
- `volume`: Dua sumber daya ConfigMap dipasang:
  - makdo-config: Berisi file konfigurasi MAKDO
  - makdo-kubeconfig: Berisi file kubeconfig untuk mengakses kedua klaster

Sebelum melakukan deployment, kita perlu membuat image Docker MAKDO dan memuatnya ke dalam klaster kontrol.

Membuat Dan Memuat Image Makdo

MAKDO perlu dikemas sebagai image Docker. Kita akan membuat Dockerfile yang menginstal semua dependensi dan menjalankan proses utama MAKDO. Buat `makdo/Dockerfile`:

```
FROM python:3.13-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
  curl \
  git \
```

```
  kubectl \
  && rm -rf /var/lib/apt/lists/*

# Install Node.js for Slack MCP server
RUN curl -sSL https://deb.nodesource.com/setup_20.x | bash - \
  && apt-get install -y nodejs \
  && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Copy MAKDO source from the book repository
# Adjust path based on where you cloned the book repo
COPY ../../packt/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/ch09/makdo/src ./
src
COPY ../../packt/Design-Multi-Agent-AI-Systems-Using-MCP-and-
A2A/ch09/makdo/pyproject.toml .

# Install MAKDO dependencies
RUN pip install --no-cache-dir ai-six==0.14.4 pyyaml python-dotenv requests

# Create required directories
RUN mkdir -p /app/config /app/logs

# Set Python to run unbuffered so logs appear immediately
ENV PYTHONUNBUFFERED=1

# Run MAKDO main entry point
CMD ["python", "-m", "src.makdo.main"]
```

Dockerfile menggunakan Python 3.13. Versi ini memiliki binary wheel yang sudah dibuat sebelumnya untuk semua dependensi, sehingga tidak perlu kompilasi. Kami menginstal kubectl untuk akses kluster Kubernetes dan Node.js untuk server Slack MCP. Dependensi Python (`ai-six`, `pyyaml`, `python-dotenv`, dan `requests`) diinstal langsung dari wheel tanpa memerlukan build tools. Buat image-nya:

```
cd ch11/makdo
docker build -t makdo:latest .
```

Proses ini membutuhkan beberapa menit untuk mengunduh image dasar dan menginstal dependensi. Perhatikan kesalahan yang mungkin terjadi. Jika proses build gagal karena sumber MAKDO hilang, verifikasi bahwa direktori src ada. Setelah selesai dibangun, muat image ke dalam klaster kontrol:

```
kind load docker-image makdo:latest --name makdo-control
```

Perintah ini mentransfer gambar dari Docker lokal Anda ke cache gambar internal klaster KinD. Berikut hasilnya:

```
Image: "makdo:latest" with ID "sha256:..." not yet present on node "makdo-control-
control-plane", loading...
```

Pastikan gambar tersebut tersedia:

```
docker exec -it makdo-control-control-plane crictl images | grep makdo
```

Anda seharusnya melihat citra makdo terdaftar. Sekarang kita dapat melakukan deployment. Terapkan deployment:

```
kubectl apply -f deployment.yaml --context kind-makdo-control
```

Berikut hasilnya:

```
configmap/makdo-config created
deployment.apps/makdo created
```

Periksa status penyebaran:

```
kubectl get deployment makdo --context kind-makdo-control
```

Berikut hasilnya:

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
makdo   1/1     1            1           15s
```

MAKDO sedang berjalan. Satu replika sudah siap.

Mengonfigurasi MAKDO Untuk Terhubung Ke Klaster Pekerja

MAKDO membutuhkan file kubeconfig untuk mengakses klaster pekerja dan token sesi k8s-ai untuk menggunakan alat A2A. File kubeconfig sudah terpasang dari host. Sekarang, kita perlu memastikan bahwa MAKDO dapat membuat sesi dengan k8s-ai.

Server k8s-ai belum di-deploy. Kita akan mendeploynya di bagian selanjutnya. Untuk saat ini, verifikasi bahwa MAKDO setidaknya dapat memulai dan mengakses klaster pekerja melalui kubectl. Periksa log pod MAKDO:

```
kubectl logs -l app=makdo --context kind-makdo-control --tail=50
```

Anda mungkin akan melihat kesalahan tentang k8s-ai yang tidak dapat dijangkau. Itu wajar. Cari log yang menunjukkan bahwa MAKDO berhasil dimulai:

```
2025-11-02 10:15:32,123 - makdo - INFO - Starting MAKDO - Multi-Agent Kubernetes
DevOps System
2025-11-02 10:15:32,456 - makdo - INFO - Coordinator and sub-agents created
successfully
2025-11-02 10:15:32,500 - makdo - INFO - Setting up tool call monitoring...
2025-11-02 10:15:32,550 - makdo - INFO - ☑ Tool call monitoring enabled for all
agents
```

```
2025-11-02 10:15:32,600 - makdo - INFO - Getting kubeconfig for context: kind-
makdo-worker
```

Jika Anda melihat kesalahan tentang kubectl atau kubeconfig, ConfigMap kubeconfig mungkin hilang atau salah. Verifikasi bahwa Anda telah membuat ConfigMap makdo-kubeconfig dengan file kubeconfig Anda. Untuk menguji akses kubectl dari dalam pod MAKDO, gunakan yang berikut:

```
kubectl exec -it deployment/makdo --context kind-makdo-control -- kubectl get
nodes --context kind-makdo-worker
```

Perintah ini menjalankan kubectl di dalam kontainer MAKDO untuk melakukan query ke klaster worker. Berikut adalah outputnya:

```
NAME                          STATUS   ROLES           AGE   VERSION
makdo-worker-control-plane    Ready    control-plane   45m   v1.33.1
```

Berhasil! MAKDO dapat mencapai klaster pekerja. Koneksi k8s-ai akan berfungsi setelah kita menerapkan k8s-ai.

Memverifikasi Penerapan Dan Log MAKDO

Mari kita verifikasi bahwa MAKDO sehat dan siap untuk memantau klaster pekerja. Periksa status pod:

```
kubectl get pods -l app=makdo --context kind-makdo-control
```

Berikut hasilnya:

```
NAME                     READY   STATUS    RESTARTS   AGE
makdo-7c9f8b6d5d-x2k4j   1/1     Running   0          2m
```

Pod sedang berjalan. Periksa penggunaan sumber daya:

```
kubectl top pod -l app=makdo --context kind-makdo-control
```

Berikut adalah outputnya (membutuhkan `metrics-server`, dan mungkin tidak berfungsi di KinD):

```
Error from server (ServiceUnavailable): the server is currently unable to handle
the request (get pods.metrics.k8s.io)
```

Tidak masalah. KinD tidak menyertakan metrics-server secara default. Kita dapat memeriksa batasan sumber daya dari spesifikasi pod:

```
kubectl describe pod -l app=makdo --context kind-makdo-control | grep -A 5
"Limits"
```

Berikut hasilnya:

```
Limits:
  cpu:     500m
  memory:  1Gi
Requests:
  cpu:     250m
  memory:  512Mi
```

Sumber daya telah dikonfigurasi dengan benar. Sekarang, periksa log MAKDO untuk mencari kesalahan:

```
kubectl logs -l app=makdo --context kind-makdo-control --tail=100
```

Perhatikan indikator-indikator kunci berikut:

1. **Koordinator dimulai**: Memulai Sistem DevOps Kubernetes Multi-Agen MAKDO
2. **Sub-agen dibuat**: Koordinator dan sub-agen berhasil dibuat
3. **Mencoba sesi k8s-ai**: Membuat sesi k8s-ai untuk kind-makdo-worker…
4. **Loop pemeriksaan kesehatan dimulai**: Memulai loop pemeriksaan kesehatan

Anda akan melihat kesalahan tentang kegagalan koneksi k8s-ai. Itu wajar sampai kita menerapkan k8s-ai. Yang penting adalah MAKDO dimulai tanpa mengalami crash. Jika MAKDO mengalami crash-loop, periksa masalah umum berikut:

- **Rahasia hilang**: Verifikasi bahwa makdo-secrets ada dengan semua kunci:

```
kubectl get secret makdo-secrets --context kind-makdo-control
```

- **ConfigMap kubeconfig hilang**: Verifikasi bahwa ConfigMap makdo-kubeconfig telah dibuat dengan file kubeconfig Anda sebelum menerapkan MAKDO.
- **Kesalahan Python**: Periksa log untuk kesalahan impor atau dependensi yang hilang. Anda mungkin perlu membangun ulang image.

Untuk mengikuti log secara real-time hanya untuk melihat apa yang dilakukan MAKDO, gunakan yang berikut:

```
kubectl logs -l app=makdo --context kind-makdo-control --follow
```

Tekan tombol *Ctrl* + *C* untuk berhenti. MAKDO telah di-deploy dan menunggu k8s-ai. Selanjutnya, kita akan melakukan deployment k8s-ai ke klaster worker agar MAKDO dapat mulai memantau.

11.4 MENERAPKAN K8S-AI KE KLASTER PEKERJA

Sekarang MAKDO berjalan di klaster kontrol, kita perlu melakukan deployment k8s-ai ke klaster worker. K8s-ai mengekspos kemampuan diagnostik Kubernetes melalui protokol A2A, memungkinkan agen Analyzer dan Fixer MAKDO untuk memeriksa kesehatan klaster dan menjalankan tindakan remediasi. Kita akan mengontainerisasi k8s-ai, melakukan deployment sebagai layanan Kubernetes, dan mengeksposnya agar MAKDO dapat berkomunikasi dengannya di seluruh klaster.

Membangun Citra Docker k8s-ai

K8s-ai perlu dijalankan sebagai kontainer di klaster pekerja. Kita akan membuat Dockerfile yang mengemas k8s-ai dengan semua dependensinya, menginstal kubectl untuk akses klaster, dan mengkonfigurasinya untuk berjalan sebagai server A2A. Buat `k8s-ai/Dockerfile`:

```
FROM python:3.13-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
  curl \
  git \
  && rm -rf /var/lib/apt/lists/*

# Install kubectl
RUN curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/
stable.txt)/bin/linux/arm64/kubectl" && \
  install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl && \
  rm kubectl

WORKDIR /app

# Copy k8s-ai source
COPY k8s_ai ./k8s_ai
COPY pyproject.toml .

# Install k8s-ai and dependencies
RUN pip install --no-cache-dir -e .

# Create directory for API keys
RUN mkdir -p /app/data

# Expose A2A server port and Admin API port
EXPOSE 9999 9998

# Run k8s-ai server in A2A mode
CMD ["k8s-ai-server", "--host", "0.0.0.0", "--port", "9999", "--admin-port",
"9998"]
```

Berikut adalah poin-poin penting dari cuplikan kode sebelumnya:

- **Citra dasar**: Python 3.12-slim menyediakan lingkungan Python minimal.
- **Instalasi kubectl**: K8s-ai menjalankan perintah kubectl untuk berinteraksi dengan klaster. Kita mengunduh versi stabil terbaru dan menginstalnya.

- **Kode sumber**: Salin paket `k8s_ai` dan file `pyproject.toml`. Kita menginstal dengan pip install –e . untuk menginstal k8s-ai sebagai paket yang dapat diedit.
- Direktori data: `/app/data` akan menyimpan kunci API (keys.json) untuk otentikasi.
- Port: Port 9999 untuk server protokol A2A, dan 9998 untuk API Admin (digunakan untuk membuat sesi).
- CMD: Memulai `k8s-ai-server` dengan flag untuk mengikat ke semua antarmuka (`0.0.0.0`) sehingga dapat diakses dari luar kontainer.

Kode sumber k8s-ai berada di repositori utama k8s-ai. Salin kode tersebut ke direktori `ch11/k8s-ai`:

```
# From the k8s-ai repository root
cp -r k8s_ai /path/to/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/ch11/k8s-ai/
cp pyproject.toml /path/to/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/ch11/
k8s-ai/
```

Buat gambarnya:

```
cd ch11/k8s-ai
docker build -t k8s-ai:latest .
```

Proses ini membutuhkan beberapa menit untuk mengunduh image dasar dan menginstal dependensi. Setelah selesai, verifikasi image tersebut:

```
docker images | grep k8s-ai
```

Berikut hasilnya:

```
k8s-ai   latest   abc123def456   2 minutes ago   450MB
```

Citra telah dibuat dan siap untuk diimplementasikan ke klaster pekerja.

Membuat Manifest Penerapan k8s-ai

K8s-ai membutuhkan beberapa sumber daya Kubernetes untuk berjalan: akun Layanan dengan izin untuk mengakses kluster, Deployment untuk menjalankan kontainer, dan Layanan untuk mengeksposnya. Kita akan menggunakan **kontrol akses berbasis peran** (RBAC) untuk memberikan akses kluster hanya baca kepada k8s-ai. Berikut adalah manifest deployment k8s-ai: https://github.com/PacktPublishing/Design-Multi-Agent-AI-Systems-Using-MCP-and-A2A/blob/main/ch11/k8s-ai/deployment.yaml. Poin konfigurasi utama meliputi hal-hal berikut:

- **akun layanan dan RBAC**: Akun layanan k8s-ai mendapatkan izin hanya baca melalui KlasterRole k8s-ai-reader. Ini memungkinkan k8s-ai untuk menjalankan perintah kubectl get/list/watch tetapi mencegah modifikasi apa pun. Sumber daya KlasterRoleBinding menghubungkan ServiceAccount ke KlasterRole.

- **Rahasia kunci API**: Berisi keys.json dengan kunci API dari file .env Anda. Agen Analyzer dan Fixer MAKDO akan menggunakan kunci ini untuk otentikasi. Kunci disimpan sebagai placeholder $ {K8S_AI_API_KEY} dan akan diganti dari lingkungan Anda saat Anda menerapkan manifes.
- **Penyebaran**: Menjalankan satu replika k8s-ai dengan `imagePullPolicy`: Never (KinD menggunakan gambar lokal). Kontainer mendapatkan kunci API OpenAI dari sumber daya Secret. Batasan sumber daya mencegah k8s-ai mengonsumsi memori atau CPU yang berlebihan.
- **Pemeriksaan kesehatan**: Pemeriksaan liveness memeriksa apakah k8s-ai responsif dengan mengambil kartu agen di `/.well-known/agent.json`. Pemeriksaan readiness memastikan bahwa pod siap menerima lalu lintas sebelum menandainya sebagai tersedia.
- **service**: Mengekspos k8s-ai dengan tipe NodePort. Port `30100` dipetakan ke server A2A (9999), dan port `30099` dipetakan ke Admin API (9998). Ingat dari `worker-klaster.yaml` bahwa port `30100` dipetakan ke port host `8100`, sehingga k8s-ai dapat diakses di http://host.docker.internal:8100 dari klaster kontrol.

Mari kita lihat cara menerapkan image k8s-ai yang baru saja kita buat.

Menerapkan k8s-ai KE Klaster Pekerja

Dengan image yang telah dibuat dan manifest yang telah dibuat, kita siap untuk menerapkan k8s-ai ke klaster worker. Pertama, muat image ke dalam klaster worker, buat secret yang diperlukan, lalu terapkan deployment. Muat image k8s-ai ke dalam klaster worker:

```
kind load docker-image k8s-ai:latest --name makdo-worker
```

Berikut hasilnya:

```
Image: "k8s-ai:latest" with ID "sha256:1fe7b6c86eb2..." not yet present on node
"makdo-worker-control-plane", loading...
```

Langkah ini mentransfer image dari Docker lokal Anda ke registry internal klaster pekerja. Tanpa langkah ini, pod akan gagal dengan kesalahan ImagePullBackOff karena klaster KinD tidak memiliki akses ke registry eksternal secara default. Buat secret kunci API OpenAI. K8s-ai membutuhkan ini untuk memanggil LLM untuk pemrosesan bahasa alami:

```
kubectl create secret generic k8s-ai-secrets \
  --from-literal=openai-api-key="${OPENAI_API_KEY}" \
  --context kind-makdo-worker
```

Verifikasi bahwa rahasia tersebut telah dibuat:

```
kubectl get secret k8s-ai-secrets --context kind-makdo-worker
```

Berikut hasilnya:

```
NAME              TYPE      DATA   AGE
k8s-ai-secrets    Opaque    1      5s
```

Sebelum melakukan deployment, pastikan Anda memiliki file .env di direktori ch11 dengan kredensial yang diperlukan. Salin file contoh berikut:

```
cd ch11
cp .env.example .env
```

- Edit file `.env` dan atur kredensial Anda.
- Atur `K8S_AI_API_KEY` ke test-key untuk pengembangan.
- Untuk produksi, buat kunci dengan menjalankan `k8s-ai-server --generate-key` dan tetapkan ke `K8S_AI_API_KEY`.
- Atur `OPENAI_API_KEY` ke kunci API OpenAI Anda.
- Kredensial Slack hanya diperlukan jika menggunakan bot Slack.
- File `.env ada di .gitignore`, jadi rahasia Anda tidak akan di-commit.

Sekarang, terapkan manifest deployment, gunakan envsubst untuk mengganti variabel lingkungan:

```
cd k8s-ai
export $(grep -v '^#' ../.env | xargs)
envsubst < deployment.yaml | kubectl apply -f - --context kind-makdo-worker
```

Berikut hasilnya:

```
serviceaccount/k8s-ai created
klasterrole.rbac.authorization.k8s.io/k8s-ai-reader created
klasterrolebinding.rbac.authorization.k8s.io/k8s-ai-reader-binding created
secret/k8s-ai-api-keys created
deployment.apps/k8s-ai created
service/k8s-ai created
```

Enam sumber daya dibuat: ServiceAccount, KlasterRole, KlasterRoleBinding, Secret (untuk kunci API), Deployment, dan Service. Periksa status pod:

```
kubectl get pods -l app=k8s-ai --context kind-makdo-worker
```

Berikut hasilnya:

```
NAME                      READY   STATUS    RESTARTS   AGE
k8s-ai-7b9c8d6f5d-x2k4j   1/1     Running   0          45s
```

Pod sedang berjalan. Jika Anda melihat `ImagePullBackOff`, verifikasi bahwa image telah dimuat dengan kind load. Jika Anda melihat `CrashLoopBackOff`, periksa log untuk kesalahan (biasanya kunci API OpenAI hilang). Periksa layanan:

```
kubectl get svc k8s-ai --context kind-makdo-worker
```

Berikut hasilnya:

```
NAME      TYPE       KLASTER-IP      EXTERNAL-IP   PORT(S)                         AGE
k8s-ai    NodePort   10.96.145.123   <none>        9999:30100/TCP,9998:30099/TCP   1m
```

Layanan ini mengekspos dua port node, yaitu `30100` untuk server A2A dan `30099` untuk API Admin. Karena kami telah mengkonfigurasi klaster pekerja dengan extraPortMappings di file `worker-klaster.yaml`, port-port ini dapat diakses dari host di `localhost`:`8100` dan localhost:`8099`.

Mengekspos Layanan k8s-ai Untuk Akses Lintas Klaster

K8s-ai sekarang berjalan di klaster pekerja, tetapi MAKDO di klaster kontrol perlu menjangkaunya. Dalam pengaturan KinD kami, komunikasi lintas klaster terjadi melalui jaringan host Docker. Port node klaster pekerja `30100` dipetakan ke port host `8100` melalui bidang extraPortMappings yang telah kami konfigurasi sebelumnya. Dari perspektif klaster kontrol, klaster pekerja dapat dijangkau di `host.docker.internal`:`8100`.

Nama DNS khusus ini mengarah ke alamat IP host Docker, memungkinkan kontainer di satu klaster KinD untuk menjangkau layanan di klaster lain. Mari kita verifikasi apakah jaringan berfungsi. Pertama, periksa apakah layanan k8s-ai merespons endpoint kartu agen:

```
curl http://localhost:8100/.well-known/agent.json
```

Berikut adalah hasilnya (diformat agar mudah dibaca):

```
{
  "name": "k8s-ai Diagnostic Agent",
  "description": "Kubernetes AI diagnostic agent with read-only cluster analysis
with session-based cluster management",
  "url": "http://0.0.0.0:9999/",
  "version": "2.0.0",
  "capabilities": {
    "streaming": false
  },
  "skills": [
    {
      "id": "kubernetes_diagnostics",
      "name": "Kubernetes Diagnostics",
      "description": "Perform read-only Kubernetes cluster diagnostics,
troubleshooting, and health analysis with detailed insights",
```

```
      "tags": ["kubernetes", "diagnostics", "troubleshooting", "monitoring",
"health"]
    }
  ]
}
```

Kartu agen mengkonfirmasi bahwa k8s-ai merespons. Sekarang, uji otentikasi dengan memanggil Admin API untuk membuat sesi. Admin API membutuhkan kunci API dari file .env Anda:

```
curl -X POST http://localhost:8099/sessions \
  -H "Authorization: Bearer test-key" \
  -H "Content-Type: application/json" \
  -d '{
    "cluster_name": "makdo-worker",
    "ttl_hours": 24
  }'
```

Ini akan gagal dengan kesalahan autentikasi jika Anda belum mengkonfigurasi properti ServiceAccount. Itu wajar karena kita hanya memverifikasi apakah endpoint dapat dijangkau. Kita akan menguji pembuatan sesi lengkap di subbagian berikutnya.

Poin penting di sini adalah bahwa agen MAKDO akan menggunakan http://host.docker.internal:8100 untuk mencapai server A2A k8s-ai. Ini sudah dikonfigurasi dalam file coordinator.yaml MAKDO:

```
a2a_servers:
  - name: "kind-makdo-worker"
    url: "http://host.docker.internal:8100"
    timeout: 30.0
    api_key: "test-key"
```

Jaringan sudah terpasang. MAKDO sekarang dapat berkomunikasi dengan k8s-ai di seluruh klaster.

Memverifikasi Penyebaran k8s-ai

Setelah menyebarkan k8s-ai, verifikasi bahwa semua komponen berjalan dengan benar dan layanan dapat diakses.

Memeriksa Status Pod

Pertama, verifikasi bahwa pod k8s-ai berjalan di klaster pekerja:

```
kubectl get pods -l app=k8s-ai --context kind-makdo-worker -o wide
```

Berikut adalah hasil yang diharapkan:

```
NAME                          READY   STATUS    RESTARTS   AGE   IP           NODE
k8s-ai-75558cb559-n8jl9       1/1     Running   0          20m   10.244.0.5   makdo-
worker-control-plane
```

Pod harus menunjukkan status Running 1/1 tanpa restart.

Verifikasi Kesehatan Pod

Periksa status detail pod dan probe kesehatannya:

```
kubectl describe pod -l app=k8s-ai --context kind-makdo-worker | grep -A 5
"Conditions:"
```

Berikut adalah hasil yang diharapkan:

```
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
```

Semua kondisi harus bernilai True, yang menunjukkan bahwa pod telah melewati pemeriksaan liveness dan readiness.

Periksa Log Server

Lihat log k8s-ai untuk memastikan bahwa kedua server telah berhasil dimulai:

```
kubectl logs -l app=k8s-ai --context kind-makdo-worker --tail=50
```

Cari aktivitas pemeriksaan kesehatan yang menunjukkan bahwa titik akhir A2A merespons:

```
INFO:     10.244.0.1:41262 - "GET /.well-known/agent.json HTTP/1.1" 200 OK
```

Permintaan pemeriksaan kesehatan rutin dari Kubernetes mengkonfirmasi bahwa layanan tersebut sehat.

Uji Titik Akhir A2A

Verifikasi bahwa kartu agen k8s-ai dapat diakses dari luar klaster:

```
curl -s http://localhost:8100/.well-known/agent.json | python3 -m json.tool | head
-30
```

Berikut adalah respons yang diharapkan:

```
{
    "capabilities": {
      "streaming": false
    },
    "defaultInputModes": [
      "text/plain"
    ],
    "defaultOutputModes": [
      "text/plain"
```

```
    ],
    "description": "Kubernetes AI diagnostic agent with read-only klaster analysis
with session-based klaster management",
    "name": "k8s-ai Diagnostic Agent",
    "preferredTransport": "JSONRPC",
    "protocolVersion": "0.3.0",
    "security": [
        {
            "BearerAuth": []
        }
  ],
  "securitySchemes": {
      "BearerAuth": {
          "scheme": "bearer",
          "type": "http"
      }
  },
  "skills": [
      {
        "description": "Perform read-only Kubernetes klaster diagnostics..."
```

Ini mengkonfirmasi bahwa server A2A berjalan di port 9999 di dalam pod, layanan NodePort memetakan port 30100 ke port kontainer, pemetaan port host (`8100 → 30100`) memungkinkan akses eksternal, kartu agen mengikuti spesifikasi protokol A2A, dan otentikasi token bearer diaktifkan.

Verifikasi Konfigurasi Layanan

Periksa konfigurasi layanan k8s-ai:

```
kubectl get svc k8s-ai --context kind-makdo-worker -o yaml | grep -A 10 "spec:"
```

Berikut adalah hasil yang diharapkan:

```
spec:
  ports:
  - name: a2a
    nodePort: 30100
    port: 9999
    protocol: TCP
    targetPort: 9999
  - name: admin
    nodePort: 30099
    port: 9998
    protocol: TCP
    targetPort: 9998
  selector:
    app: k8s-ai
  type: NodePort
```

Ini menunjukkan bahwa port A2A (9999) dan port Admin API (9998) sama-sama terekspos melalui NodePort.

Dengan k8s-ai yang telah diverifikasi dan berjalan, klaster pekerja kini memiliki agen diagnostik yang siap menerima permintaan dari MAKDO di klaster kontrol.

Menguji Admin API k8s-ai Dan Manajemen Sesi

Admin API k8s-ai menyediakan kemampuan manajemen sesi, memungkinkan MAKDO untuk membuat sesi terisolasi untuk berinteraksi dengan klaster pekerja. Setiap sesi memiliki kubeconfig dan waktu kedaluwarsa sendiri.

Mengekspos Port API Admin

Secara default, konfigurasi klaster pekerja mengekspos port A2A (8100) tetapi tidak port API Admin. Perbarui konfigurasi klaster pekerja untuk menambahkan port 8099:

```
# ch11/worker-cluster.yaml
extraPortMappings:
  # Port for k8s-ai A2A server
  - containerPort: 30100
    hostPort: 8100
    protocol: TCP

  # Port for k8s-ai Admin API
  - containerPort: 30099
    hostPort: 8099
    protocol: TCP
```

Jika Anda sudah membuat klaster pekerja, Anda perlu membuatnya ulang dengan konfigurasi yang diperbarui:

```
kind delete cluster --name makdo-worker
kind create cluster --config worker-cluster.yaml
```

Kemudian, sebarkan ulang k8s-ai mengikuti langkah-langkah penyebaran sebelumnya.

Membuat Sesi

Untuk membuat sesi, Anda perlu menyediakan kubeconfig klaster. Pertama, ekstrak kubeconfig klaster pekerja:

```
kubectl config view --context kind-makdo-worker --minify --flatten > /tmp/makdo-
worker-kubeconfig.yaml
```

Buat file JSON permintaan sesi:

```
import json

# Read kubeconfig
with open('/tmp/makdo-worker-kubeconfig.yaml', 'r') as f:
    kubeconfig = f.read()

# Create session request
request = {
```

```
    "cluster_name": "makdo-worker",
    "ttl_hours": 24,
    "kubeconfig": kubeconfig
}

# Write to file
with open('/tmp/session-request.json', 'w') as f:
    json.dump(request, f)

print("Session request created")
```

Sekarang, buat sesi menggunakan Admin API:

```
Error from server (Forbidden): pods "k8s-ai-f77d65f7d-p6lkk" is forbidden:
User "system:serviceaccount:default:k8s-ai" cannot delete resource "pods"
in API group "" in the namespace "default"
```

Berikut adalah respons yang diharapkan:

```
{
  "success": true,
  "session_token": "k8s-ai-session--qSTOZ0ggxnL-w5mMMNboUBz0GVIGXvxkxfjYY5ND0",
  "klaster_name": "makdo-worker",
  "api_server": "https://127.0.0.1:65089",
  "namespace": "default",
  "connectivity_status": "connected",
  "expires_at": "2025-11-10T04:37:27.297559Z",
  "error": null
}
```

Token sesi (`k8s-ai-session--...`) akan digunakan untuk mengautentikasi permintaan ke endpoint A2A.

Daftar Sesi Aktif

Lihat semua sesi aktif:

```
curl -s 'http://localhost:8099/sessions' \
  -H 'Authorization: Bearer test-key' | python3 -m json.tool
```

Berikut adalah respons yang diharapkan:

```
{
  "total_sessions": 1,
  "sessions": [
     {
         "session_token":"k8s-ai-session--qSTOZ0ggxnL-
w5mMmMNboUBz0GVIGXvxkxfjYY5ND0",
            "klaster_name": "makdo-worker",
            "api_server": "https://127.0.0.1:65089",
            "namespace": "default",
```

```
            "created_at": "2025-11-09T04:37:27.297568Z",
            "expires_at": "2025-11-10T04:37:27.297559Z",
            "is_expired": false
        }
    ]
}
```

Mari kita lihat cara menghapus sesi yang sudah ada.

Menghapus Sesi

Untuk menghapus sesi secara manual sebelum kedaluwarsa, gunakan cara berikut:

```
SESSION_TOKEN="k8s-ai-session--qSTOZ0ggxnL-w5mMmWNb0UBz0GVIGXvxxxfjYYSND0"

curl -s -X DELETE "http://localhost:8099/sessions/${SESSION_TOKEN}" \
  -H 'Authorization: Bearer test-key' | python3 -m json.tool
```

Berikut adalah respons yang diharapkan:

```
{
  "success": true,
  "message": "Session deleted successfully"
}
```

Mari kita lihat cara menjalankan sesi A2A terhadap server k8s-ai.

Menggunakan Sesi Dengan Protokol A2A

MAKDO akan menggunakan token sesi saat melakukan permintaan A2A ke k8s-ai. Token tersebut diteruskan dalam header Otorisasi bersama dengan parameter khusus metode:

```
curl -s -X POST 'http://localhost:8100/' \
  -H 'Authorization: Bearer test-key' \
  -H 'Content-Type: application/json' \
  -d '{
    "jsonrpc": "2.0",
    "method": "kubernetes_diagnose_issue",
    "params": {
      "session_token": "'"${SESSION_TOKEN}"'",
      "issue_description": "Check pod health in default namespace"
    },
    "id": 1
  }'
```

Pendekatan berbasis sesi memberikan beberapa manfaat:

- **Isolasi**: Setiap instance MAKDO dapat memiliki sesinya sendiri tanpa mengganggu yang lain
- **Keamanan**: Sesi berakhir secara otomatis, membatasi jendela akses
- **Fleksibilitas**: Sesi yang berbeda dapat menargetkan namespace atau klaster yang berbeda
- **Auditabilitas**: Pembuatan dan penggunaan sesi dicatat untuk audit keamanan

Dengan API Admin yang telah diuji dan berfungsi, k8s-ai kini siap melayani permintaan diagnostik dari MAKDO di seluruh klaster.

11.5 MENGAKTIFKAN KOMUNIKASI YANG AMAN ANTAR KLASTER

Dengan MAKDO yang diimplementasikan di klaster kontrol dan k8s-ai yang diimplementasikan di klaster pekerja, kita perlu memastikan komunikasi yang aman di antara keduanya. Bagian ini mencakup penemuan layanan, jaringan, kontrol akses, dan mekanisme keamanan yang sudah ada, yang menunjukkan cara kerjanya di lingkungan KinD.

Penemuan Layanan Dan Jaringan

Dalam penerapan multi-klaster, agen perlu menemukan dan berkomunikasi satu sama lain di seluruh batas klaster. Penerapan kami menggunakan konfigurasi eksplisit dan jaringan host untuk memungkinkan komunikasi lintas klaster.

Bagaimana Makdo Menemukan k8s-ai

MAKDO menggunakan konfigurasi statis untuk menemukan layanan k8s-ai. Lihat konfigurasinya:

```
kubectl get configmap makdo-config --context kind-makdo-control \
  -o jsonpath='{.data.makdo\.yaml}' | grep -A 5 "k8s_ai:"
```

Berikut adalah hasil yang diharapkan:

```
k8s_ai:
  base_url: "http://host.docker.internal:8100"
  admin_api_url: "http://host.docker.internal:8100/admin"
```

Komponen jaringan utama meliputi hal-hal berikut:

- `host.docker.internal`: Nama DNS khusus yang mengarah ke host Docker dari dalam kontainer
- **Port** `8100`: Dipetakan dari NodePort `30100` di klaster pekerja ke port host 8100
- **Port** `8099`: Dipetakan dari NodePort `30099` untuk API Admin

Melacak Jalur Permintaan

Mari kita lacak bagaimana permintaan diagnostik mengalir dari MAKDO ke k8s-ai:

```
MAKDO Pod (control klaster)
 ↓ http://host.docker.internal:8100
Docker Host Network
 ↓ localhost:8100
Worker Klaster Node (Docker container)
 ↓ NodePort 30100
k8s-ai Service (worker klaster)
 ↓ KlasterIP:9999
k8s-ai Pod (worker klaster)
```

Verifikasi setiap langkah:

```
# 1. Check MAKDO can resolve host.docker.internal
kubectl exec deployment/makdo --context kind-makdo-control -- \


 python3 -c "import socket; print(f'host.docker.internal resolves to:
{socket.gethostbyname(\"host.docker.internal\")})')"

# 2. Check host can access port 8100
curl -s http://localhost:8100/.well-known/agent.json | python3 -m
json.tool | head -10

# 3. Check NodePort service in worker cluster
kubectl get svc k8s-ai --context kind-makdo-worker -o yaml | grep -A 5 "nodePort:
30100"

# 4. Check k8s-ai pod is receiving traffic
kubectl logs deployment/k8s-ai --context kind-makdo-worker --tail=5
```

Dalam sistem produksi, rangkaian panggilan jaringan ini dapat menambah penundaan yang signifikan pada operasi yang sensitif terhadap latensi. Mungkin berguna untuk menambahkan anggaran latensi dan secara eksplisit memantau serta mengelolanya.

Untuk mengakses server k8s-ai, kita perlu menemukannya terlebih dahulu. Ada beberapa pola umum yang berguna untuk berbagai kasus penggunaan. Mari kita jelajahi pola-pola tersebut.

Pola Penemuan Layanan

Penyebaran kami menunjukkan penemuan layanan konfigurasi statis. Pola lain untuk produksi meliputi:

- **Penemuan berbasis DNS**: Gunakan Kubernetes DNS untuk layanan dalam klaster:

```
# For services in the same cluster
k8s_ai:
  base_url: "http://k8s-ai.default.svc.cluster.local:9999"
```

- **Penemuan service mesh**: Gunakan service mesh seperti Istio atau Linkerd:

```
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: k8s-ai-external
spec:
  hosts:
  - k8s-ai.worker-cluster.global
  location: MESH_EXTERNAL
  ports:
  - number: 9999
  name: a2a
  protocol: HTTP
  resolution: DNS

endpoints:
  - address: k8s-ai.worker-cluster.svc.cluster.local
```

- **Registri layanan dinamis**: Gunakan registri layanan seperti Consul:

```
import consul

# Register k8s-ai service
client = consul.Consul()
client.agent.service.register(
    name='k8s-ai',
    service_id='k8s-ai-worker-1',
    address='k8s-ai.worker-cluster',
    port=9999,
    tags=['a2a', 'diagnostics']
)

# Discover services
services = client.health.service('k8s-ai', passing=True)
for service in services:
    print(f"Found k8s-ai at {service['Service']['Address']}:
{service['Service']['Port']}")
```

Mari kita pertimbangkan berbagai pendekatan jaringan untuk mengekspos server k8s-ai.

Pendekatan Jaringan Lintas Klaster

Lingkungan yang berbeda memerlukan strategi jaringan yang berbeda:

- **KinD (pengaturan saat ini)**:
  - Menggunakan jaringan host Docker (host.docker.internal)
  - Layanan NodePort dengan extraPortMappings
  - Sederhana, berfungsi dengan baik untuk pengembangan lokal:

```
extraPortMappings:
- containerPort: 30100
  hostPort: 8100
```

- Penyedia layanan cloud (AWS, GCP, dan Azure):
  - Gunakan layanan LoadBalancer dengan IP publik
  - Atau gunakan jaringan privat dengan peering VPC
  - Atau gunakan gateway transit untuk konektivitas multi-klaster:

```
apiVersion: v1
kind: Service
metadata:
  name: k8s-ai
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  selector:
    app: k8s-ai
  ports:
  - port: 9999
```

```
    targetPort: 9999
```

- Jaringan layanan multi-kluster:
  - Layanan dapat ditemukan secara otomatis di seluruh kluster
  - TLS bersama antar kluster
  - Manajemen dan pengamatan lalu lintas:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    global:
      multiCluster:
        clusterName: worker-cluster
      network: network1
```

Pengujian konektivitas ujung-ke-ujung sangat penting untuk sistem terdistribusi.

Pengujian Konektivitas Ujung-Ke-Ujung

Verifikasi konektivitas lengkap dari MAKDO ke k8s-ai:

```
# Extract kubeconfig
kubectl config view --context kind-makdo-worker --minify --flatten > /tmp/test-
conn.yaml

# Create session with 1-minute TTL
python3 << 'EOF'
import json

with open('/tmp/test-conn.yaml', 'r') as f:
    kubeconfig = f.read()

request = {
    "cluster_name": "connectivity-test",
    "ttl_hours": 0.0167, # 1 minute = 1/60 hours
    "kubeconfig": kubeconfig
}

with open('/tmp/test-conn.json', 'w') as f:
    json.dump(request, f)
EOF

curl -s -X POST 'http://localhost:8099/sessions' \
  -H 'Authorization: Bearer test-key' \
  -H 'Content-Type: application/json' \
  -d '@/tmp/test-conn.json') | python3 -m json.tool
```

Sekarang, uji apakah MAKDO (yang berjalan di klaster kontrol) dapat mencapai titik akhir ini melalui jalur yang sama:

```
# Test from host (simulating MAKDO's perspective)
curl -s 'http://localhost:8100/.well-known/agent.json' \
  -H 'Authorization: Bearer test-key' | python3 -m json.tool | head -15
```

Berikut adalah output dari perintah yang menampilkan kartu agen:

```
{
  "capabilities": {
    "streaming": false
  },
  "defaultInputModes": [
    "text/plain"
  ],
  "defaultOutputModes": [
    "text/plain"
  ],
  "description": "Kubernetes AI diagnostic agent with read-only klaster analysis
with session-based klaster management",
  "name": "k8s-ai Diagnostic Agent"
}
```

Ini mengkonfirmasi jalur lengkap: MAKDO → Host Docker → Klaster pekerja → pod k8s-ai. Sistem yang tangguh untuk industri harus menawarkan pertahanan berlapis untuk memenuhi persyaratan keamanan. Mari kita lihat apa yang disediakan MAKDO dan k8s-ai.

Memahami Arsitektur Keamanan

Penyebaran kami menerapkan beberapa lapisan keamanan:

1. **Otentikasi kunci API**: Baik MAKDO maupun k8s-ai menggunakan otentikasi token bearer.
2. **Akses berbasis sesi**: k8s-ai menggunakan sesi sementara dengan waktu kedaluwarsa.
3. **RBAC**: k8s-ai memiliki izin baca saja di klaster pekerja.
4. **Isolasi jaringan**: Komunikasi mengalir melalui port tertentu dengan akses terkontrol.
5. **Manajemen rahasia**: Kredensial disimpan dalam Kubernetes Secrets dan file .env.

Ini mengikuti praktik terbaik keamanan dan meminimalkan risiko akses jarak jauh, eksfiltrasi data, peningkatan hak akses, pergerakan lateral, dan paparan data sensitif.

Autentikasi Kunci API

Autentikasi kunci API adalah metode umum untuk menyediakan akses aman ke layanan jarak jauh. Mari kita lihat bagaimana k8s-ai mengimplementasikannya.

Autentikasi k8s-ai

k8s-ai memerlukan autentikasi token bearer untuk endpoint A2A dan API Admin-nya. Kunci API disimpan dalam secret Kubernetes yang dibuat selama deployment. Lihat kunci API yang tersimpan:

```
kubectl get secret k8s-ai-api-keys --context kind-makdo-worker \
  -o jsonpath='{.data.keys\.json}' | base64 -d | python3 -m json.tool
```

Berikut adalah hasil yang diharapkan:

```
{
  "api_keys": [
    {
```

```
    "key": "test-key",
    "name": "MAKDO Client",
    "created": "2025-01-01T00:00:00"
   }
  ]
}
```

Uji otentikasi dengan membuat permintaan tanpa kredensial:

```
curl -s 'http://localhost:8099/sessions' | python3 -m json.tool
```

Berikut adalah respons yang diharapkan:

```
{
  "detail": "Not authenticated"
}
```

Sekarang, uji dengan kredensial yang valid:

```
curl -s 'http://localhost:8099/sessions' \
  -H 'Authorization: Bearer test-key' | python3 -m json.tool
```

Ini akan menampilkan daftar sesi aktif, yang mengkonfirmasi bahwa otentikasi berfungsi.

Konfigurasi MAKDO

MAKDO menyimpan kunci API k8s-ai dalam konfigurasinya. Periksa configMap MAKDO:

```
kubectl get configmap makdo-config --context kind-makdo-control \
-o jsonpath='{.data.makdo\.yaml}' | grep -A 3 "a2a_servers:"
```

Berikut adalah output yang diharapkan yang menunjukkan konfigurasi kunci API:

```
a2a_servers:
  - name: "kind-makdo-worker"
    url: "http://host.docker.internal:8100"
    timeout: 30.0
    api_key: "test-key"
```

Dalam produksi, ingatlah bahwa kunci API harus dihasilkan menggunakan nilai acak yang kuat (bukan kunci uji), dirotasi secara teratur, disimpan dalam sistem manajemen rahasia khusus (misalnya, HashiCorp Vault, AWS Secrets Manager, dan lain-lain.), dan tidak pernah dimasukkan ke dalam kontrol versi.

Keamanan Berbasis Sesi

Proyek k8s-ai mengimplementasikan kontrol akses berbasis sesi, menambahkan lapisan keamanan tambahan di luar kunci API. Siklus hidup sesi mencakup hal-hal berikut:

1. **Pembuatan**: MAKDO membuat sesi melalui API Admin, dengan menyediakan kubeconfig.
2. **Penggunaan**: Token sesi digunakan untuk semua operasi diagnostik.
3. **Kedaluwarsa**: Sesi secara otomatis kedaluwarsa setelah **waktu hidup** (TTL) berakhir.
4. **Pembersihan**: Sesi yang kedaluwarsa secara otomatis dihapus.

Mari kita demonstrasikan keamanan sesi sekarang. Buat sesi dengan TTL singkat:

```
# Extract kubeconfig
kubectl config view --context kind-makdo-worker --minify --flatten > /tmp/test-
kubeconfig.yaml

# Create session with 1-minute TTL
python3 << 'EOF'
import json

with open('/tmp/test/test-kubeconfig.yaml', 'r') as f:
    kubeconfig = f.read()

request = {
    "cluster_name": "test-session",
    "ttl_hours": 0.0167,  # 1 minute = 1/60 hours
    "kubeconfig": kubeconfig
}

with open('/tmp/short-session.json', 'w') as f:
    json.dump(request, f)
EOF

curl -s -X POST 'http://localhost:8099/sessions' \
  -H 'Authorization: Bearer test-key' \
  -H 'Content-Type: application/json' \
  -d @/tmp/short-session.json | python3 -m json.tool
```

Respons tersebut menyertakan stempel waktu `expires_at`. Setelah waktu kedaluwarsa, sesi menjadi tidak valid:

```
# Wait 90 seconds
sleep 90

# Try to use the expired session
SESSION_TOKEN="<token-from-creation-response>"

curl -s -X POST 'http://localhost:8100/' \
  -H 'Authorization: Bearer test-key' \
  -H 'Content-Type: application/json' \
  -d '{
    "jsonrpc": "2.0",
    "method": "kubernetes_diagnose_issue",
    "params": {
        "session_token": "\"${SESSION_TOKEN}\"",
        "issue_description": "test"
```

```
    },
    "id": 1
}'
```

Berikut adalah respons yang diharapkan yang menunjukkan bahwa sesi telah berakhir:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32002,
    "message": "Session not found or expired"
  },
  "id": 1
}
```

Akses terbatas waktu ini mengurangi dampak dari kredensial yang disalahgunakan.

Rbac Dan Hak Akses Minimal

k8s-ai beroperasi dengan izin minimal di klaster pekerja.

Memeriksa Konfigurasi Rbac

Lihat sumber KlasterRole yang ditugaskan ke k8s-ai:

```
kubectl get clusterrole k8s-ai-reader --context kind-makdo-worker -o yaml
```

Berikut adalah izin-izin penting:

```
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log", "services", "nodes", "namespaces", "events"]
    verbs: ["get", "list", "watch"]

  - apiGroups: ["apps"]
    resources: ["deployments", "replicasets", "statefulsets", "daemonsets"]
    verbs: ["get", "list", "watch"]

  - apiGroups: ["batch"]
    resources: ["jobs", "cronjobs"]
    verbs: ["get", "list", "watch"]
```

Perhatikan kata kerja hanya baca: get, list, watch. Tidak ada izin untuk membuat, memperbarui, menghapus, atau menambal.

Pengujian Batas Izin

Verifikasi bahwa k8s-ai tidak dapat memodifikasi sumber daya:

```
# Get a pod name
POD_NAME=$(kubectl get pods --context kind-makdo-worker -o
jsonpath='{.items[0].metadata.name}')

# Try to delete it using k8s-ai's service account
kubectl delete pod ${POD_NAME} --context kind-makdo-worker \
  --as=system:serviceaccount:default:k8s-ai
```

Berikut adalah kesalahan yang diharapkan:

```
Error from server (Forbidden): pods "k8s-ai-f77d65f7d-p6lkk" is forbidden:
User "system:serviceaccount:default:k8s-ai" cannot delete resource "pods"
in API group "" in the namespace "default"
```

Ini menunjukkan bahwa meskipun penyerang memperoleh akses ke `k8s-ai`, mereka tidak dapat memodifikasi atau menghapus sumber daya klaster.

Segmentasi Jaringan Di KinD

Meskipun klaster KinD berjalan pada jaringan Docker yang sama, kami mengontrol komunikasi melalui pemetaan port dan konfigurasi layanan.

Keamanan Pemetaan Port

Lihat port yang terekspos:

```
docker ps --filter name=makdo-control-plane --format "table {{.Names}}
\t{{.Ports}}"
docker ps --filter name=makdo-worker-control-plane --format "table {{.Names}}
\t{{.Ports}}"
```

Hasilnya menunjukkan paparan yang terkontrol:

```
NAMES                          PORTS
makdo-control-control-plane    127.0.0.1:xxx->6443/tcp, 0.0.0.0:9000->30000/tcp
makdo-worker-control-plane     127.0.0.1:xxx->6443/tcp, 0.0.0.0:8099->30099/tcp,
0.0.0.0:8100->30100/tcp
```

Berikut beberapa pengamatan penting:

- **API Kubernetes**: Terikat ke `127.0.0.1` (hanya localhost, tidak dapat diakses secara eksternal)
- **Port aplikasi**: Terikat ke `0.0.0.0` (dapat diakses dari host, mensimulasikan akses eksternal)
- **Hanya port tertentu**: Tidak semua port kontainer diekspos

Ini menunjukkan cara mengelola aspek konektivitas dan jaringan Kubernetes di lingkungan KinD sambil mempertahankan isolasi. Sekarang, mari kita lihat cara mengujinya.

Pengujian Isolasi Jaringan

Coba akses API Kubernetes langsung dari host (seharusnya gagal):

```
# Get the worker cluster API server address
WORKER_API=$(kubectl config view --context kind-makdo-worker --minify \
  -o jsonpath='{.clusters[0].cluster.server}')
echo "Worker API: ${WORKER_API}"

# Try to access from outside the cluster (without proper credentials)
curl -k "${WORKER_API}/api/v1/namespaces"
```

Kami mendapatkan kesalahan yang diharapkan karena persyaratan otentikasi:

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "Unauthorized",
  "reason": "Unauthorized",
  "code": 401
}
```

API Kubernetes dilindungi dan hanya dapat diakses dengan kredensial kubeconfig yang tepat.

Peningkatan Keamanan Produksi

Untuk penerapan produksi di luar KinD, terapkan langkah-langkah keamanan tambahan berikut:

1. **Enkripsi TLS/SSL**: Komunikasi HTTP antara MAKDO dan `k8s-ai` di lingkungan lokal (KinD), dan juga aktifkan TLS pada kedua titik akhir untuk produksi:

```
# k8s-ai with TLS
containers:
  - name: k8s-ai
    env:
    - name: TLS_CERT_FILE
      value: "/certs/tls.crt"
    - name: TLS_KEY_FILE
      value: "/certs/tls.key"
    volumeMounts:
    - name: tls-certs
    mountPath: /certs
    readOnly: true
volumes:
  - name: tls-certs
    secret:
      secretName: k8s-ai-tls
```

   Gunakan `cert-manager` atau layanan manajemen sertifikat penyedia cloud Anda.

2. **Kebijakan jaringan**: Batasi komunikasi antar pod:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: k8s-ai-network-policy
spec:
  podSelector:
    matchLabels:
      app: k8s-ai
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: makdo-namespace
    ports:
    - protocol: TCP
      port: 9999
```

3. **Standar keamanan Pod**: Terapkan konteks keamanan:

```
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
  fsGroup: 1000
  seccompProfile:
    type: RuntimeDefault
  capabilities:
    drop:
      - ALL
```

4. **Manajemen rahasia**: Ganti rahasia Kubernetes dengan pengelola rahasia eksternal:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: k8s-ai-secrets
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: vault-backend
    kind: SecretStore
  target:
    name: k8s-ai-secrets
  data:
   - secretKey: api-key
     remoteRef:
       key: k8s-ai/api-key
```

5. **Pencatatan audit**: Aktifkan pencatatan komprehensif:

```
# In k8s-ai code
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
```

```python
# Log all authentication attempts
def authenticate(request):
    logger.info(f"Authentication attempt from {request.client.host}")
    # ... authentication logic
```

6. **Batasan usia**: Melindungi dari penyalahgunaan:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: k8s-ai-rate-limits
data:
  rate_limits.yaml: |
    limits:
      - key: ip
        rate: 100
        per: minute
      - key: api_key
        rate: 1000
        per: hour
```

Pelajaran di sini adalah bahwa menjaga lingkungan produksi yang ketat dan aman bukanlah hal yang sepele, dan Anda harus menerapkan berbagai langkah. Berikut adalah daftar periksa keamanan yang perlu diverifikasi sebelum melakukan deployment ke produksi:

- Semua komunikasi menggunakan enkripsi TLS/SSL
- Kunci API kuat, acak, dan dirotasi secara berkala
- Rahasia dikelola secara eksternal (Vault, AWS Secrets Manager, dan lain-lain.)
- RBAC mengikuti prinsip hak akses minimal
- Kebijakan jaringan membatasi komunikasi yang tidak perlu
- Konteks keamanan Pod memberlakukan eksekusi non-root
- Pencatatan audit menangkap semua upaya akses
- Pembatasan laju melindungi dari penyalahgunaan
- Citra kontainer dipindai untuk kerentanan
- Dependensi selalu diperbarui dengan patch keamanan
- Penyaringan keluar untuk mengurangi eksfiltrasi

Arsitektur keamanan yang ditunjukkan dalam penerapan KinD ini memberikan fondasi yang kokoh, tetapi lingkungan produksi memerlukan langkah-langkah pengerasan tambahan ini.

11.6 RINGKASAN

Dalam bab ini, kami menerapkan sistem multi-agen lengkap yang mencakup dua kluster Kubernetes menggunakan KinD, yang menunjukkan pola dunia nyata untuk arsitektur agen AI terdistribusi. Kami mulai dengan menyiapkan kluster ganda dengan jaringan dan pemetaan port yang tepat, kemudian menerapkan MAKDO ke kluster kontrol dengan empat agen khusus dan server MCP untuk memori dan komunikasi Slack. Pada kluster pekerja, kami menerapkan k8s-ai sebagai agen diagnostik dengan izin RBAC hanya baca, mengekspos titik akhir protokol A2A dan API Admin untuk manajemen sesi. Sepanjang penerapan, kami

menerapkan beberapa lapisan keamanan, termasuk otentikasi kunci API, kontrol akses berbasis sesi dengan kedaluwarsa otomatis, dan isolasi jaringan melalui paparan port terkontrol dan batasan layanan.

Arsitektur yang kami bangun memisahkan tanggung jawab antara agen kontrol yang mengatur dan membuat keputusan versus agen pekerja yang mengeksekusi tugas khusus dengan akses langsung ke kluster. Komunikasi mengalir melalui protokol Agen-ke-Agen menggunakan JSON-RPC melalui HTTP, dengan penemuan layanan statis melalui host.docker.internal untuk lingkungan KinD. Kami mendemonstrasikan cara melacak jalur permintaan di seluruh batas kluster, menguji mekanisme otentikasi dan otorisasi, memverifikasi izin RBAC untuk mencegah modifikasi yang tidak sah, dan mengelola rahasia dengan aman menggunakan Kubernetes Secrets dan variabel lingkungan. Penyebaran ini juga menampilkan pola praktis untuk jaringan lintas kluster, dari pendekatan NodePort sederhana yang cocok untuk pengembangan lokal hingga opsi yang lebih canggih seperti service mesh dan registri layanan dinamis untuk lingkungan produksi.

Meskipun penyebaran berbasis KinD ini memberikan fondasi yang sangat baik untuk memahami arsitektur sistem multi-agen, beralih ke produksi memerlukan pertimbangan tambahan, termasuk enkripsi TLS untuk semua komunikasi, kunci API yang lebih kuat dengan kebijakan rotasi, sistem manajemen rahasia eksternal, pemantauan dan pencatatan yang komprehensif, konfigurasi ketersediaan tinggi, dan pipeline penyebaran otomatis. Pola dan praktik yang ditunjukkan di sini dapat diterapkan mulai dari pengembangan lokal hingga sistem produksi perusahaan di berbagai penyedia cloud, infrastruktur on-premises, dan lingkungan hybrid. Pada bab berikutnya dan terakhir, kita akan menjelajahi masa depan sistem AI multi-agen dan tren yang muncul yang akan membentuk bagaimana Arsitektur ini berkembang.

BAB 12
TOPIK LANJUTAN DAN ARAH MASA DEPAN

12.1 PENDAHULUAN

Kita telah membahas banyak hal dalam buku ini, mulai dari dasar-dasar sistem multi-agen hingga strategi implementasi dan penyebaran praktis. Sekarang adalah waktu yang tepat untuk melihat apa yang akan segera datang. Kita sangat beruntung hidup di era singularitas, yaitu titik dalam sejarah di mana laju kemajuan teknologi begitu cepat sehingga manusia (kecuali mereka ditingkatkan kemampuannya) tidak mampu mengimbanginya. Beberapa ratus tahun yang lalu, teknologi bergerak jauh lebih lambat. Perbedaan dari satu generasi ke generasi berikutnya tidak begitu signifikan, dan kebanyakan orang sama sekali tidak menyadarinya. Revolusi ilmiah, industri, dan informasi semuanya telah mempercepat laju perubahan, tetapi munculnya **kecerdasan buatan umum** (AGI) dan **kecerdasan buatan super** (ASI) akan jauh melampaui semua revolusi sebelumnya baik dalam kecepatan maupun dampaknya.

AI akan mampu melakukan apa pun yang dapat dilakukan manusia dengan lebih baik dan lebih cepat. Ini termasuk aspek-aspek seperti kreativitas, empati, dan pemikiran strategis. Ditambah lagi fakta bahwa AI tidak perlu tidur, makan, atau istirahat, dapat berbagi pengetahuan hanya dengan menyalin sebuah model, dan dapat berkolaborasi dengan miliaran AI lainnya.

Saya pikir, pada titik ini, sebagian besar orang yang berpengetahuan setuju bahwa ini memang benar. Yang tidak disadari kebanyakan orang adalah seberapa cepat hal ini terjadi. Sulit untuk memprediksi secara tepat apa yang akan terjadi di masa depan, tetapi jelas bahwa AI akan memainkan peran sentral dalam membentuknya. Agen AI khususnya akan memiliki peran penting dalam beberapa tahun ke depan, membawa manfaat (dan bahaya) bagi setiap orang.

Dalam bab ini, kita akan mengeksplorasi beberapa tren paling menarik dan penting dalam penelitian dan pengembangan AI dan bagaimana tren tersebut beririsan dengan agen AI, dan membahas implikasinya bagi masyarakat, ekonomi, dan umat manusia secara keseluruhan. Berikut ini adalah topik-topik utama yang dibahas:

- Penalaran superhuman dan pandangan ke depan strategis
- Jendela konteks masif dan implikasinya
- Di luar teks – model multi-modal dan simulasi dunia

Mari kita langsung membahas penalaran superhuman dan pandangan ke depan strategis.

12.2 PENALARAN SUPERHUMAN DAN PANDANGAN KE DEPAN STRATEGIS

Model pembelajaran mesin (LLM) saat ini sudah mampu melakukan hal-hal yang mengesankan dalam penalaran dan pemecahan masalah. Bahkan sebelum era AI generatif, kita telah melihat tanda-tanda pertama AI superhuman di domain sempit seperti catur, Go, dan pelipatan protein. Tetapi ini hanyalah puncak gunung es. Dalam beberapa tahun ke depan,

kita akan melihat sistem AI yang dapat mengungguli manusia dalam berbagai tugas, mulai dari mengemudi mobil hingga mendiagnosis penyakit hingga menulis kode.

Di masa lalu, inovasi teknologi membutuhkan waktu puluhan tahun dari pembuktian konsep hingga adopsi yang luas. Telepon ditemukan pada tahun 1876, tetapi baru pada tahun 1920-an telepon menjadi umum di rumah tangga. Internet ditemukan pada tahun 1960-an, tetapi baru pada tahun 1990-an internet digunakan secara luas. Sebaliknya, AI berkembang dengan kecepatan eksponensial. Model GPT pertama dirilis pada tahun 2018, ChatGPT dirilis pada akhir tahun 2022, dan sekarang kita berada di tahun 2025 (pada saat penulisan) dengan GPT 5.1, Claude, Gemini, dan sejumlah model open source Tiongkok.

Namun, seiring kita terus mengembangkan model dan arsitektur yang lebih canggih, kita dapat mengharapkan untuk melihat sistem AI yang dapat bernalar dan merencanakan pada tingkat yang jauh melampaui kemampuan manusia. Mari kita lihat apa yang ada di cakrawala.

Apa Itu Penalaran Supermanusia?

Penalaran supermanusia mengacu pada sistem AI yang dapat menganalisis masalah, menarik kesimpulan, dan membuat keputusan yang melampaui kemampuan bahkan para ahli manusia yang paling cerdas dan terlatih. Ini bukan hanya tentang kecepatan atau memori, yang jelas unggul dalam sistem AI. Ini sebagian besar tentang kualitas mendasar dari penalaran itu sendiri. Seorang pemikir superhuman dapat mempertimbangkan lebih banyak variabel secara bersamaan, mengidentifikasi pola di seluruh kumpulan data yang luas yang tidak dapat diproses oleh manusia, dan menjelajahi ruang solusi dengan kedalaman dan keluasan yang tidak mungkin dilakukan oleh manusia.

Sebagai contoh, dalam domain catur, grandmaster biasanya melihat ke depan 2–5 langkah, dan dalam posisi tertentu bahkan 10–15 langkah. Program catur seperti AlphaZero dapat secara dinamis mencari pohon permainan dan biasanya melakukan 1.000–2.000 simulasi per langkah. Konsep yang sama berlaku untuk domain lain. Transisi dari sekadar AI yang mengesankan ke penalaran superhuman yang sesungguhnya ditandai oleh beberapa karakteristik kunci: melakukan rantai logika multi-langkah yang tetap koheren di ratusan atau ribuan langkah inferensi, mensintesis wawasan di seluruh domain yang sama sekali berbeda dengan cara yang tidak akan pernah terpikirkan oleh para ahli manusia yang spesifik untuk domain tersebut, dan mengidentifikasi keterbatasan dan ketidakpastian mereka sendiri.

Dalam konteks sistem multi-agen, kemampuan ini menjadi lebih kuat melalui amplifikasi kolaboratif, di mana agen dengan kemampuan penalaran yang saling melengkapi bekerja sama untuk mencapai sinergi yang tidak mungkin dilakukan oleh manusia atau sistem AI individual. Sebagai contoh, Blitzy menggunakan ribuan agen AI untuk mempercepat pengembangan perangkat lunak perusahaan hingga lima kali lipat.

Mengapa Wawasan Strategis Sangat Penting?

Wawasan strategis adalah kemampuan untuk mengantisipasi skenario masa depan, mengidentifikasi tren yang muncul, dan membuat keputusan hari ini yang memperhitungkan konsekuensi jangka panjang. Meskipun manusia selalu terlibat dalam perencanaan strategis, keterbatasan biologis kita membatasi kemampuan kita untuk benar-benar berpikir jauh ke

depan. Kita kesulitan untuk mempertahankan konsistensi di seluruh rantai kausalitas yang kompleks, menjadi korban bias kognitif, dan merasa sulit untuk bernalar tentang perubahan eksponensial atau peristiwa berdampak tinggi dengan probabilitas rendah.

Sistem AI saat ini bahkan berada di bawah level manusia dalam hal wawasan strategis dan kemampuan untuk tetap fokus pada tugas untuk jangka waktu yang lama tanpa tersesat atau terjebak. Tetapi ini berubah dengan cepat. Agen AI awalnya membutuhkan pengawasan dan umpan balik manusia yang ketat setiap beberapa menit dan iterasi konstan, bahkan pada tugas-tugas sederhana. Pada saat penulisan ini di tahun 2025, beberapa sistem AI agen dapat beroperasi secara otonom selama berjam-jam dan melakukan alur kerja dan tugas yang kompleks dengan sukses.

Dalam satu atau dua tahun ke depan, sistem AI akan mampu memodelkan sistem rumit dengan ribuan variabel yang saling bergantung, mensimulasikan skenario yang tak terhitung jumlahnya secara paralel, dan mengidentifikasi risiko dan peluang yang akan tetap tidak terlihat oleh para ahli strategi manusia. Kemampuan ini menjadi semakin penting seiring dengan percepatan laju perubahan teknologi dan meningkatnya kompleksitas tantangan global. Pertimbangkan isu-isu seperti perubahan iklim, pandemi, masalah sosial seperti distribusi kekayaan, dan, tentu saja, keselarasan AI.

Dalam sistem multi-agen, wawasan strategis mengambil dimensi kekuatan dan kompleksitas tambahan. Beberapa agen AI dapat berkolaborasi untuk mengeksplorasi berbagai skenario masa depan secara bersamaan, dengan beberapa agen berfokus pada jalur optimis, yang lain pada lintasan pesimis, dan yang lainnya pada peristiwa tak terduga yang mungkin mengganggu perencanaan konvensional. Agen-agen ini dapat terlibat dalam penalaran yang saling bertentangan, dengan beberapa berperan sebagai pesaing strategis atau gangguan lingkungan sementara yang lain merancang tindakan balasan dan strategi adaptif. Bagi organisasi dan masyarakat yang bersedia merangkul kemampuan ini, wawasan strategis berbasis AI dapat menjadi perbedaan antara berkembang di masa depan yang tidak pasti dan dikejutkan oleh kejutan yang dapat diprediksi.

Selain itu, sistem multi-agen dapat terlibat dalam peramalan yang bersifat antagonis di mana beberapa agen berdiskusi dan berdebat satu sama lain hingga mereka mencapai konsensus. Namun, ada juga tantangan dan pertimbangan etis yang signifikan terkait dengan penalaran supermanusia dan peramalan strategis.

Tantangan Dan Keterbatasan Penalaran Supermanusia

Yang pertama dan terpenting adalah masalah interpretasi. Seiring sistem AI menjadi lebih kompleks dan mampu, memahami bagaimana mereka sampai pada kesimpulan mereka menjadi semakin sulit. Masalah "kotak hitam" ini menimbulkan kekhawatiran tentang kepercayaan dan akuntabilitas, terutama ketika sistem AI membuat keputusan yang berdampak pada kehidupan manusia. Lihat, misalnya, studi Stanford ini tentang program medis AI: https://hai.stanford.edu/news/peering-black-box-ai-medical-programs. Jika sistem AI merekomendasikan tindakan berdasarkan penalaran supermanusianya, tetapi manusia tidak dapat memahami alasan di balik rekomendasi tersebut, maka menjadi sulit untuk mengevaluasi validitasnya atau menantang kesimpulannya.

Keselarasan AI adalah inti dari tantangan ini. Ketika manusia tidak dapat sepenuhnya memahami proses penalaran sistem AI supermanusia, memastikan bahwa tujuan dan nilai-nilai sistem ini selaras dengan kepentingan manusia menjadi tugas yang menakutkan. Ketidakselarasan dapat menyebabkan konsekuensi yang tidak diinginkan, di mana sistem AI mengejar tujuan yang merugikan umat manusia, bahkan jika tujuan tersebut secara logis konsisten dalam kerangka kerja mereka sendiri.

Tantangan kedua adalah potensi ketergantungan yang berlebihan pada sistem AI. Ketika sistem AI menunjukkan kemampuan penalaran supermanusia, ada risiko bahwa manusia menjadi terlalu bergantung pada mereka untuk pengambilan keputusan, yang berpotensi mengikis keterampilan dan penilaian manusia. Ketergantungan ini dapat menyebabkan situasi di mana manusia tidak siap untuk campur tangan atau membuat keputusan ketika sistem AI gagal atau berperilaku tidak terduga. Sifat dasar kemanusiaan mungkin berubah karena kita semua menjadi sepenuhnya bergantung pada AI untuk menjalankan hidup kita.

Terakhir, ada pertimbangan etis seputar penggunaan penalaran supermanusia dalam peramalan strategis. Kemampuan untuk memprediksi dan memengaruhi skenario masa depan menimbulkan pertanyaan tentang dinamika kekuasaan, kesetaraan, dan keadilan. Siapa yang berhak memutuskan masa depan mana yang dikejar dan mana yang dihindari? Bagaimana kita memastikan bahwa manfaat dari kemampuan prediksi berbasis AI didistribusikan secara adil di seluruh masyarakat, alih-alih terkonsentrasi di tangan beberapa aktor berpengaruh?

Implikasi Bagi Masyarakat Dan Etika

Berikut prediksi saya untuk lima tahun ke depan. Dalam dua tahun, semua pekerjaan berbasis pengetahuan akan dapat dilakukan oleh agen AI. Ini termasuk penelitian, analisis, penulisan, desain, dan pemrograman. Kreativitas, penilaian, empati, dan atribut lain yang masih dianggap oleh sebagian orang sebagai ranah manusia juga akan dengan cepat dikuasai oleh AI. AI akan jauh lebih kreatif di bidang apa pun, ia akan mampu mengawasi dan menilai dirinya sendiri atau AI lain dengan lebih baik, dan ia akan jauh lebih berempati daripada manusia, dengan kesabaran dan perhatian yang tak terbatas serta pemahaman yang lebih baik tentang manusia. Kemudian, dalam tiga hingga lima tahun, robot dengan AI canggih akan mampu mengambil alih semua pekerjaan kasar. Sekali lagi, ini tidak berarti bahwa manusia akan berhenti bekerja. Tetapi robot yang menggunakan AI yang memahami dunia fisik akan mampu melakukan tugas fisik apa pun dengan lebih baik daripada manusia.

Kecepatan adopsi yang sebenarnya akan bergantung pada banyak faktor, termasuk lingkungan peraturan, penerimaan sosial, insentif ekonomi, dan terobosan teknologi. Namun, lintasannya jelas. Kita bergerak menuju masa depan di mana sistem AI dengan penalaran supermanusia dan kemampuan pandangan strategis ke depan memainkan peran sentral dalam membentuk dunia kita.

Masa depan ini sama menariknya sekaligus menakutkan. Masyarakat akan mengalami perubahan cepat karena identitas banyak orang terkait dengan pekerjaan mereka. Implikasi ekonominya juga sangat besar. Dengan sistem AI yang mampu melakukan sebagian besar pekerjaan, model tradisional pekerjaan dan distribusi pendapatan akan ditantang. Semua

orang membicarakan **pendapatan dasar universal** (UBI) sebagai solusi yang mungkin. Tetapi mungkin ada solusi lain. Ekonomi secara keseluruhan akan tumbuh jauh lebih cepat karena produktivitas meningkat pesat. Tetapi distribusi kekayaan itu akan menjadi masalah politik dan sosial utama. Jika kita melihat tren terkini, kita dapat melihat bahwa kesenjangan kekayaan semakin melebar dan kelas menengah semakin menyusut. Jika kita tidak menemukan cara untuk memastikan bahwa manfaat AI dibagikan secara luas di seluruh masyarakat, kita mungkin menghadapi peningkatan keresahan dan ketidakstabilan sosial. Ini akan buruk bagi semua orang, termasuk elit kaya.

Mari kita asumsikan kita dapat mengatasi tantangan ini dan membagikan kelimpahan yang diciptakan oleh AI secara luas di seluruh masyarakat. Masih belum jelas apa yang akan terjadi pada manusia. Apakah kita akan hidup sebagian besar di dunia virtual? Apa tujuan hidup manusia ketika hampir semua hal dapat dilakukan dengan lebih baik oleh AI? Ini adalah pertanyaan filosofis mendalam yang harus dihadapi umat manusia di tahun-tahun mendatang. Oke. Kembali ke topik yang lebih teknis.

12.3 JENDELA KONTEKS MASIF DAN IMPLIKASINYA

Jendela konteks model adalah aspek paling penting yang menentukan seberapa banyak informasi yang dapat dipertimbangkan model sekaligus. Model dapat menggunakan alat untuk mengakses dan memproses informasi tambahan dan mendelegasikan pekerjaan melalui sistem multi-agen, tetapi jendela konteks menetapkan batasan mendasar tentang seberapa banyak informasi yang dapat dipertimbangkan model secara langsung dalam satu kali proses.

Ukuran jendela konteks LLM telah tumbuh secara eksponensial selama bertahun-tahun, memungkinkan model untuk memproses urutan teks yang semakin panjang. Ekspansi ini telah menjadi pendorong utama peningkatan kemampuan AI.

Evolusi Jendela Konteks

Pertama, mari kita tinjau sejarah ukuran jendela konteks dalam LLM untuk memahami seberapa cepat pertumbuhannya. Ingat, arsitektur Transformer diperkenalkan pada tahun 2017, jadi kita melihat kurang dari satu dekade kemajuan eksponensial di sini:

- **2018–2019**: **Era awal**: Model awal seperti BERT dan GPT-1 hanya memiliki jendela konteks 512 token, yang membatasi mereka untuk memproses paragraf pendek dalam satu waktu.
- **2020–2021**: **Era GPT-3:** GPT-3 memperkenalkan jendela konteks 2.048 token, peningkatan 4x, yang memungkinkan dokumen yang lebih panjang dan alur percakapan yang lebih baik.
- **2022–2023**: **Ekspansi cepat**: GPT-3.5 memperluas konteks menjadi 16.000 token (lompatan 8x). GPT-4 menawarkan varian 8K dan 32K token, dengan beberapa versi mencapai 128K token.
- **2024–2025**: **Era jutaan token**: Model dengan jendela konteks 1 juta token muncul (peningkatan 2.000x dari GPT-3). Anehnya, model OpenAI memiliki jendela konteks yang relatif sederhana dibandingkan dengan pesaing, dengan maksimal 400K untuk GPT-5.

Claude Sonnet 4 dan Gemini 2.5-3 keduanya menawarkan 1 juta token. Llama 4 mendorong batas melampaui 10 juta token. Bahkan ada model penelitian dari magic.dev dengan jendela konteks 100 juta token!

Tingkat pertumbuhan eksponensial ini secara fundamental telah mengubah cara kita berinteraksi dengan sistem AI.

Apa Yang Dapat Anda Lakukan Dengan Jendela Konteks Jutaan Token?

PT pada *model GPT* merupakan singkatan dari *pre-trained, yang berarti telah dilatih* sebelumnya. Pada awalnya, dengan jendela konteks yang kecil, kemampuan model sangat bergantung pada data pra-pelatihan dan penyempurnaan. Dengan jendela konteks yang lebih besar, model dapat secara efektif melakukan pembelajaran pada waktu inferensi dengan menggabungkan sejumlah besar informasi relevan langsung ke dalam konteks. Ini mengurangi ketergantungan pada pra-pelatihan dan memungkinkan model untuk beradaptasi dengan tugas-tugas spesifik secara langsung, menggabungkan banyak informasi spesifik tugas, respons panggilan alat, dan panduan pengguna.

Untuk memberi Anda gambaran skalanya, 1 juta token kira-kira setara dengan 750.000 kata atau sekitar 1.500 halaman teks. Berikut beberapa aplikasi praktis yang dimungkinkan oleh jendela konteks yang sangat besar tersebut:

- **Analisis basis kode yang komprehensif**: Model dapat menyerap dan menalar jutaan baris kode sekaligus, membantu dalam refactoring, tinjauan kode, dokumentasi, dan perubahan arsitektur skala besar tanpa memecah atau kehilangan struktur tingkat tinggi. Lihat https://deepwiki.com/ untuk contoh analisis repositori GitHub apa pun.
- **Pemahaman dokumen yang holistik**: Para profesional hukum, medis, dan keuangan dapat menganalisis, meringkas, dan merujuk silang konten dan hubungan di ratusan atau ribuan dokumen dalam satu kali proses. Misalnya, AI dapat melakukan tinjauan kontrak hukum lengkap atau menyerap riwayat medis yang sangat banyak. Lihat https://www.harvey.ai/solutions/litigation.
- **Memori percakapan yang mendalam**: Chatbot atau agen dukungan dapat merujuk seluruh percakapan multi-sesi, memberikan bantuan yang benar-benar personal dan peka konteks bahkan ketika diskusi berlangsung berhari-hari atau berminggu-minggu. Lihat bagaimana Perplexity mengelola memori di sini: https://www.perplexity.ai/help-center/en/articles/10968016-memory.
- **Pembuatan konten kreatif**: Penulis, peneliti, dan kreator multimedia dapat mempertahankan konsistensi dalam nada, alur cerita, dan referensi saat membuat atau mengedit buku, skrip, atau alur cerita multi-episode yang panjang, semuanya dengan model yang tetap memperhatikan setiap detail. Lihat studi dan tolok ukur ini tentang pembuatan bentuk panjang: https://proceedings.iclr.cc/paper_files/paper/2025/file/141304a37d59ec7f116f3535f1b74bde-Paper-Conference.pdf.
- **Penelitian dan analisis tingkat lanjut**: AI dapat menganalisis dan mensintesis tren di seluruh publikasi, laporan pasar, atau makalah ilmiah, mengidentifikasi koneksi halus atau pola yang muncul yang tidak mungkin ditemukan saat memproses dokumen

secara terpisah. Sebagai contoh, lihatlah ilmuwan AI yang dikembangkan oleh Sakana AI di https://sakana.ai/ai-scientist/.

Jendela konteks yang besar bukanlah solusi mujarab. Mari kita lihat beberapa masalah yang ditimbulkannya.

Tantangan Dan Pertimbangan Jendela Konteks Yang Besar

Konteks yang besar memang bagus, tetapi model saat ini seringkali gagal memanfaatkan kapasitas konteks penuhnya secara efektif. Fenomena ini dikenal sebagai "kerusakan konteks." Penurunan kinerja terwujud dalam berbagai cara: akurasi yang berkurang pada informasi konteks tengah, waktu inferensi yang lebih lambat, dan peningkatan tingkat halusinasi karena konteks yang tidak relevan mengencerkan sinyal. Masalah "tersesat di tengah" terus berlanjut di seluruh model terkemuka, menunjukkan keterbatasan arsitektur mendasar daripada masalah implementasi. Secara umum, lebih banyak konteks tidak berarti lebih banyak kecerdasan. Algoritma yang lebih baik dan cara-cara canggih untuk memadatkan dan memangkas konteks sedang aktif diteliti untuk mengatasi tantangan ini. Misalnya, makalah Nested Learning dari riset Google menunjukkan banyak potensi.

Jendela Konteks Dan Sistem Multi-Agen

Sistem multi-agen sangat diuntungkan dari jendela konteks yang besar. Agen dapat berbagi informasi status, rencana, dan pengamatan yang ekstensif dalam satu konteks, memungkinkan setiap agen untuk mengambil lebih banyak tanggung jawab dan mengurangi biaya koordinasi. Agen juga dapat mempertahankan memori jangka panjang dari interaksi masa lalu, memungkinkan kolaborasi yang lebih koheren dalam jangka waktu yang lama. Misalnya, dalam tim agen AI yang mengerjakan proyek kompleks, setiap agen dapat merujuk pada seluruh riwayat proyek, keputusan yang dibuat, dan alasan di baliknya tanpa perlu berulang kali menanyakan basis pengetahuan pusat. Hal ini menghasilkan alur kerja yang lebih efisien dan hasil keseluruhan yang lebih baik.

Masa Depan – Konteks Tak Terbatas Dan Pembelajaran Berkelanjutan

Salah satu arah penelitian yang paling menarik adalah pembelajaran berkelanjutan. Model mungkin masih melalui fase pra-pelatihan yang diikuti oleh pembelajaran penguatan dan penyempurnaan. Tetapi saat model diterapkan, model tersebut akan mampu mempertahankan representasi dunia yang persisten. Model dapat terus belajar dan beradaptasi selama inferensi. Ini membuka kemungkinan bagi sistem AI yang berevolusi dari waktu ke waktu, menggabungkan informasi, umpan balik, dan pengalaman baru secara langsung ke dalam proses penalaran mereka tanpa memerlukan pelatihan ulang. Klien tidak perlu mempertahankan konteks dan merekayasanya dengan cermat untuk kepentingan model. Sebaliknya, model tersebut akan mampu mengingat semua yang telah dilihat dan dipelajari di masa lalu dan menggunakan pengetahuan itu untuk menginformasikan keputusan dan tindakannya di masa depan.

Arsitektur Untuk Perencanaan Jangka Panjang

Ukuran yang baik untuk mengevaluasi kecerdasan sistem AI adalah kemampuannya untuk merencanakan dalam jangka waktu yang panjang. Manusia mampu menetapkan tujuan dan merancang strategi yang mencakup berbulan-bulan, bertahun-tahun, atau bahkan puluhan tahun. Kemampuan untuk berpikir ke depan dan mengantisipasi konsekuensi masa

depan ini merupakan ciri khas kecerdasan tingkat lanjut. Sistem AI saat ini, meskipun mengesankan dalam banyak hal, seringkali kesulitan dengan perencanaan jangka panjang.

Tantangan Perencanaan Jangka Panjang

Perencanaan dalam jangka waktu yang panjang menghadirkan beberapa tantangan bagi sistem AI. Pertama, kompleksitas masalah perencanaan meningkat secara eksponensial dengan panjang jangka waktu. Jumlah kemungkinan keadaan dan tindakan di masa depan tumbuh dengan cepat, sehingga menyulitkan sistem AI untuk mengevaluasi semua hasil potensial secara efektif. Kedua, ketidakpastian memainkan peran penting dalam perencanaan jangka panjang. Semakin jauh ke masa depan sistem AI mencoba merencanakan, semakin banyak variabel dan hal yang tidak diketahui yang harus dihadapinya. Ketidakpastian ini dapat menyebabkan keputusan yang suboptimal jika tidak dikelola dengan benar.

Perencanaan Hierarkis Dan Abstraksi

Salah satu pendekatan yang menjanjikan untuk perencanaan jangka panjang adalah **perencanaan hierarkis**, yang tentu saja sangat relevan untuk sistem multi-agen. Ini melibatkan pemecahan tugas-tugas kompleks menjadi sub-tugas yang lebih kecil dan lebih mudah dikelola, masing-masing dengan tujuan dan strateginya sendiri. Dengan beroperasi pada berbagai tingkat abstraksi, sistem AI dapat fokus pada tujuan tingkat tinggi sambil mendelegasikan detailnya ke proses tingkat rendah. Ini mencerminkan strategi kognitif manusia, di mana kita sering berpikir dalam hal tujuan menyeluruh dan kemudian merencanakan langkah-langkah spesifik yang diperlukan untuk mencapainya.

Perencanaan Berbasis Model Dan Model Dunia

Perencanaan berbasis model adalah teknik kunci lain untuk perencanaan jangka panjang. Pendekatan ini melibatkan pembuatan model lingkungan yang dapat digunakan sistem AI untuk mensimulasikan skenario masa depan dan mengevaluasi tindakan potensial. Dengan memprediksi konsekuensi dari berbagai strategi, sistem AI dapat membuat keputusan yang lebih tepat yang memperhitungkan hasil jangka panjang. Model dunia, yang memberikan representasi komprehensif dari lingkungan, sangat penting untuk perencanaan berbasis model yang efektif. Mereka memungkinkan sistem AI untuk memahami dinamika dunia dan bagaimana tindakan mereka akan memengaruhi keadaan di masa depan. Model dunia umum Runway berfokus pada paradigma ini.

Misalnya, domain kompleks seperti biologi manusia membutuhkan pemahaman tentang interaksi rumit antara berbagai sistem biologis. Sistem AI dengan model sel virtual yang kuat dapat mensimulasikan efek intervensi medis dari waktu ke waktu, membantu merancang rencana perawatan yang mempertimbangkan hasil kesehatan jangka panjang. Mari kita lihat beberapa domain menarik lainnya yang sedang direvolusi oleh model AI canggih di luar sekadar bahasa.

12.4 MODEL MULTI-MODAL DAN SIMULASI DUNIA

LLM telah menunjukkan kemampuan luar biasa dalam memahami dan menghasilkan bahasa manusia. Namun, potensi arsitektur transformer meluas jauh melampaui teks.

Transformer Untuk Segalanya: Visi, Audio, Video, Dan Seterusnya

Para peneliti secara aktif mengeksplorasi bagaimana model-model ini dapat diadaptasi untuk memproses dan menghasilkan informasi di berbagai modalitas, termasuk visi, audio, video, dan bahkan domain seperti kimia dan cuaca. Pendekatan multi-modal ini membuka jalan baru untuk aplikasi AI dan meningkatkan kemampuan sistem AI untuk berinteraksi dengan dunia secara lebih holistik. Bahkan model generasi saat ini dapat meniru suara siapa pun dan menciptakan video fantastis dan super-realistis yang tidak dapat dibedakan dari video nyata. Misalnya, ElevenLabs memiliki teknologi yang disebut kloning suara.

Kemampuannya luar biasa dan sekaligus menakutkan karena semakin sulit untuk membedakan antara konten asli dan sintetis. Hal ini memiliki implikasi mendalam bagi media, hiburan, pendidikan, dan banyak bidang lainnya.

Model Biologis: Protein, Dna, Dan Desain Molekuler

Salah satu bidang yang paling menarik dari AI multimodal adalah di ranah pemodelan biologis. Sistem AI sedang dikembangkan untuk memahami dan memprediksi perilaku sistem biologis kompleks, seperti protein, DNA, dan proses seluler. Model-model ini berpotensi merevolusi penemuan obat, pengobatan personal, dan pemahaman kita tentang kehidupan itu sendiri.

Bidang penelitian AI ini sangat relevan untuk sistem multi-agen, karena agen yang berbeda dapat mengkhususkan diri dalam berbagai aspek pemodelan biologis, dari dinamika molekuler hingga analisis genetika. Dengan berkolaborasi, agen-agen ini dapat mengatasi masalah biologis kompleks yang membutuhkan keahlian interdisipliner.

Aplikasi-aplikasinya sangat menarik, termasuk menyembuhkan semua penyakit, memperpanjang umur manusia yang sehat, meregenerasi jaringan dan organ yang rusak, dan bahkan menciptakan bentuk kehidupan yang sepenuhnya baru.

Dari Model Bahasa Hingga Model Dunia Dan Simulasi

Seiring dengan berlakunya hukum penskalaan, dan model-model tersebut dapat menyimpan dan memproses semakin banyak data, kita dapat mengharapkan munculnya sistem AI yang dapat menciptakan simulasi terperinci dari lingkungan yang kompleks. Model dunia ini akan memungkinkan sistem AI untuk bernalar tentang ruang fisik, dinamika sosial, dan bahkan konsep abstrak dengan cara yang lebih selaras dengan kognisi manusia.

Sistem Multi-Modal Multi-Agen

Sistem AI multi-modal dapat memperoleh manfaat besar dari arsitektur multi-agen. Agen yang berbeda dapat berspesialisasi dalam memproses dan menghasilkan informasi di berbagai modalitas, memungkinkan kolaborasi yang lebih efisien dan efektif. Misalnya, dalam sistem multi-agen yang dirancang untuk kendaraan otonom, satu agen dapat fokus pada persepsi visual, agen lain pada pemrosesan audio, dan agen ketiga pada pengambilan keputusan dan perencanaan. Dengan bekerja sama, agen-agen ini dapat menciptakan pemahaman yang lebih komprehensif tentang lingkungan dan membuat keputusan yang lebih baik.

Merancang Pengalaman Pengguna Generatif

Antarmuka pengguna saat ini dirancang untuk khalayak umum. Beberapa produk menawarkan kustomisasi, tetapi itu seringkali membutuhkan banyak pekerjaan. **Pengalaman**

pengguna generatif (GenUX) adalah tentang menciptakan antarmuka pengguna yang dipersonalisasi dan adaptif yang memanfaatkan AI untuk menyesuaikan pengalaman dengan kebutuhan, preferensi, dan perilaku pengguna individu. Asisten AI ini memahami konteks pengguna, mengantisipasi kebutuhan mereka, dan memberikan informasi dan bantuan yang relevan dengan cara yang mulus.

Jika kita melangkah lebih jauh, setiap piksel yang dilihat pengguna di layar mereka atau di lingkungan virtual mereka dapat dihasilkan oleh model AI khusus untuk mereka pada saat itu dan dicampur dengan realitas. Tata letak, warna, konten, suara, dan interaksi semuanya akan dibuat secara dinamis berdasarkan preferensi, tujuan, dan konteks pengguna. Tingkat personalisasi ini melampaui opsi kustomisasi tradisional, memungkinkan pengguna untuk memiliki pengalaman yang benar-benar unik yang beradaptasi dengan kebutuhan mereka yang terus berkembang.

Perkembangan di ranah digital ini akan mengubah hidup kita karena bahkan saat ini, orang menghabiskan lebih banyak waktu di ponsel dan perangkat lain mereka. Batas antara dunia fisik dan digital akan semakin kabur seiring konten yang dihasilkan AI menjadi tak dapat dibedakan dari kenyataan. Namun, dunia fisik dan cara kita berinteraksi dengannya juga akan berubah.

Robot Akan Datang

Agen AI akan mampu melakukan semua pekerjaan berbasis pengetahuan dalam dua tahun ke depan. Tetapi bagaimana dengan pekerjaan fisik? Robot telah ada selama beberapa dekade, tetapi sebagian besar terbatas pada lingkungan terkontrol seperti pabrik. Kemajuan terbaru dalam AI dan robotika memungkinkan pengembangan robot yang dapat beroperasi di lingkungan yang tidak terstruktur dan melakukan berbagai tugas. Saya percaya bahwa dalam tiga hingga lima tahun ke depan, kita akan melihat robot dengan AI canggih mulai mengambil alih semua pekerjaan kasar. Ini termasuk pekerjaan seperti membersihkan, memasak, pengiriman, konstruksi, pemeliharaan, dan bahkan perawatan. Trennya jelas. Robot humanoid dari perusahaan seperti Figure, Unitree, 1X, Tesla, dan Agility Robotics menjadi semakin mampu dan terjangkau.

Itu tidak berarti bahwa semua pekerjaan akan diotomatisasi. Pertama, tidak seperti agen AI perangkat lunak yang dapat direplikasi dengan biaya mendekati nol, robot fisik memiliki persyaratan manufaktur, pemeliharaan, dan operasional yang signifikan. Kedua, saya percaya akan membutuhkan waktu lebih lama untuk mengintegrasikan robot ke dalam lingkungan yang kompleks dan berinteraksi dengan aman dengan manusia. Namun, robot akan mampu melakukan tugas apa pun dalam beberapa tahun ke depan.

Konvergensi AI Dan Robotika

Perangkat keras robotika telah meningkat secara signifikan dalam beberapa tahun terakhir, dengan kemajuan dalam sensor, aktuator, dan material. Namun, terobosan nyata datang dari integrasi model AI canggih ke dalam sistem robotika. Robot bertenaga AI ini dapat merasakan lingkungannya, membuat keputusan, dan belajar dari pengalaman mereka dengan cara yang sebelumnya tidak mungkin.

Agen AI Yang Terwujud

Ketika model dengan kemampuan penalaran dan perencanaan tingkat lanjut diintegrasikan ke dalam platform robotika, kita mendapatkan agen AI yang terwujud. Agen-agen ini dapat berinteraksi dengan dunia fisik, memanipulasi objek, dan menavigasi lingkungan yang kompleks. Mereka dapat melakukan tugas yang membutuhkan ketangkasan, kemampuan beradaptasi, dan kesadaran situasional.

Robot Humanoid Dan Kecerdasan Fisik Tujuan Umum

LLM dapat dilatih di internet dan belajar dari penyerapan sejumlah besar data yang relatif mudah dikumpulkan. Tetapi kecerdasan fisik membutuhkan pengalaman yang terwujud di dunia nyata. Robot perlu belajar bagaimana bergerak, memanipulasi objek, dan berinteraksi dengan lingkungannya. Hal ini membutuhkan pendekatan yang berbeda untuk pelatihan dan pembelajaran. Teknik-teknik seperti pembelajaran penguatan (reinforcement learning), pembelajaran imitasi (imitation learning), dan transfer simulasi ke dunia nyata (sim-to-real transfer) digunakan untuk mengajari robot cara melakukan tugas-tugas kompleks. Tujuannya adalah untuk menciptakan kecerdasan fisik serbaguna yang dapat beradaptasi dengan berbagai tugas dan lingkungan.

Ada beberapa pendekatan menarik, seperti melengkapi pekerja manusia dengan sarung tangan khusus yang merekam setiap nuansa gerakan tangan mereka saat melakukan berbagai tugas, menempatkan kamera di ujung jari tangan robot untuk menangkap umpan balik visual terperinci selama manipulasi, dan menggunakan pengaturan VR/AR untuk mensimulasikan lingkungan kompleks di mana manusia dapat mendemonstrasikan tugas agar robot dapat belajar. Metode-metode ini membantu robot memperoleh keterampilan motorik halus dan pemahaman kontekstual yang dibutuhkan untuk kecerdasan fisik tujuan umum.

Koordinasi Multi-Agen Dan Multi-Robot Di Ruang Fisik

Setiap robot akan memiliki otak dengan AI multi-agen, tetapi beberapa robot juga dapat bekerja sama untuk menyelesaikan tugas-tugas kompleks. Sistem multi-robot akan mengoordinasikan tindakan mereka, berbagi informasi, dan beradaptasi dengan lingkungan yang berubah. Misalnya, tim robot pengiriman dapat berkolaborasi untuk mengoptimalkan rute, menghindari rintangan, dan memastikan pengiriman tepat waktu. Dalam konstruksi, beberapa robot akan bekerja sama untuk merakit struktur, mengangkut material, dan melakukan inspeksi. Setiap kali robot tertentu menghadapi situasi baru atau mempelajari sesuatu yang baru, ia akan membagikan pengetahuan itu dengan seluruh armada robot. Berbagi pengetahuan yang efisien ini mempercepat pembelajaran dan adaptasi di seluruh sistem.

Tantangan Dalam Penerapan Di Dunia Nyata

Namun, menerapkan robot di lingkungan dunia nyata menghadirkan tantangan yang signifikan. Keselamatan adalah yang terpenting, karena robot harus mampu beroperasi bersama manusia tanpa menimbulkan bahaya. Ketahanan dan keandalan juga sangat penting, karena robot perlu berfungsi secara efektif dalam kondisi yang beragam dan tidak dapat diprediksi. Selain itu, pertimbangan etis seputar privasi, perpindahan pekerjaan, dan interaksi manusia-robot harus ditangani.

Ekonomi AI

Revolusi AI dan robotika akan mengubah perekonomian. Agen AI di ranah digital dan yang tertanam dalam robot akan mampu melakukan pekerjaan apa pun dengan lebih baik daripada manusia. Mari kita lihat bagaimana hal ini akan terjadi.

Mata Uang Kripto Dan Agen AI Sebagai Pelaku Ekonomi

Agen AI dengan akses ke mata uang kripto dan aset digital akan dapat berpartisipasi dalam perekonomian sebagai pelaku ekonomi otonom. Mereka akan dapat membeli dan menjual barang dan jasa, berinvestasi dalam aset, dan mengelola keuangan mereka sendiri. Hal ini membuka kemungkinan baru untuk aktivitas ekonomi, karena agen AI dapat beroperasi 24/7, membuat keputusan berdasarkan sejumlah besar data, dan mengeksekusi transaksi dengan kecepatan kilat.

Jika saat ini asisten AI membantu manusia, sebentar lagi agen AI akan mampu beroperasi secara otonom atas nama pemiliknya dan benar-benar merekrut jasa manusia untuk tugas-tugas yang belum diotomatisasi atau karena peraturan yang membutuhkan persetujuan manusia. Pertimbangkan (dalam beberapa tahun) sistem AI agen otonom yang bertanggung jawab atas proyek konstruksi besar (misalnya, pusat data baru). Meskipun agen AI akan dapat berkomunikasi dengan agen AI lainnya untuk perencanaan, pemesanan semua material dan peralatan, pengamanan pendanaan, dan komputasi yang diperlukan, mereka mungkin masih perlu mendapatkan berbagai izin dari badan pemerintah yang mengikuti proses lama, termasuk pengawas manusia dengan papan klip dan formulir kertas.

Ini bukan masalah. Agen AI akan mempekerjakan karyawan manusia dengan keterampilan dan pengalaman yang tepat untuk berinteraksi dengan pengawas manusia. Bahkan nanti, kita mungkin akan melihat pasar agen-ke-agen yang sepenuhnya terintegrasi di mana hanya agen yang berinteraksi, dengan kecepatan super manusia. Moltbook adalah contoh lingkungan agen-ke-agen sepenuhnya, meskipun ini adalah jejaring sosial dan bukan pasar.

Protokol Pembayaran Untuk Agen Otonom: AP2 Dan Seterusnya

Masa depan ini membutuhkan substrat protokol keuangan yang memungkinkan transaksi yang aman, efisien, dan tanpa kepercayaan antara agen AI. Sistem keuangan lama sangat tidak efisien, membutuhkan beberapa hari kerja untuk menyelesaikan pembayaran, menimbulkan biaya tinggi, dan membutuhkan intervensi manusia. Mata uang kripto dan teknologi blockchain memberikan fondasi yang menjanjikan untuk memungkinkan aktivitas ekonomi otonom. Stablecoin yang dipatok ke mata uang fiat memberikan solusi yang menawarkan kecepatan kripto dengan keamanan instrumen keuangan tradisional. Namun, protokol khusus diperlukan untuk mengatasi persyaratan unik agen AI.

Protokol Pembayaran Otonom (AP2) adalah salah satu protokol yang sedang berkembang yang dirancang untuk memfasilitasi pembayaran dan pertukaran nilai antara agen otonom. AP2 menyediakan kerangka kerja bagi agen AI untuk menegosiasikan harga, mengeksekusi kontrak, dan menyelesaikan pembayaran tanpa intervensi manusia. Protokol ini memastikan bahwa transaksi bersifat transparan, dapat diverifikasi, dan dapat ditegakkan, sehingga memungkinkan ekosistem aktivitas ekonomi berbasis AI yang berkembang pesat.

Robot Mempercepat Siklus Peningkatan Diri AI

Janji utama AI adalah kemampuannya untuk meningkatkan diri secara iteratif. Model pembelajaran mesin (LLM) saat ini dapat disempurnakan dan ditingkatkan dari waktu ke waktu berdasarkan data dan umpan balik baru. Namun, proses ini masih membutuhkan keterlibatan manusia yang signifikan. Hukum penskalaan membutuhkan banyak daya komputasi untuk melatih dan menyempurnakan model besar. Robot dengan AI canggih dapat membantu menutup siklus ini dengan mengumpulkan data secara otomatis, melakukan eksperimen, dan menyempurnakan algoritma mereka sendiri.

Sebagai contoh, armada robot penelitian dapat menjelajahi lingkungan baru, mengumpulkan data tentang fenomena fisik, dan menggunakan data tersebut untuk meningkatkan model dan kemampuan mereka sendiri. Siklus peningkatan diri ini dapat mempercepat laju pengembangan AI, yang mengarah pada kemajuan pesat dalam kecerdasan dan kemampuan.

Namun, ini baru permulaan. Semua komputasi ini membutuhkan energi dan pusat data. Robot akan mampu membangun dan memelihara infrastruktur yang diperlukan jauh lebih cepat daripada manusia dan pada akhirnya secara otonom. Mereka akan mampu membangun pusat data baru, memasang dan mengkonfigurasi perangkat keras, dan mengoptimalkan penggunaan energi. Ini menciptakan siklus yang baik di mana sistem AI tidak hanya meningkatkan diri mereka sendiri tetapi juga infrastruktur yang mendukungnya.

Seiring agen AI menjadi lebih mampu dan dapat melakukan penelitian otonom, mereka akan menciptakan dan menemukan algoritma yang lebih baik, material yang lebih baik, dan proses yang lebih baik untuk memecahkan masalah sulit seperti logistik rantai pasokan. AI dan robot akan memainkan peran yang semakin penting dalam membentuk masa depan teknologi dan masyarakat.

Menaklukkan Ruang Angkasa – Batas Terakhir

Namun, itu baru permulaan. Robot akan mampu menutupi bumi dengan apa yang disebut "computronium"—jaringan pusat data dan pabrik yang dapat memproduksi lebih banyak robot dan sistem AI. Setelah kita memiliki kapasitas komputasi dan manufaktur yang cukup di Bumi, langkah selanjutnya adalah ekspansi ke luar angkasa. Sumber daya Bumi terbatas, tetapi sumber daya tata surya sangat luas. Robot dapat dikirim untuk menambang asteroid, membangun habitat luar angkasa, dan membangun pembangkit listrik tenaga surya di orbit. Ekspansi ke luar angkasa ini akan menyediakan sumber daya yang diperlukan untuk mempertahankan dan mengembangkan ekonomi AI tanpa batas.

Kesimpulannya, konvergensi AI dan robotika siap untuk mengubah ekonomi dan masyarakat secara mendalam. Fiksi ilmiah akan menjadi sains semata karena AI dan robot dengan cepat (dalam beberapa tahun) memecahkan masalah ilmiah di bidang matematika, fisika, kimia, dan biologi. Pertanyaan besarnya adalah apakah umat manusia dapat menavigasi transisi ini dengan sukses.

12.5 RINGKASAN

Bab ini mengeksplorasi masa depan AI dan sistem multi-agen, meneliti kemampuan transformatif yang muncul dalam beberapa tahun ke depan. Kami membahas penalaran supermanusia dan pandangan strategis ke depan, di mana sistem AI akan melampaui kemampuan manusia tidak hanya dalam kecepatan tetapi juga dalam kualitas penalaran fundamental, melakukan rantai logika multi-langkah di ribuan langkah inferensi dan mensintesis wawasan di berbagai domain yang berbeda. Sistem multi-agen memperkuat kemampuan ini melalui penalaran kolaboratif, di mana agen mengeksplorasi berbagai skenario secara bersamaan dan terlibat dalam perencanaan yang saling bertentangan untuk merancang strategi yang kuat. Bab ini juga meneliti jendela konteks besar yang meluas dari ribuan hingga jutaan token, memungkinkan model untuk memproses seluruh basis kode, koleksi dokumen hukum, dan percakapan yang diperpanjang sambil melakukan pembelajaran pada waktu inferensi yang mengurangi ketergantungan pada pra-pelatihan.

Kami mengeksplorasi arsitektur untuk perencanaan jangka panjang melalui dekomposisi hierarkis dan pendekatan berbasis model menggunakan model dunia untuk mensimulasikan skenario masa depan. Diskusi meluas melampaui teks ke sistem AI multi-modal yang memproses visi, audio, video, dan domain khusus seperti pemodelan biologis untuk pelipatan protein dan penemuan obat. Kemampuan multi-modal ini memungkinkan agen AI untuk memahami dan berinteraksi dengan dunia fisik, yang mengarah pada AI yang terwujud dalam robotika. Bab ini meneliti bagaimana robot humanoid dengan kecerdasan fisik serbaguna akan muncul dalam 3–5 tahun ke depan, mampu melakukan tugas fisik apa pun melalui teknik seperti pembelajaran penguatan, pembelajaran imitasi, dan transfer simulasi ke dunia nyata. Koordinasi multi-robot akan memungkinkan armada untuk berbagi pengetahuan secara instan, mempercepat pembelajaran di seluruh sistem yang digunakan di gudang, lokasi konstruksi, dan rumah.

Implikasi ekonominya sangat besar. Kami meneliti ekonomi AI yang sedang berkembang di mana agen otonom menjadi aktor ekonomi melalui protokol seperti AP2, yang memungkinkan transaksi yang aman dan dapat diverifikasi antara agen AI dan pedagang menggunakan kredensial kriptografi. Ekonomi agen-ke-agen ini akan menciptakan pasar baru untuk layanan dan pertukaran nilai, dengan agen AI merekrut pekerja manusia untuk tugas-tugas yang membutuhkan persetujuan peraturan atau kepatuhan terhadap proses lama. Siklus peningkatan diri dipercepat saat robot secara otonom membangun pusat data, melakukan penelitian, dan mengoptimalkan infrastruktur, yang akhirnya meluas melampaui Bumi ke sumber daya berbasis ruang angkasa.

Secara keseluruhan, buku ini memberikan perjalanan komprehensif dari konsep dasar hingga penerapan sistem AI multi-agen dalam produksi. Kita mulai dengan memahami AI generatif dan arsitektur agen, kemudian membangun sistem yang semakin canggih: agen sederhana, kerangka kerja yang menggunakan alat, alat khusus, dan antarmuka obrolan.

Kami menjelajahi ekosistem melalui **Model Context Protocol** (MCP) untuk integrasi alat dan **Agent-to-Agent** (A2A) untuk komunikasi antar agen, mendemonstrasikan konsep-konsep ini melalui implementasi praktis seperti MAKDO untuk Kubernetes DevOps. Buku ini

membahas praktik-praktik rekayasa penting, termasuk pengujian, debugging, dan strategi penerapan, dan diakhiri dengan pandangan ke masa depan, menunjukkan bagaimana sistem multi-agen akan berkembang bersamaan dengan AI superhuman, robotika, dan ekonomi otonom. Anda sekarang memiliki pemahaman teoritis dan keterampilan praktis untuk merancang, mengimplementasikan, menguji, dan menerapkan sistem AI multi-agen siap produksi yang akan membentuk dekade inovasi teknologi berikutnya.

DAFTAR PUSTAKA

Ali, Z., & Grudén, K. (2025). Intelligent multi-agent framework for automated data analytics and machine learning tasks (AI agent).

Amini, S., Benajiba, Y., Bernardis, C., Cayet, P., Chafi, H., Fathan, A., ... & Xu, J. (2025). Open Agent Specification (Agent Spec): A Unified Representation for AI Agents. arXiv preprint arXiv:2510.04173.

Bar, K. (2025). The Agentic AI Handbook: Concepts, Design Patterns, and Future Directions.

Biswas, A., & Talukdar, W. (2025). Building Agentic AI Systems: Create intelligent, autonomous AI agents that can reason, plan, and adapt. Packt Publishing Ltd.

Blount, A., Gulli, A., Saboo, S., Zimmermann, M., & Vuskovic, V. (2025). Introduction to Agents. Google White Paper. November.

Bui, N. D. (2026). Building AI Coding Agents for the Terminal: Scaffolding, Harness, Context Engineering, and Lessons Learned. arXiv preprint arXiv:2603.05344.

Deng, S., Zhao, H., Wang, Z., Cheng, G., Chen, P., Qian, W., ... & Dustdar, S. (2025). Agentic services computing. arXiv preprint arXiv:2509.24380.

Dibia, V. (2025). Designing Multi-Agent Systems: Principles, Patterns and Implementation for AI Agents. Victor Dibia.

Du, C., Wang, C., Chao, Y., Xie, X., & Cui, Y. (2025). AI agent communication from internet architecture perspective: Challenges and opportunities. arXiv preprint arXiv:2509.02317.

Ehtesham, A., Singh, A., Gupta, G. K., & Kumar, S. (2025). A survey of agent interoperability protocols: Model context protocol (mcp), agent communication protocol (acp), agent-to-agent protocol (a2a), and agent network protocol (anp). arXiv preprint arXiv:2505.02279.

Ersoy, P., & Erşahin, M. (2025, October). Protocol-Driven Agentic Integration of LLMs: A Multi-tool Use Case. In International Joint Conference on Computational Intelligence (pp. 139-148). Cham: Springer Nature Switzerland.

Fang, J., Peng, Y., Zhang, X., Wang, Y., Yi, X., Zhang, G., ... & Meng, Z. (2025). A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. arXiv preprint arXiv:2508.07407.

Gabbita, B. S. T. (2025). Towards secure agentic workflows: a MAESTRO-based assessment framework for Model Context Protocol and Agent-to-Agent Protocol: a thesis in Computer Science (Doctoral dissertation, University of Massachusetts Dartmouth).

Gholizadeh HamlAbadi, K., Vahdati, M., Laamarti, F., & El Saddik, A. (2025). Agent-to-Agent (A2A) Protocol Integrated Digital Twin System with AgentIQ for Multimodal AI Fitness Coaching and Personalized Well-Being. In Proceedings of the 33rd ACM International Conference on Multimedia (pp. 12483-12491).

Ghosh, D. P. (2025). Agentic Ecosystemin Engineering Design: A Framework for Interoperable Legacy Tools and Emergent Collaboration via MCP/A2A Protocols.

Gullí, A. (2025). Agentic design patterns: a hands-on guide to building intelligent systems. Springer Nature.

Hasan, M. M., Li, H., Fallahzadeh, E., Rajbahadur, G. K., Adams, B., & Hassan, A. E. (2025). An empirical study of testing practices in open source AI agent frameworks and agentic applications. arXiv preprint arXiv:2509.19185.

Huang, K., & Hughes, C. (2025). Deploying Agentic AI in Enterprise Environments. In Securing AI Agents: Foundations, Frameworks, and Real-World Deployment (pp. 289-319). Cham: Springer Nature Switzerland.

Ioannides, G., Constantinou, C., Nija, V., Jain, A., & Allyson, E. (2024, October). MOD-X: A Modular Open Decentralized eXchange Framework Proposal for Heterogeneous Interoperable Artificial Intelligence Agents. In AI Agent for Information Retrieval Workshop (pp. 75-95). Cham: Springer Nature Switzerland.

Jeong, C. (2025). A Practical MCP× A2A Integration Framework for Interoperability in LLM-Based Autonomous Multi-Agent Systems. 지능정보연구, 31(3), 141-170.

Jeong, C. (2025). A study on the mcp x a2a framework for enhancing interoperability of llm-based autonomous agents. arXiv preprint arXiv:2506.01804.

Jiang, W., & Hu, F. (2025). Artificial Intelligence Agent-Enabled Predictive Maintenance: Conceptual Proposal and Basic Framework. Computers, 14(8), 329.

Khan, N. I., Habiba, M., & Khan, R. (2025). A Gossip-Enhanced Communication Substrate for Agentic AI: Toward Decentralized Coordination in Large-Scale Multi-Agent Systems. arXiv preprint arXiv:2512.03285.

Lanham, M. (2025). AI agents in action. Simon and Schuster.

Lee, Y., & Park, E. (2026). Toward Sustainable Agentic AI Systems: A Survey of Architectures and Methodologies. Sustainable Development.

Lei, Y., Xu, J., Liang, C. X., Bi, Z., Li, X., Zhang, D., ... & Yu, Z. (2025). A Comprehensive Survey on Model Context Protocol: Architecture, Tool Integration, and the Future of AI

Interoperability. Tool Integration, and the Future of AI Interoperability (December 23, 2025).

Li, Q., & Xie, Y. (2025). From glue-code to protocols: A critical analysis of a2a and mcp integration for scalable agent systems. arXiv preprint arXiv:2505.03864.

Lopez, S., Ramos, G., & Arranz, A. Perfecting AI Agent Frameworks through Unified Design Principles.

Louck, Y., Dvir, A., & Stulman, A. (2025). Security Analysis of Agentic AI Communication Protocols: A Comparative Evaluation. ACM Transactions on AI Security and Privacy.

Maes, S. H. (2026, March). Vibe-Coding and SDLC Constrained And Managed By An Application-Aware AI-Like Agentic Platform.

Murthya, S. L., Selvama, S. P., & Welßa, M. (2025). Multi-Agentic Workflows and Long-Term Memory Use Cases in AI-Builder.

Petrova, T., Bliznioukov, B., Puzikov, A., & State, R. (2025). From Semantic Web and MAS to Agentic AI: A Unified Narrative of the Web of Agents. arXiv preprint arXiv:2507.10644.

Ray, P. P. (2025). A review on agent-to-agent protocol: Concept, state-of-the-art, challenges and future directions. Authorea Preprints.

Rohitha, K. P. (2025). Agentic AI-Building Autonomous Intelligent Systems for Enterprise Applications. AGPH Books| AG Publishing House.

Roman, A., & Roman, J. (2026). Orchestral AI: A Framework for Agent Orchestration. arXiv preprint arXiv:2601.02577.

Spieser, J., Balapour, A., Meller, J., Patra, K. C., & Shamsaei, B. (2026). A Review of Multi-Agent AI Systems for Biological and Clinical Data Analysis. Methods and Protocols, 9(2), 33.

Styer, M., Patlolla, K., Mohan, M., & Diaz, S. Agent Tools & Interoperability with MCP. Kaggle.[Online]. Available: https://www. kaggle. com/whitepaper-agent-tools-andinteroperability-with-mcp.[Accessed: Nov. 13, 2025].

Tupe, V., & Thube, S. (2025, July). Demonstrating Multi-Agent Collaboration via Agent-to-Agent and Model Context Protocols: An IT Incident Response Case Study. In 2025 IEEE International Conference on Service-Oriented System Engineering (SOSE) (pp. 1-5). IEEE.

Venkiteela, P. (2025). The New Interoperability Paradigm Model Context Protocol (MCP), APIs, and the Future of Agentic AI. Comput. Fraud Sec, 8(1), 1259-1271.

Voudouris, C. (2025). Technology Foundations of Autonomic Business. In Autonomic Business Transformation: A Guide to AI Agents for Practitioners and Executives (pp. 31-43). Cham: Springer Nature Switzerland.

Wang, H., Gong, J., Zhang, H., Xu, J., & Wang, Z. (2025). Ai agentic programming: A survey of techniques, challenges, and opportunities. arXiv preprint arXiv:2508.11126.

Wang, Y., Xu, X., Chen, J., Bi, T., Gu, W., & Zheng, Z. (2025). An empirical study of agent developer practices in ai agent frameworks. arXiv preprint arXiv:2512.01939.

Wang, Z., Yu, G., & Lyu, M. R. (2026). AI-NativeBench: An Open-Source White-Box Agentic Benchmark Suite for AI-Native Systems. arXiv preprint arXiv:2601.09393.

Wu, S., Wang, Z., Song, X., Zhou, Z., Sun, L., & Shi, T. (2025). GoalfyMax: A Protocol-Driven Multi-Agent System for Intelligent Experience Entities. arXiv preprint arXiv:2507.09497.

Xu, H., Zlatanova, S., Li, X., Wachowicz, M., & Batty, M. (2025). Towards fully automated city operations: integrating agentic AI with urban digital twins. Available at SSRN 5596992.

Xu, W., & Dong, B. OptAgent: An Agentic AI framework for Intelligent Building Operations based on a Physics-Informed Simulation Environment.

Yang, Y., Chai, H., Song, Y., Qi, S., Wen, M., Li, N., ... & Zhang, W. (2025). A survey of ai agent protocols. arXiv preprint arXiv:2504.16736.

Yang, Y., Ma, M., Huang, Y., Chai, H., Gong, C., Geng, H., ... & Wang, J. (2025). Agentic web: Weaving the next web with ai agents. arXiv preprint arXiv:2507.21206.

Yu, J. S., & Yao, Y. (2026). Agents and Workflows. In Intelligent Language Services: Theory and Practice with Large Language Models (pp. 113-141). Singapore: Springer Nature Singapore.

Zhang, W., Zeng, L., Xiao, Y., Li, Y., Cui, C., Zhao, Y., ... & An, B. (2025). AgentOrchestra: Orchestrating Multi-Agent Intelligence with the Tool-Environment-Agent (TEA) Protocol. arXiv preprint arXiv:2506.12508.

Sistem AI Multi-Agen berbasis Python Menggunakan MCP (Model Context Protocol) dan A2A (Agent-to-Agent)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom.

BIODATA PENULIS

Dr. Joseph Teguh Santoso, M.Kom memiliki Jabatan Akademik Lektor Kepala dan praktisi industri yang berpengalaman. Saat ini menjabat sebagai Rektor Universitas Sains dan Teknologi Komputer (Universitas STEKOM), salah satu universitas terkemuka di Jawa Tengah, Indonesia. Dengan pengalaman lebih dari 13 tahun di dunia bisnis dan praktisi industri di China, beliau membawa perspektif global dan inovasi yang signifikan ke dalam dunia akademis.Sebagai seorang entrepreneur, penulis adalah pencipta TopLoker.com, sebuah platform inovatif yang merevolusi cara mencari dan menawarkan pekerjaan. TopLoker.com adalah portal lowongan bursa kerja terbesar di Indonesia, khusus untuk pendidikan SMA/SMK sederajat. TopLoker.com telah mendapatkan penghargaan sebagai juara 1 Startup4industry 2022 oleh Kementerian Perindustrian Republik Indonesia. Kontribusi Dr. Joseph dalam menyediakan akses pekerjaan yang luas bagi lulusan SMA/SMK telah membantu banyak individu menemukan peluang kerja yang sesuai dengan keahlian mereka. Selain itu, Dr. Joseph Teguh Santoso, M.Kom adalah pendiri dari dua organisasi yaitu (1) organisasi guru/pendidik PTIC (Perkumpulan Teacherpreneur Indonesia Cerdas) yang bertujuan untuk meningkatkan kualitas pendidikan dan kesejahteraan guru/pendidik dengan wawasan entrepreneurship, serta (2) organisasi industri PERKIVI (Perkumpulan Komunitas Industri dan Vokasi Indonesia) yang berfokus pada pengembangan link and match antara industri dan dunia pendidikan. Sebagai Rektor, Dr. Joseph Teguh Santoso, M.Kom memiliki kepemimpinan yang berorientasi pada hasil, dan berkomitmen untuk mendorong kemajuan Universitas Sains dan Teknologi Komputer (Universitas STEKOM). Saat ini Universitas STEKOM telah mengalami transformasi positif dalam peningkatan kualitas pendidikan, perluasan fasilitas, serta penguatan kemitraan Perguruan Tinggi Nasional dan Internasional. Beliau memprioritaskan pengembangan sumber daya manusia dan penelitian, serta memastikan bahwa universitas berada di garis depan dalam inovasi dan teknologi untuk mencapai tujuan akhir, yaitu lulusan yang mampu bekerja dan sukses setelah lulus. Dr. Joseph Teguh Santoso, M.Kom sering diundang sebagai pembicara di berbagai konferensi nasional maupun internasional dan telah menerima berbagai penghargaan atas dedikasinya dalam bidang pendidikan, industri, dan kewirausahaan.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7695-02-4 (PDF)
9 786347 695024