

O'REILLY®

Building Applications with AI Agents

Designing and Implementing Multi-Agent Systems

Early Release

RAW & UNEDITED



Michael Albada

Building Applications with AI Agents

Designing and Implementing Multi-Agent Systems

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Michael Albada

O'REILLY®

Building Applications with AI Agents

by Michael Albada

Copyright © 2026 Advance AI LLC. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Shira Evans
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- October 2025: First Edition

Revision History for the Early Release

- 2024-10-18: First Release
- 2024-11-05: Second Release
- 2024-12-13: Third Release
- 2025-01-30: Fourth Release
- 2025-04-11: Fifth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098176501> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Applications with AI Agents*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and

instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17650-1

Brief Table of Contents (*Not Yet Final*)

Chapter 1: Introduction to Agents (available)

Chapter 2: Overview of Designing Agent Systems (available)

Chapter 3: User Experience Design for Agentic Systems
(available)

Chapter 4: Skills (available)

Chapter 5: Orchestration (available)

Chapter 6: Knowledge and Memory (available)

Chapter 7: Learning in Agentic Systems (available)

Chapter 8: From One Agent to Many (available)

Chapter 9: Measurement and Validation (unavailable)

Chapter 10: Monitoring in Production (unavailable)

Chapter 11: Protecting Agent Systems (unavailable)

Chapter 12: Your New Teammate (unavailable)

Chapter 13: Humans and Agent (unavailable)

Chapter 1. Introduction to Agents

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at

sevans@oreilly.com.

In recent years, the field of artificial intelligence (AI) has seen remarkable advancements, particularly in the development and deployment of autonomous agents. These agents promise to revolutionize various industries by automating complex tasks, making intelligent decisions, and interacting seamlessly with both humans and other systems. This chapter provides an overview of autonomous agents, their unique capabilities, and

the context in which they operate, setting the stage for a deeper exploration of their design and implementation.

What are Agents?

Autonomous agents represent a significant leap forward in AI, offering the potential to enhance productivity, improve decision-making, and tackle problems that were previously beyond the reach of traditional software. These agents combine the power of large language models with sophisticated planning, execution skills, and the ability to adapt to dynamic environments.

Agents are software entities that perform tasks autonomously on behalf of users or other programs. Unlike traditional programs that require explicit instructions for each action, agents can make decisions and take actions independently based on their understanding of the environment and their objectives. This autonomy allows agents to handle complex tasks that involve multiple steps, varying conditions, and interaction with other systems or humans.

Similarities and Differences from Traditional Machine Learning

Traditional machine learning (ML) systems are designed to process specific tasks, such as image recognition, language translation, or predictive analytics. These systems are trained on large datasets to identify patterns and make predictions. While ML models can be powerful, they typically operate within predefined boundaries and lack the flexibility to adapt to new, unforeseen scenarios without additional retraining.

In contrast, agents extend the capabilities of traditional ML by incorporating decision-making and planning abilities. They use ML models, particularly large language models, as a foundation but go beyond static predictions. Agents can interpret the context, plan sequences of actions, execute skills, and respond to changes in real time. This makes them suitable for more dynamic and complex environments where flexibility and adaptability are crucial.

Recent Advancements

The development of large language models, such as OpenAI's GPT-4, has been a major driving force behind the recent surge in autonomous agent capabilities. These models provide a deep understanding of language, enabling agents to interpret and generate human-like text. Combined with advancements in reinforcement learning, planning algorithms, and real-time data processing, agents are now able to perform sophisticated tasks with minimal human intervention.

Recent advancements also include improvements in the robustness and scalability of agents. Enhanced training techniques, more efficient algorithms, and the integration of diverse data sources have contributed to the development of agents that are not only smarter but also more reliable and efficient.

From Synchronous to Asynchronous

Traditional software systems often operate in a synchronous manner, where tasks are performed in a linear sequence, waiting for each step to complete before proceeding to the next. While this approach is straightforward, it can be inefficient, especially for tasks that involve waiting for external inputs or processing large amounts of data.

Autonomous agents, however, are designed to operate asynchronously. This means they can perform multiple tasks concurrently, react to new information as it becomes available, and prioritize actions based on changing conditions. Asynchronous operation enables agents to handle more complex scenarios and improve their overall efficiency by minimizing idle times and making better use of available resources. It also enables them to begin handling tasks as they occur. Imagine that emails you need to respond to already have drafts ready for you by the time you read them, or accountants that receive an invoice already are presented with a pre-drafted payment template, or software engineers that receive a ticket already have a first draft of code that they can review and modify to address the issue. This is gradually changing each of us from workers to managers, as we move to reviewing drafts of work generated for us, and focusing our efforts on the most important and challenging aspects of the task.

When Are Agents Useful?

Agents are particularly useful in scenarios that require complex decision-making, real-time responsiveness, and the ability to operate in dynamic environments. While traditional machine learning models excel in specific, well-defined tasks such as risk

prediction or churn prediction, agents shine in areas where summarizing large amounts of information, operating over unstructured text or information, and handling repetitive processes are critical.

For instance, traditional ML models are excellent at predicting customer churn based on historical data or assessing financial risks by analyzing structured datasets. These models are purpose-built and highly optimized for these tasks, offering precise and reliable predictions. However, their utility diminishes in scenarios where the problem is less structured or requires continuous adaptation and reasoning over varied data types.

Agents, leveraging the power of large language models and advanced AI techniques, are particularly effective in scenarios such as:

Summarizing Large Amounts of Information

Agents can process vast volumes of text, extracting key insights and summarizing information in a coherent and concise manner. This is invaluable in fields like research, legal analysis, and content curation, where sifting through large documents is a common task.

Operating Over Unstructured Text or Information

Agents can interpret and generate human-like text, making them suitable for tasks involving unstructured data such as emails, reports, and social media content. They can identify relevant information, answer queries, and provide context-aware responses.

Handling Repetitive Processes

In industries like customer service, agents can automate repetitive tasks such as answering common inquiries, processing routine transactions, and managing simple workflows. This automation frees up human workers to focus on more complex and creative tasks.

Reasoning Over Text or Images

Agents can perform tasks that require reasoning and interpretation of textual or visual information. For example, they can analyze customer feedback, provide diagnostic support in healthcare by interpreting symptoms described in text, or even generate creative content.

These capabilities open up a class of problems that were almost impossible to address or automate with traditional ML.

However, agents also have their limitations. Complex multi-step reasoning, where a task involves intricate dependencies and long chains of logic, remains a challenge. While agents are proficient at processing and generating information, ensuring accurate and consistent outcomes across complex reasoning tasks often requires further advancements and integration with specialized systems.

In summary, agents are most useful in environments where flexibility, adaptability, and the ability to handle unstructured or large-scale information are paramount. They complement traditional ML models by extending the range of problems that AI can tackle, but they also require careful consideration of their limitations and the specific needs of each application. As we explore the capabilities of autonomous agents in this book, we will highlight how frameworks like Autogen enhance their utility and address some of these challenges, providing practical solutions for real-world scenarios.

Managing Expectations

While the promise of autonomous agents is vast, it is essential to manage expectations regarding their capabilities. Agents are powerful tools but are not infallible. They rely on the quality of

their underlying models and the data they are trained on. Moreover, the complexity of real-world environments means that agents can encounter situations that challenge their decision-making abilities.

It is important for stakeholders to understand that agents may require continuous monitoring, updates, and human oversight to ensure they operate effectively and ethically. Setting realistic expectations helps in leveraging the strengths of agents while being prepared to address their limitations. Now that we've considered when agents can be useful, let's look at some example scenarios.

Use Cases for Agents

The versatility of autonomous agents opens up a myriad of applications across different industries. By leveraging large language models and sophisticated planning and execution frameworks, these agents can perform a wide array of tasks, providing significant value in various contexts. This section explores some prominent use cases for agents, illustrating their potential impact and benefits.

Customer Support Agent

Customer support is one of the most prevalent applications for autonomous agents. These agents can handle a vast number of customer inquiries efficiently and effectively, providing 24/7 support without the need for human intervention. Key functionalities of customer support agents include:

Automated Responses

Agents can answer frequently asked questions, provide information about products and services, and guide customers through troubleshooting processes.

Personalized Assistance

By analyzing customer data and interaction history, agents can offer tailored recommendations and solutions, enhancing the customer experience.

Escalation Management

For complex issues, agents can seamlessly escalate the query to human support representatives, ensuring that customers receive the necessary attention without long wait times.

Sentiment Analysis

Agents can monitor customer sentiment during interactions and adjust their responses accordingly to maintain a positive customer experience.

These capabilities not only improve customer satisfaction but also reduce operational costs and free up human agents to focus on more complex and value-added tasks.

Personal Assistant Agent

Personal assistant agents are designed to help individuals manage their daily tasks and routines more efficiently. These agents leverage natural language processing to interact with users, understand their preferences, and provide timely and relevant assistance. Some key functions include:

Scheduling and Reminders

Personal assistant agents can manage calendars, schedule meetings, set reminders, and send notifications about important events.

Information Retrieval

Agents can quickly access and present information on various topics, such as news updates, weather forecasts, and travel information.

Task Automation

Agents can automate repetitive tasks, such as sending emails, managing to-do lists, and ordering supplies, thus saving users time and effort.

Integration with Smart Devices

Personal assistants can control smart home devices, such as lights, thermostats, and security systems, providing a seamless and integrated user experience.

By simplifying routine activities and offering proactive assistance, personal assistant agents enhance productivity and convenience for users.

Legal Agent

In the legal domain, agents can assist lawyers and legal professionals by automating routine tasks, providing research support, and enhancing decision-making processes. Legal agents offer several key benefits:

Document Analysis

Agents can review and analyze legal documents, contracts, and case files, identifying relevant information

and potential issues.

Legal Research

Agents can conduct comprehensive legal research, gathering information from case law, statutes, and legal precedents to support legal arguments.

Compliance Monitoring

Agents can monitor changes in regulations and ensure that legal practices comply with the latest laws and standards.

Case Management

Legal agents can assist in managing case workflows, tracking deadlines, and organizing documentation to streamline legal processes.

These functionalities help legal professionals increase their efficiency, reduce the risk of errors, and focus on higher-level strategic tasks.

Advertising Agent

In the advertising industry, agents can optimize campaign management, targeting, and performance analysis, driving

better results and higher return on investment. Advertising agents can perform various critical tasks, including:

Audience Targeting

Agents can analyze demographic and behavioral data to identify and target specific audience segments, ensuring that ads reach the most relevant viewers.

Content Creation

Using natural language generation, agents can create compelling ad copy, social media posts, and other marketing materials tailored to different platforms and audiences.

Performance Analysis

Agents can monitor and analyze the performance of advertising campaigns in real time, providing insights and recommendations for optimization.

Budget Management:

Agents can allocate and adjust advertising budgets across different channels based on performance metrics and strategic goals.

By leveraging these capabilities, advertising agents enhance the effectiveness of marketing efforts, maximize engagement, and improve overall campaign outcomes.

In conclusion, autonomous agents offer significant potential across various use cases, from customer support and personal assistance to legal services and advertising. By integrating these agents into their operations, organizations can achieve greater efficiency, improve service quality, and unlock new opportunities for innovation and growth. As we continue to explore the capabilities and applications of autonomous agents in this book, it becomes evident that their impact will be profound and far-reaching across multiple industries. Now that we've looked at some example agents, in the next section, we'll discuss some of the key considerations when designing our agentic systems.

Building with Change in Mind

The rapid pace of technological advancement and the dynamic nature of real-world environments necessitate designing autonomous agents with adaptability and flexibility at their core. Building agents that can not only perform their current tasks effectively but also evolve in response to new challenges

and opportunities is crucial for long-term success. This section explores key considerations for creating adaptive agents, emphasizing the importance of scalability, modularity, continuous learning, and robust architecture.

Scalability

Scalability is essential for ensuring that agents can handle increasing workloads and expanding tasks as their deployment grows. To build scalable agents, developers should focus on:

Distributed Architecture

Implementing a distributed system allows agents to leverage multiple processing nodes, ensuring that they can handle large volumes of data and complex computations efficiently. This approach also provides redundancy, enhancing the system's reliability.

Cloud Integration

Utilizing cloud services offers virtually unlimited resources for storage and computation, enabling agents to scale seamlessly. Cloud platforms provide the flexibility to dynamically allocate resources based on demand, ensuring optimal performance.

Efficient Algorithms

Designing algorithms that can efficiently process data and perform tasks is critical for scalability. Optimization techniques, such as parallel processing and load balancing, help distribute the workload evenly across the system.

For organizations to unlock the potential for agentic systems, building for scale is essential, which requires a robust and cost-efficient architecture.

Modularity

Modularity involves designing agents with interchangeable and independent components, allowing for easy updates and modifications. This design philosophy enhances the agent's ability to adapt to new requirements and integrate with different systems. Key strategies include:

Component-Based Design

Breaking down the agent's functionality into discrete, self-contained modules allows developers to update or replace individual components without affecting the entire

system. This approach simplifies maintenance and enhances flexibility.

Clear Interfaces

Establishing well-defined interfaces between modules ensures smooth communication and integration. By adhering to standardized protocols and data formats, agents can easily interact with other systems and components.

Plug-and-Play Capabilities

Designing modules that can be added or removed with minimal configuration enables rapid adaptation to changing needs. This capability allows agents to incorporate new skills or functionalities as they become available.

Modularity enables agents to adapt easily to new requirements by using interchangeable, independent components. Key strategies include component-based design, clear interfaces, and plug-and-play capabilities, which together support flexibility, seamless integration, and easy updates.

Continuous Learning

Continuous learning is vital for agents to remain effective in dynamic environments. By constantly acquiring new knowledge and refining their skills, agents can improve their performance and adapt to evolving tasks. Strategies for fostering continuous learning include:

Reinforcement Learning

Implementing reinforcement learning algorithms allows agents to learn from their experiences, adapting their behavior based on feedback and rewards. This approach enables agents to optimize their actions over time and improve their decision-making processes.

Incremental Updates

Regularly updating the agent's knowledge base and models with new data ensures that they remain current and relevant. Incremental learning techniques enable agents to incorporate new information without retraining from scratch.

User Feedback Integration

Leveraging feedback from users helps agents refine their interactions and responses. By analyzing user input and

adapting accordingly, agents can enhance their effectiveness and user satisfaction.

One of the key limitations of previous generations of automation is that manual updates have generally been required, making them brittle and less useful over time. This new generation of autonomous agents are capable of continuous learning from implicit and explicit feedback, enabling them to improve performance and adjust to evolving tasks.

Resilience

A resilient architecture ensures that agents can operate reliably under various conditions and handle unexpected challenges gracefully. Key elements of a resilient architecture include:

Error Handling

Implementing comprehensive error handling mechanisms allows agents to detect and recover from failures, ensuring continuous operation. This includes anticipating potential issues and designing fallback strategies.

Security Measures

Ensuring the security of the agent and its data is paramount. Implementing encryption, access controls, and regular security audits protects against unauthorized access and data breaches.

Redundancy

Building redundancy into the system provides backup resources that can take over in case of component failures. This enhances the overall reliability and availability of the agent.

A resilient architecture enables agents to function reliably across diverse conditions and manage unexpected challenges effectively. Essential components include error handling, security measures, and redundancy, which together enhance resilience, security, and system reliability.

Future-Proofing

Future-proofing involves designing agents that can easily adapt to emerging technologies and trends. This requires a forward-thinking approach, considering potential developments in AI, data processing, and user expectations. Strategies for future-proofing include:

Open Standards

Adopting open standards and protocols ensures that agents can integrate with future systems and technologies. This approach minimizes the risk of obsolescence and enhances compatibility.

Scalable Infrastructure

Investing in scalable infrastructure from the outset allows agents to accommodate future growth and technological advancements without significant overhauls.

Continuous Innovation

Encouraging a culture of continuous innovation ensures that agents remain at the cutting edge of technology. Regularly exploring new tools, techniques, and methodologies helps maintain the agent's relevance and effectiveness.

Building autonomous agents with change in mind is essential for their long-term success and adaptability. By focusing on scalability, modularity, continuous learning, robust architecture, and future-proofing, developers can create agents that are not only effective in their current tasks but also capable of evolving with the ever-changing technological landscape.

This approach ensures that agents remain valuable assets, capable of meeting new challenges and seizing emerging opportunities. In the next section, we'll discuss when and why we would use multi-agent systems, and discuss the relationship between foundation models and autonomous agents.

Towards Multi-Agent Systems

Multi-agent systems involve multiple autonomous agents working together to achieve common goals or perform distributed tasks. While these systems are more complex to develop, configure, and maintain, they open up additional capabilities and can often improve performance on specific tasks. These systems are designed to leverage the collective intelligence and capabilities of individual agents, allowing for more complex and scalable solutions. Multi-agent systems are particularly useful in environments where tasks are distributed, dynamic, or require collective problem-solving, such as code-generation, cybersecurity monitoring, supply chain management, customer support automation, sales automation, or healthcare coordination.

Foundation Models and Autonomous Agents

Recent advancements in large language models, such as GPT-4, Anthropic's Claude, and Meta's Llama have significantly impacted the design of autonomous agents. These models provide a deep understanding of language, enabling agents to process natural language input, generate coherent responses, and perform complex linguistic tasks. Incorporating large language models into autonomous agents offers several advantages:

Natural Language Understanding

Agents can understand and interpret user queries, commands, and conversations, enabling more natural and intuitive interactions.

Context-Aware Responses

By maintaining context over longer interactions, agents can provide more relevant and accurate responses.

Content Generation

Agents can generate text, code, and other content, enhancing their ability to assist with creative and analytical tasks.

The integration of large language models with planning and execution frameworks allows for the development of highly

sophisticated agents capable of performing a wide range of tasks autonomously. By leveraging monitoring and feedback systems, developers can ensure that autonomous agents remain reliable, efficient, and aligned with their intended goals.

Conclusion

Autonomous agents represent a transformative development in AI, capable of performing complex, dynamic tasks with a high degree of autonomy. This chapter has outlined the foundational concepts of agents, highlighted their advancements over traditional machine learning systems, and discussed their practical applications and limitations. As we delve deeper into the design and implementation of these systems, it becomes clear that the thoughtful integration of agents into various domains holds the potential to drive significant innovation and efficiency.

While the various approaches to designing autonomous agents discussed in this chapter have demonstrated significant capabilities and potential, they also highlight the complexity and challenges involved in creating effective and adaptable systems. Each method, from rule-based systems to advanced cognitive architectures, offers unique strengths but also comes

with inherent limitations. In this book, I aim to bridge these gaps.

There is currently a wide variety of frameworks for developing agents, skills, memory, planning, orchestration, learning, and multi-agent coordination. While each of these frameworks has its own pros and cons, I choose to focus on the fundamentals. The code examples in this book focus on LangGraph, a leading framework. By focusing on LangGraph, we will explore how this innovative framework simplifies the design and implementation of autonomous agents, enabling them to better meet the demands of dynamic and complex environments. Through detailed explanations, practical examples, and real-world applications, I will demonstrate why LangGraph stands out as a superior approach for building the next generation of intelligent agents. In the next chapter, we'll provide a bird's-eye view of the key components in designing an effective agentic system.

Chapter 2. Overview of Designing Agent Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Creating effective agent-based systems starts with selecting the right scenario and defining tasks that align precisely with real-world needs. Scenario selection serves as the cornerstone of agent design, ensuring that each agent is deployed in an environment where it can provide maximum impact. Choosing the right scenario involves understanding the problem context,

setting clear objectives, and determining specific tasks suited to the agent's strengths. Well-defined tasks give agents a purposeful roadmap, helping them perform efficiently and successfully within the system's broader goals.

In this chapter, we explore the principles and practices that guide scenario selection and task definition. We begin with scoping: understanding the operational environment, identifying stakeholders, and recognizing the challenges agents are expected to solve. Next, we outline how to set objectives that are realistic, precise, and timely, providing agents with actionable goals. Finally, we examine constraints—technical, regulatory, and operational—that shape the agent's design and deployment. By approaching these elements carefully, developers can avoid common pitfalls in task definition and harness agent potential to deliver impactful, real-world solutions.

Scenario Selection and Task Definition

Scenario selection is the cornerstone of designing successful agent-based systems. It requires precise scoping of tasks, setting clear objectives, and identifying the right problems for agents

to solve. An agent is only as effective as the problem it's designed to tackle, and the way in which this problem is framed can dramatically influence the system's overall success. This section explores the intricacies of defining tasks for agents, outlines common pitfalls in problem definition, and highlights exemplary scenarios that illustrate best practices.

Defining the Problem: Scoping Tasks for Agents

The foundation of any agent-based system is rooted in a well-scoped problem. Scoping the problem involves thoroughly understanding the domain, identifying clear objectives, and accounting for constraints. This process enables the agent to operate with a well-defined purpose, focusing its capabilities on delivering measurable and valuable outcomes.

Understanding the Problem Context

Scoping begins with a comprehensive analysis of the problem context—the environment in which the agent will operate, the stakeholders involved, and the specific challenges that the agent is expected to address. This involves identifying the real-world scenario where the agent will be deployed and the interactions

it will have with other systems or users. Some of these factors to consider are:

Environmental Factors

What data, functions, and resources will the agent have access to? What regulatory environment does it operate under? What actions can reasonably be placed in scope of the agent, and what falls outside of the guardrails?

Stakeholders and User Interactions

Who will interact with the agent? What are the expectations of these users? Clearly understanding the stakeholders' goals ensures that the agent's objectives align with real-world needs.

Ethical and Social Considerations

How will the agent's actions impact users, employees, society at large? Assessing ethical implications, such as privacy, fairness, and potential biases, ensures that the agent operates responsibly and respects user rights. By accounting for these considerations in the scope, developers can better align the agent's behavior with societal expectations and ethical standards, fostering trust and minimizing unintended consequences.

Effective scoping sets the foundation for effective agent deployment by thoroughly analyzing the problem context, including the operating environment, key stakeholders, and targeted challenges. A well-defined scope considers environmental factors such as data access, regulatory constraints, and the agent's actionable boundaries, ensuring clear guardrails and alignment with real-world demands.

Defining Objectives

After you understand the problem context, the next step is to articulate clear objectives. These objectives are the roadmap that guides the agent's behavior and ensures its tasks are aligned with the system's goals. Well-defined objectives provide direction, focus, and measurable outcomes, shaping the agent's performance from development through deployment. Each objective must strike a balance between ambition and feasibility, ensuring that the agent has a clear and achievable path to success, and include the following attributes:

Precision

Objectives should be sharp and unambiguous, outlining exactly what the agent is expected to achieve. Vague or overly broad goals can lead to misalignment in development and inconsistent performance in

production. A precise objective clarifies the agent's purpose, cutting through potential distractions or deviations.

Realistic

Ambition is important, but objectives need to be realistic given the resources, technology, and data available.

Unrealistic goals can lead to wasted time and frustration, both for the developers and the users. Objectives should challenge the agent while staying within the realm of what's possible.

Timely

Objectives must be anchored in time to maintain momentum and accountability. A clear timeline not only creates urgency but also provides checkpoints for evaluation and course correction. Deadlines help ensure the project moves forward consistently, preventing the agent from languishing in perpetual development.

After defining the problem context, setting clear, well-articulated objectives is crucial to guide the agent's development and ensure alignment with system goals. These objectives should be precise, realistic, and timely, providing a

focused roadmap that balances ambition with feasibility and promotes consistent progress.

Identifying Constraints

Constraints are the technical, operational, or regulatory factors that impact the agent's ability to perform its tasks. Recognizing these limitations upfront helps ensure that the agent's design is both feasible and practical.

Technical Constraints

These include hardware limitations, computational resources, and network connectivity. For instance, an agent running on a mobile device will have different constraints compared to one running on a server farm.

Regulatory Constraints

In certain industries, agents must comply with regulatory standards (e.g., data privacy in healthcare or financial services). These constraints affect how the agent collects, processes, and stores data.

Operational Constraints

These include the agent's expected operational environment, such as physical constraints in robotics or

interaction limitations in a customer service chatbot. Understanding how and where the agent will be used is crucial for proper design.

By thoroughly defining the problem, understanding the context, and identifying constraints, developers can ensure that the agent is focused on delivering tangible outcomes, making efficient use of resources.

Avoiding Common Pitfalls in Task Definition

Even with a clear understanding of the problem, it's easy to fall into several common traps when defining tasks for agents.

Tasks that are too narrow, too broad, or too vague can all lead to inefficiencies and suboptimal performance. Properly scoping tasks is a balancing act, ensuring that the agent can perform its duties while fully leveraging its potential.

Choosing Too Narrow a Task

Focusing too narrowly on a single, specific problem can limit the agent's overall usefulness and result in underutilization of its capabilities. Tasks that are overly focused on small, isolated

functions might not allow the agent to demonstrate its broader potential or add significant value to the larger system.

For instance, an agent designed solely to check for spelling errors in text could miss opportunities to improve overall writing quality through grammar checking, style suggestions, or context-aware corrections. The agent's potential for text processing is much broader, and confining it to a single narrow task may result in a solution that only addresses a fraction of the problem.

The risk of over-narrowing a task is twofold:

Underutilization

The agent's capabilities are limited to a small subset of actions, reducing its impact and return on investment.

Lack of Flexibility

An overly specific task might make it difficult to adapt the agent to future needs or integrate it into a broader system.

Balancing specificity and generality ensures that the agent can provide comprehensive solutions, thus enhancing its value proposition.

Choosing Too Broad a Task

Conversely, defining a task that is too broad can overwhelm the agent, resulting in incomplete or inefficient solutions. Broad tasks often encompass multiple complex components, each requiring specialized knowledge, making it difficult for the agent to handle all aspects effectively.

Consider an agent tasked with managing all aspects of a smart city. This might include responsibilities ranging from traffic control and waste management to public safety and energy distribution. Such a task is not only too broad for a single agent but also poses significant challenges in designing the agent's decision-making capabilities. The sheer complexity of managing multiple interrelated systems can lead to performance degradation, delays, and increased resource consumption.

When a task is too broad you run into two issues:

1. *Complexity Increases*: Managing multiple domains or systems can make the agent's architecture unnecessarily complex and difficult to maintain.
2. *Lack of Focus*: A broad task might dilute the agent's attention, leading to lower performance in each of the individual sub-tasks.

To mitigate these risks, broad tasks should be broken down into more manageable components, with specialized agents or sub-agents tackling each aspect. This modular approach allows for higher efficiency and better performance.

Choosing Too Vague a Task

Vague tasks can lead to ambiguity in an agent's objectives, making it difficult to design targeted solutions. When the task is not well-defined, it becomes challenging to determine what success looks like, how to measure the agent's performance, and what actions the agent should prioritize.

For example, if an agent is tasked with "improving customer satisfaction," but the goals are not clearly defined (e.g., what specific actions will improve satisfaction, how success will be measured), the agent may struggle to deliver meaningful outcomes. Without clear metrics, the agent's actions might be inconsistent or even counterproductive.

The dangers of vague tasks include:

Scope Creep

The agent's responsibilities may gradually expand beyond the original intent, often without the necessary resources

or capabilities.

Inconsistent Performance

Without clear direction, the agent may lack focus, leading to unpredictable or unsatisfactory results.

To avoid this, tasks should be framed with precise objectives and measurable outcomes. Clear, actionable goals help keep the agent focused on delivering specific, valuable results.

Great Example Scenarios

Great example scenarios showcase the effectiveness of well-scoped agent systems in real-world applications. These scenarios illustrate how agents, when given well-defined and appropriately scoped tasks, can deliver impactful solutions across various industries:

Customer Support Chatbots

Customer support chatbots are a prime example of agents that operate within clearly defined tasks. These agents handle specific types of customer inquiries, such as account information, order status, and basic troubleshooting. By focusing on these well-defined tasks, chatbots provide efficient and consistent customer

service, reducing the workload on human agents and improving response times.

Personal Assistants

Virtual personal assistants, like Siri, Alexa, and Google Assistant, are designed to manage a variety of user requests, ranging from setting reminders and sending messages to providing weather updates and controlling smart home devices. These agents handle broad tasks but within a well-defined context, ensuring they perform effectively within their limitations. Each request is scoped clearly enough for the assistant to understand and act upon in real-time.

Healthcare Monitoring Systems

In healthcare, monitoring agents track patient vitals, manage medication schedules, and provide alerts for abnormal conditions. These tasks are clearly defined and focused on improving patient outcomes, leveraging real-time data and advanced analytics. By scoping tasks around well-defined patient care goals, healthcare agents ensure that they provide timely and critical information to healthcare providers.

These scenarios demonstrate the importance of choosing well-defined and appropriately scoped tasks for agent-based systems. By focusing on clear, manageable problems, agents can deliver significant value and achieve their intended outcomes effectively.

By thoroughly scoping problems, avoiding tasks that are too narrow, broad, or vague, and learning from successful example scenarios, developers can create agents that are well-equipped to address real-world challenges. Properly scoped tasks not only ensure the agent's effectiveness but also maximize the system's value and efficiency. Now that we've discussed how to approach the task of designing an agentic system, let's look at the most important parts of the agentic system.

Core Components of Agent Systems

Designing an effective agent-based system requires a deep understanding of the core components that enable agents to perform their tasks successfully. Each component plays a critical role in shaping the agent's capabilities, efficiency, and adaptability. From selecting the right models to equipping the agent with skills, memory, and planning capabilities, these elements must work together to ensure that the agent can

operate in dynamic and complex environments. This section delves into the key components—model selection, skills, memory, and planning—and explores how they interact to form a cohesive agent system.

Model Selection

At the heart of every agent-based system lies the *model* that drives the agent's decision-making processes. Model selection is a foundational step, as the model determines how the agent interprets data, makes decisions, and executes tasks. The choice of model depends on the complexity of the tasks the agent is expected to handle, the environment in which it will operate, and the specific requirements of the application.

Model Selection: Choosing the Best Foundation Model for the Job

The foundation of any agent-based system is the foundation model that drives its decision-making, interaction, and learning capabilities. Choosing the right model is a critical decision, as it will influence the system's performance, scalability, cost, and flexibility. Depending on the specific needs of the agent, the choice could range from large, general-purpose models to smaller, task-specific ones, or even custom-trained models

designed for niche applications. This section explores the various factors to consider when selecting the best model, including size, modality, openness, and customizability.

When it comes to language models, size often correlates with capability—but bigger isn't always better. Large models, such as GPT-4 or LLaMA, are highly versatile and capable of handling a wide range of tasks, from generating complex text to understanding nuanced queries. However, they also require significant computational resources and may be overkill for simpler tasks.

Large models are ideal for complex, multi-task agents that require deep language understanding or need to handle unstructured, diverse inputs. These models excel at managing ambiguity, generating creative outputs, and offering broad generalization across domains. However, they come with high computational demands, are expensive to run, and often involve higher latency, typically requiring cloud-based infrastructure to operate efficiently. They are best suited for applications like personal assistants, content generation, complex customer interactions, and creative problem-solving.

In contrast, smaller models are often more efficient for focused, well-defined tasks. Models like GPT-2 or BERT variants can

perform specific functions with lower resource requirements, deliver faster response times, and can often run on local devices. These models are ideal for task-specific automation, such as customer support or simple question answering, as well as embedded systems and edge computing where resource constraints are a priority.

Agents increasingly need to process a variety of data types beyond text, and multi-modal models are particularly advantageous in environments where agents interact with rich, diverse data formats. These models enable agents to interpret images, generate text from visual input, and analyze speech alongside text-based information, expanding their versatility in complex applications. Multi-modal models, such as OpenAI's DALL-E or Google's Flamingo, are well-suited for tasks that require interpreting multiple input types, making them ideal for AI-powered assistants in healthcare or autonomous systems that process both textual and visual data. They excel in autonomous systems, interactive assistants, and complex decision-making tasks that span multiple data formats.

In contrast, text-only models are typically more efficient and straightforward to implement for agents that work with strictly textual data. These models, focusing solely on generating or analyzing text, offer simplicity and effectiveness for a wide

range of applications. Text-only models are best used in pure conversational agents, documentation assistants, and customer service bots where interactions are heavily text-based.

The choice between *open-source* and *closed-source* models depends largely on the need for flexibility, transparency, and control over the model's behavior, versus the convenience and support provided by proprietary solutions.

Open-source models, such as GPT-Neo, BLOOM, or LLaMA, offer full transparency, enabling developers to inspect, modify, and fine-tune them according to specific needs. This level of control is particularly valuable for teams requiring extensive customization or those with strict privacy, ethical, or domain-specific standards. Open-source models also present cost advantages by eliminating licensing fees, though they may demand more infrastructure and maintenance for effective hosting. These models are ideal for custom applications that need domain-specific tuning, operate in privacy-sensitive environments, or prioritize transparency and control over the model's data and behavior.

On the other hand, closed-source, proprietary models like GPT-4, Google's PaLM, or Cohere's API provide high performance with less infrastructure and management overhead. They are

typically easier to deploy and include built-in optimizations, regular updates, and support, making them highly convenient. However, closed-source models come with less flexibility, as their internal workings are opaque and customization options are often limited. These models are best suited for rapid development, applications that value ease of integration, or teams without the resources to handle infrastructure management.

For many tasks, *pre-trained models*—which have been trained on massive, general-purpose datasets—are sufficient to deliver excellent performance out of the box. Pretrained, general-purpose models are ideal when the agent’s tasks fall within the scope of everyday language understanding or common knowledge domains. These models have been trained on extensive, diverse datasets and can generalize well across a variety of topics. Pre-trained models allow for fast deployment and can be fine-tuned for specific tasks with minimal additional training. This is best for applications where domain specificity is not critical, or where general understanding and language generation suffice (e.g., chatbots, content summarizers, writing assistants).

However, in specialized domains or when high accuracy is required, *custom-trained models* may be the better option. In

legal, medical, or technical domains—custom-trained models provide a significant advantage. By training a model on a domain-specific dataset, developers can ensure the agent has deep knowledge in that area and can deliver highly accurate and context-aware results.

When selecting a model, cost and performance are critical factors, especially when operating at scale. Large models or custom-trained solutions, while powerful, often come with high computational costs and require specialized infrastructure.

Large models, such as GPT-4 or multi-modal systems, require substantial computational power and are often best supported by cloud-based GPUs or TPUs to operate efficiently. This reliance on high-performance infrastructure can increase both initial development and ongoing operational costs, making them potentially impractical when budgets are limited. For less complex tasks, smaller or more efficient models can be more cost-effective. Additionally, response time, or latency, is crucial for real-time applications like customer service or automated trading, where rapid decision-making is essential. Larger models generally introduce higher latency, so it's important to balance their performance capabilities with the need for swift responsiveness in these scenarios.

In some cases, a *hybrid approach* that combines different types of models can provide the best solution. A large pre-trained model might handle general language understanding, while a smaller custom-trained model focuses on domain-specific tasks. Similarly, an agent might use a large, powerful model for complex interactions but switch to a smaller model for routine queries to optimize costs and latency. A customer service bot could rely on a large multi-modal model for visual troubleshooting or complex product inquiries but default to a smaller model for handling routine text queries like order tracking or account updates.

Choosing the right language model depends on a range of factors, including task complexity, data modalities, control and customization needs, and infrastructure limitations. By evaluating the trade-offs between large and small models, multi-modal capabilities, open-source flexibility versus proprietary ease, and pre-trained convenience versus custom domain knowledge, developers can tailor agent-based systems to meet specific performance, scalability, and cost requirements.

Complexity vs. Efficiency

A key tradeoff in model selection is balancing complexity and efficiency. While deep learning models may offer superior accuracy for complex tasks like image recognition or natural language processing, they also require significant computational resources and training time. Conversely, simpler models (e.g., rule-based or linear models) may be faster and easier to implement but could struggle with more nuanced or dynamic environments.

For instance, an agent tasked with responding to customer service inquiries may use a simpler, rule-based model if queries are straightforward and well-defined. However, if the agent needs to handle more complex conversations, a neural network-based natural language processing model may be more appropriate, albeit at the cost of higher computational demands.

Scalability and Adaptability

The selected model should also be scalable and adaptable to future changes in the agent's environment or task scope. Models that can scale to handle increasing data volumes or adapt to changing requirements are crucial for long-term success. For example, reinforcement learning models can adapt over time, learning from new experiences to improve their

decision-making capabilities, making them highly adaptable in dynamic environments like autonomous vehicles or smart city management.

Skills

In agent-based systems, *skills* are the fundamental capabilities that enable agents to perform specific actions or solve problems. Skills represent the functional building blocks of an agent, providing the ability to execute tasks and interact with both users and other systems. An agent's effectiveness depends on the range and sophistication of its skills.

Designing Capabilities for Specific Tasks

Skills are typically tailored to the tasks that the agent is designed to solve. When designing skills, developers must consider how the agent will perform under different conditions and contexts. A well-designed skill set ensures that the agent can handle a variety of tasks with precision and efficiency. Skills can be divided into two main categories:

Local Skills

These are actions that the agent performs based on internal logic and computations without external

dependencies. Local skills are often rule-based or involve executing predefined functions. Examples include mathematical calculations, data retrieval from local databases, or simple decision-making based on predefined rules (e.g., deciding whether to approve or deny a request based on set criteria).

API-Based Skills

API-based skills enable agents to interact with external services or data sources. These skills allow agents to extend their capabilities beyond the local environment by fetching real-time data or leveraging third-party systems. For instance, a virtual assistant might use an API to pull weather data, stock prices, or social media updates, enabling it to provide more contextual and relevant responses to user queries.

While local skills allow agents to perform tasks independently using internal logic and rule-based functions, such as calculations or data retrieval from local databases, API-based skills enable agents to connect with external services, accessing real-time data or third-party systems to provide contextually relevant responses and extended functionality.

Skill Integration and Modularity

Modular design is critical for skill development. Each skill should be designed as a self-contained module that can be easily integrated or replaced as needed. This approach allows developers to update or extend the agent's functionality without overhauling the entire system. A customer service chatbot might start with a basic set of skills for handling simple queries and later have more complex skills (e.g., dispute resolution or advanced troubleshooting) added without disrupting the agent's core operations.

Memory

Memory is an essential component that enables agents to store and retrieve information, allowing them to maintain context, learn from past interactions, and improve decision-making over time. Effective memory management ensures that agents can operate efficiently in dynamic environments and adapt to new situations based on historical data.

Short-Term Memory

Short-term memory refers to an agent's ability to store and manage information relevant to the current task or conversation. This type of memory is typically used to maintain context during an interaction, enabling the agent to make

coherent decisions in real time. A customer service agent that remembers a user's previous queries within a session can provide more accurate and context-aware responses, enhancing user experience.

Short-term memory is often implemented using *rolling context windows*, which allow the agent to maintain a sliding window of recent information while discarding outdated data. This is particularly useful in applications like chatbots or virtual assistants, where the agent must remember recent interactions but can forget older, irrelevant details.

Long-Term Memory

Long-term memory, on the other hand, enables agents to store knowledge and experiences over extended periods, allowing them to draw on past information to inform future actions. This is particularly important for agents that need to improve over time or provide personalized experiences based on user preferences.

Long-term memory is often implemented using databases, knowledge graphs, or fine-tuned models. These structures allow agents to store structured data (e.g., user preferences, historical performance metrics) and retrieve it when needed. A

healthcare monitoring agent might retain long-term data on a patient's vital signs, allowing it to detect trends or provide historical insights to healthcare providers.

Memory Management and Retrieval

Effective memory management involves organizing and indexing stored data so that it can be easily retrieved when needed. Agents that rely on memory must be able to differentiate between relevant and irrelevant data and retrieve information quickly to ensure seamless performance. In some cases, agents may also need to forget certain information to avoid cluttering their memory with outdated or unnecessary details.

An e-commerce recommendation agent must store user preferences and past purchase history to provide personalized recommendations. However, it must also prioritize recent data to ensure that recommendations remain relevant and accurate as user preferences change over time.

Planning

Planning is the component that allows agents to sequence their actions and make decisions to achieve specific goals. An agent's

planning capability determines how it navigates complex tasks, prioritizes actions, and adapts to changes in the environment. Without effective planning, agents may execute tasks inefficiently or fail to reach their intended objectives.

Action Sequencing and Goal-Oriented Behavior

Planning involves generating potential action sequences, evaluating the likely outcomes, and selecting the optimal path to achieve the desired result. This is especially important for agents that must complete multi-step tasks, where the order of actions and dependencies between tasks are crucial.

For instance, a logistics management agent might need to plan the sequence of delivery routes based on various factors such as traffic conditions, delivery windows, and vehicle availability. By planning its actions in a logical sequence, the agent can optimize delivery times and reduce costs.

Dynamic Planning and Adaptability

Agents often operate in dynamic environments where conditions can change rapidly. As such, planning systems must be flexible and adaptive, allowing the agent to adjust its plans in response to new information or unexpected events. An autonomous drone tasked with monitoring crop health might

need to alter its flight path if weather conditions change or if it detects an area of higher priority.

Dynamic planning capabilities can be implemented using *search-based algorithms* (e.g., A* search), *optimization techniques*, or *probabilistic models*. These methods enable agents to evaluate multiple possible plans and choose the most efficient or likely to succeed under given constraints.

Incremental and Real-Time Planning

Some agents may benefit from *incremental planning*, where actions are planned in stages, with the agent continuously updating its plan as new information becomes available. This is particularly useful in environments where complete knowledge of the task or environment is not available at the start. For example, a virtual assistant handling a series of interdependent tasks may plan the next action based on the outcome of the previous one, ensuring flexibility and responsiveness. In the next section, we'll discuss some of the most important questions to answer when designing an agentic system.

Design Tradeoffs

Designing agent-based systems involves balancing multiple tradeoffs to optimize performance, scalability, reliability, and cost. These tradeoffs require developers to make strategic decisions that can significantly impact how the agent performs in real-world environments. This section explores the critical tradeoffs involved in creating effective agent systems and provides guidance on how to approach these challenges.

Performance: Speed vs. Accuracy Tradeoffs

A key tradeoff in agent design is balancing speed and accuracy. High performance often enables an agent to quickly process information, make decisions, and execute tasks, but this can come at the expense of precision. Conversely, focusing on accuracy can slow the agent down, particularly when complex models or computationally intensive techniques are required.

In real-time environments, such as autonomous vehicles or trading systems, rapid decision-making is essential, with milliseconds sometimes making a critical difference; here, prioritizing speed over accuracy may be necessary to ensure timely responses. However, tasks like legal analysis or medical diagnostics require high precision, making it acceptable to sacrifice some speed to ensure reliable results.

A hybrid approach can also be effective, where an agent initially provides a fast, approximate response and then refines it with a more accurate follow-up. This approach is common in recommendation systems or diagnostics, where a quick initial suggestion is validated and improved with additional time and data.

Scalability: Engineering Scalability for Agent Systems

Scalability is a critical challenge for modern agent-based systems, especially those that rely heavily on deep learning models and real-time processing. As agent systems grow in complexity, data volume, and task concurrency, managing computational resources, particularly GPUs, becomes paramount. GPUs are the backbone for accelerating the training and inference of large AI models, but efficient scaling requires careful engineering to avoid bottlenecks, underutilization, and rising operational costs. This section outlines strategies for effectively scaling agent systems by optimizing GPU resources and architecture.

GPU resources are often the most expensive and limiting factor in scaling agent systems, making their efficient use a top

priority. Proper resource management allows agents to handle increasing workloads while minimizing the latency and cost associated with high-performance computing. A critical strategy for scalability is dynamic GPU allocation, which involves assigning GPU resources based on real-time demand. Instead of statically allocating GPUs to agents or tasks, dynamic allocation ensures that GPUs are only used when necessary, reducing idle time and optimizing utilization.

Elastic GPU provisioning further enhances efficiency, using cloud services or on-premises GPU clusters that automatically scale resources based on current workloads.

Priority queuing and intelligent task scheduling add another layer of efficiency, allowing high-priority tasks immediate GPU access while queuing less critical ones during peak times.

In large-scale agent systems, latency can become a significant issue, particularly when agents need to interact in real-time or near-real-time environments. Optimizing for minimal latency is essential to ensure that agents remain responsive and capable of meeting performance requirements. Scheduling GPU tasks efficiently across distributed systems can reduce latency and ensure that agents operate smoothly under heavy loads.

One effective strategy is asynchronous task execution, which allows GPU tasks to process in parallel without waiting for previous tasks to complete, maximizing GPU resource utilization and reducing idle time between tasks.

Another strategy is dynamic load balancing across GPUs, which prevents any single GPU from becoming a bottleneck by distributing tasks to underutilized resources. For agent systems reliant on GPU-intensive tasks, such as running complex inference algorithms, scaling effectively requires more than simply adding GPUs; it demands careful optimization to ensure that resources are fully utilized, allowing the system to meet growing demands efficiently.

Scaling GPU-intensive systems effectively requires more than just adding GPUs—it involves ensuring that GPU resources are fully utilized and that the system can scale efficiently as demands grow.

Horizontal scaling involves expanding the system by adding more GPU nodes to handle increasing workloads. In a cluster setup, GPUs can work together to manage high-volume tasks such as real-time inference or model training.

For agent systems with varying workloads, using a hybrid cloud approach can improve scalability by combining on-premises GPU resources with cloud-based GPUs. During peak demand, the system can use burst scaling, in which tasks are offloaded to temporary cloud GPUs, scaling up computational capacity without requiring a permanent investment in physical infrastructure. Once demand decreases, these resources can be released, ensuring cost-efficiency.

Using cloud-based GPU instances during off-peak hours, when demand is lower and pricing is more favorable, can significantly reduce operating costs while maintaining the flexibility to scale up when needed.

Scaling agent systems effectively—particularly those reliant on GPU resources—requires a careful balance between maximizing GPU efficiency, minimizing latency, and ensuring that the system can handle dynamic workloads. By adopting strategies such as dynamic GPU allocation, multi-GPU parallelism, distributed inference, and using hybrid cloud infrastructures, agent systems can scale to meet growing demands while maintaining high performance and cost-efficiency. GPU resource management tools play a critical role in this process, providing the oversight necessary to ensure

seamless scalability as agent systems grow in complexity and scope.

Reliability: Ensuring Robust and Consistent Agent Behavior

Reliability refers to the agent's ability to perform its tasks consistently and accurately over time. A reliable agent must handle expected and unexpected conditions without failure, ensuring a high level of trust from users and stakeholders. However, improving reliability often involves tradeoffs in system complexity, cost, and development time.

Fault Tolerance

One key aspect of reliability is ensuring that agents can handle errors or unexpected events without crashing or behaving unpredictably. This may involve building in *fault tolerance*, where the agent can detect failures (e.g., network interruptions, hardware failures) and recover gracefully. Fault-tolerant systems often employ *redundancy*—duplicating critical components or processes to ensure that failures in one part of the system do not affect overall performance.

Consistency and Robustness

For agents to be reliable, they must perform consistently across different scenarios, inputs, and environments. This is particularly important in safety-critical systems, such as autonomous vehicles or healthcare agents, where a mistake could have serious consequences. Developers must ensure that the agent performs well not only in ideal conditions but also under edge cases, stress tests, and real-world constraints. Achieving reliability requires:

Extensive Testing

Agents should undergo rigorous testing, including unit tests, integration tests, and simulations of real-world scenarios. Tests should cover edge cases, unexpected inputs, and adversarial conditions to ensure that the agent can handle diverse environments.

Monitoring and Feedback Loops

Reliable agents require continuous monitoring in production to detect anomalies and adjust their behavior in response to changing conditions. Feedback loops allow agents to learn from their environment and improve performance over time, increasing their robustness.

Balancing Costs and Returns

Cost is an often-overlooked but critical tradeoff in the design of agent-based systems. The costs associated with developing, deploying, and maintaining an agent must be weighed against the expected benefits and return on investment (ROI). Cost considerations affect decisions related to model complexity, infrastructure, and scalability.

Development Costs

Developing sophisticated agents can be expensive, especially when using advanced machine learning models that require large datasets, specialized expertise, and significant computational resources for training. Additionally, the need for iterative design, testing, and optimization increases development costs.

Complex agents frequently necessitate a team with specialized talent, including data scientists, machine learning engineers, and domain experts to create high-performing systems. Additionally, building a reliable and scalable agent system requires extensive testing infrastructure, often involving simulation environments and investments in testing tools and frameworks to ensure robust functionality.

Operational Costs

After deployment, the operational costs of running agents can become substantial, particularly for systems requiring high computational power, such as those involving real-time decision-making or continuous data processing. Key contributors to these expenses include the need for significant compute power, as agents running deep learning models or complex algorithms often rely on costly hardware like GPUs or cloud services.

Additionally, agents that process vast amounts of data or maintain extensive memory incur higher costs for data storage and bandwidth. Regular maintenance and updates, including bug fixes and system improvements, further add to operational expenses as resources are needed to ensure the system's reliability and performance over time.

Cost vs. Value

Ultimately, the cost of an agent-based system must be justified by the value it delivers. In some cases, it may make sense to prioritize cheaper, simpler agents for less critical tasks, while investing heavily in more sophisticated agents for mission-critical applications. Decisions around cost must be made in the context of the system's overall goals and expected lifespan. Some optimization strategies include:

Lean Models

Using simpler, more efficient models where appropriate can help reduce both development and operational costs. For example, if a rule-based system can achieve similar results to a deep learning model for a given task, the simpler approach will often be more cost-effective.

Cloud-Based Resources

Leveraging cloud computing resources can reduce upfront infrastructure costs, allowing for a more scalable, pay-as-you-go model.

Open-Source Models and Tools

Utilizing open-source machine learning libraries and frameworks can help minimize software development costs while still delivering high-quality agents.

Designing agent systems involves balancing several critical tradeoffs. Prioritizing *performance* may require sacrificing some accuracy, while scaling to a multi-agent architecture introduces challenges in coordination and consistency. Ensuring *reliability* demands rigorous testing and monitoring, but can increase development time and complexity. Finally, *cost* considerations must be factored in from both a development

and operational perspective, ensuring that the system delivers value within budget constraints. In the next section, we'll review some of the most common design patterns used when building effective agentic systems.

Architecture Design Patterns

The architectural design of agent-based systems determines how agents are structured, how they interact with their environment, and how they perform tasks. The choice of architecture influences the system's scalability, maintainability, and flexibility. This section explores three common design patterns for agent-based systems—*single-agent* and *multi-agent* architectures—and discusses their advantages, challenges, and appropriate use cases.

Single-Agent Architectures

A single-agent architecture is among the simplest and most straightforward designs, where a single agent is responsible for managing and executing all tasks within a system. This agent interacts directly with its environment and independently handles decision-making, planning, and execution without relying on other agents.

Ideal for well-defined, narrow tasks, this architecture is best suited for workloads manageable by a single entity. The simplicity of single-agent systems makes them easy to design, develop, and deploy, as they avoid complexities related to coordination, communication, and synchronization across multiple components. With clear use cases, single-agent architectures excel in narrow-scope tasks that do not require collaboration or distributed efforts, such as simple chatbots handling basic customer queries (like FAQs and order tracking) and task-specific automation for data entry or file management.

Single-agent setups work well in environments where the problem domain is well-defined, tasks are straightforward, and there is no significant need for scaling, making them a fit for customer service chatbots, general-purpose assistants, and code generation agents. We'll discuss single-agent and multi-agent architectures much more in chapter 7.

2.4.2 Multi-Agent Architectures: Collaboration, Parallelism, and Coordination

In *multi-agent architectures*, multiple agents work together to achieve a common goal. These agents may operate

independently, in parallel, or through coordinated efforts, depending on the nature of the tasks. Multi-agent systems are often used in complex environments where different aspects of a task need to be managed by specialized agents or where parallel processing can improve efficiency and scalability, and they bring many advantages:

Collaboration and Specialization

Each agent in a multi-agent system can be designed to specialize in specific tasks or areas. For example, one agent may focus on data collection while another processes the data, and a third agent manages user interactions. This division of labor enables the system to handle complex tasks more efficiently than a single agent would.

Parallelism

Multi-agent architectures can leverage parallelism to perform multiple tasks simultaneously. For instance, agents in a logistics system can simultaneously plan different delivery routes, reducing overall processing time and improving efficiency.

Improved Scalability

As the system grows, additional agents can be introduced to handle more tasks or to distribute the workload. This makes multi-agent systems highly scalable and capable of managing larger and more complex environments.

Redundancy and Resilience

Since multiple agents operate independently, failure in one agent does not necessarily compromise the entire system. Other agents can continue to function or even take over the failed agent's responsibilities, improving overall system reliability.

Despite these advantages, multi-agent systems also come with significant challenges, which include:

Coordination and Communication

Managing communication between agents can be complex. Agents must exchange information efficiently and coordinate their actions to avoid duplication of efforts, conflicting actions, or resource contention.

Without proper orchestration, multi-agent systems can become disorganized and inefficient.

Increased Complexity

While multi-agent systems are powerful, they are also more challenging to design, develop, and maintain. The need for communication protocols, coordination strategies, and synchronization mechanisms adds layers of complexity to the system architecture.

Lower Efficiency

While not always the case, multi-agent systems often encounter reduced efficiency due to higher token consumption when completing tasks. Because agents must frequently communicate, share context, and coordinate actions, they consume more processing power and resources compared to single-agent systems. This increased token usage not only leads to higher computational costs but can also slow task completion if communication and coordination are not optimized. Consequently, while multi-agent systems offer robust solutions for complex tasks, their efficiency challenges make careful resource management crucial.

Multi-agent architectures are well-suited for environments where tasks are complex, distributed, or require specialization across different components. In these systems, multiple agents contribute to solving complex, distributed problems, such as in

financial trading systems, cybersecurity investigations, or collaborative AI research platforms.

Single-agent systems offer simplicity and are ideal for well-defined tasks. Multi-agent systems provide collaboration, parallelism, and scalability, making them suitable for complex environments. Choosing the right architecture depends on the complexity of the task, the need for scalability, and the expected lifespan of the system. In the next section, we'll discuss some principles we can follow to get the best results from the agentic systems we build.

Best Practices

Designing agent-based systems requires more than just building agents with the right models, skills, and architecture. To ensure that these systems perform optimally in real-world conditions and continue to evolve as the environment changes, it's essential to follow best practices throughout the development lifecycle. This section highlights three critical best practices—*iterative design*, *agile development*, and *real-world testing*—that contribute to creating adaptable, efficient, and reliable agent systems.

Iterative Design with Continuous Feedback

Iterative design is a fundamental approach in agent development, emphasizing the importance of building systems incrementally while continually incorporating feedback. Instead of aiming for a perfect solution in the initial build, iterative design focuses on creating small, functional prototypes that you can evaluate, improve, and refine over multiple cycles. This process allows for quick identification of flaws, rapid course correction, and continuous system improvement, and has multiple benefits:

Early Detection of Issues

By releasing early prototypes, developers can identify design flaws or performance bottlenecks before they become deeply embedded in the system. This allows for swift remediation of issues, reducing long-term development costs and avoiding major refactors.

User-Centric Design

Iterative design encourages frequent feedback from stakeholders, end users, and other developers. This feedback ensures that the agent system remains aligned

with the users' needs and expectations. As agents are tested in real-world scenarios, iterative improvements can fine-tune their behaviors and responses to better suit the users they serve.

Scalability

Starting with a minimal viable product (MVP) or basic agent allows the system to grow and evolve in manageable increments. As the system matures, new features and capabilities can be introduced gradually, ensuring that each addition is thoroughly tested before full deployment.

To adopt iterative design effectively, development teams should:

1. *Develop Prototypes Quickly*: Focus on building core functionality first. Don't aim for perfection at this stage—build something that works and delivers value, even if it's basic.
2. *Test and Gather Feedback*: After each iteration, collect feedback from users, developers, and other stakeholders. Use this feedback to guide improvements and decide on the next iteration's priorities.
3. *Refine and Repeat*: Based on feedback and performance data, make necessary changes and refine the system in the next

iteration. Continue this cycle until the agent system meets its performance, usability, and scalability goals.

Effective iterative design involves quickly developing functional prototypes, gathering feedback after each iteration, and continuously refining the system based on insights to meet performance and usability goals.

Robust Evaluation

Evaluating the performance and reliability of agent-based systems is a critical part of the development process. A robust evaluation ensures that agents are capable of handling real-world scenarios, performing under varying conditions, and meeting performance expectations. It involves a systematic approach to testing and validating agents across different dimensions, including accuracy, efficiency, robustness, and scalability. This section explores key strategies for creating a comprehensive evaluation framework for agent systems.

A robust evaluation process involves developing a comprehensive testing framework that covers all aspects of the agent's functionality. This framework ensures that the agent is thoroughly tested under a variety of scenarios, both expected and unexpected.

Functional testing focuses on verifying that the agent performs its core tasks correctly. Each skill or module of the agent should be individually tested to ensure that it behaves as expected across different inputs and scenarios. Key areas of focus include:

Correctness

Ensuring that the agent consistently delivers accurate and expected outputs based on its design.

Boundary Testing

Evaluating how the agent handles edge cases and extreme inputs, such as very large datasets, unusual queries, or ambiguous instructions.

Task-Specific Metrics

For agents handling domain-specific tasks (e.g., legal analysis, medical diagnostics), ensure the system meets the domain's accuracy and compliance requirements.

For agent systems, particularly those powered by machine learning models, it is essential to evaluate the agent's ability to generalize beyond the specific scenarios it was trained on. This ensures the agent can handle new, unseen situations while maintaining accuracy and reliability.

Agents often encounter tasks outside of their original training domain. A robust evaluation should test the agent's ability to adapt to these new tasks without requiring extensive retraining. This is particularly important for general-purpose agents or those designed to operate in dynamic environments.

User experience is a key factor in determining the success of agent systems. It's important to evaluate not only the technical performance of the agent but also how well it meets user expectations in real-world applications.

Collecting feedback from actual users provides critical insights into how well the agent performs in practice. This feedback helps refine the agent's behaviors, improving its effectiveness and user satisfaction, and can consist of the following:

User Satisfaction Scores

Use metrics like **Net Promoter Score (NPS)** or **Customer Satisfaction (CSAT)** to gauge how users feel about their interactions with the agent.

Task Completion Rates

Measure how often users successfully complete tasks with the agent's help. Low completion rates may indicate confusion or inefficiencies in the agent's design.

Explicit Signals

Provide opportunities for users to provide their feedback, in such forms as thumbs-up and thumbs-down, star ratings, and the ability to accept, reject, or modify the generated results, depending on the context. These signals can provide a wealth of insight.

Implicit Signals

Analyze user-agent interactions to identify common points of failure, such as misinterpretations, delays, sentiment, or inappropriate responses. Interaction logs can be mined for insights into areas where the agent needs improvement.

In some cases, it's necessary to involve human experts in the evaluation process to assess the agent's decision-making accuracy. Human-in-the-loop validation combines automated evaluation with human judgment, ensuring that the agent's performance aligns with real-world standards. When feasible, human experts should review a sample of the agent's outputs to verify correctness, ethical compliance, and alignment with best practices, and these reviews can then be used to calibrate and improve automated evaluations.

We should evaluate agents in environments that closely simulate their real-world applications. This helps ensure that the system can perform reliably outside of controlled development conditions. Evaluate the agent across the full spectrum of its operational environment, from data ingestion and processing to task execution and output generation. End-to-end testing ensures that the agent functions as expected across multiple systems, data sources, and platforms.

Real-World Testing: Validating Agents in Production Environments

While building agents in a controlled development environment is crucial for initial testing, it's equally important to validate agents in *real-world settings* to ensure they perform as expected when interacting with live users or environments. Real-world testing involves deploying agents in actual production environments and observing their behavior under real-life conditions. This stage of testing allows developers to uncover issues that may not have surfaced during earlier development stages and to evaluate the agent's robustness, reliability, and user impact.

Real-world testing is essential to ensure agents can manage the unpredictability and complexity of live environments. Unlike controlled testing, this approach reveals edge cases, unexpected user inputs, and performance under high demand, helping developers refine the agent for robust, reliable operation.

Exposure to Real-World Complexity

In controlled environments, agents operate with predictable inputs and responses. However, real-world environments are dynamic and unpredictable, with diverse users, edge cases, and unforeseen challenges. Testing in these environments ensures that the agent can handle the complexity and variability of real-world scenarios.

Uncovering Edge Cases

Real-world interactions often expose edge cases that may not have been accounted for in the design or testing phases. For example, a chatbot tested with scripted queries might perform well in development, but when exposed to real users, it may struggle with unexpected inputs, ambiguous questions, or natural language variations.

Evaluating Performance Under Load

Real-world testing also allows developers to observe how the agent performs under high workloads or increased user demand. This is particularly important for agents that operate in environments with fluctuating traffic, such as customer service bots or e-commerce recommendation engines.

Real-world testing ensures an agent's readiness for deployment by validating its performance under real-life conditions. This process involves a phased rollout, continuous monitoring of key metrics, collecting user feedback, and iteratively refining the agent to optimize its capabilities and usability.

1. *Deploy in Phases*: Roll out the agent in stages, starting with small-scale testing in a limited environment before scaling up to full deployment. This phased approach helps identify and address issues incrementally, without overwhelming the system or users.
2. *Monitor Agent Behavior*: Use monitoring tools to track the agent's behavior, responses, and performance metrics during real-world testing. Monitoring should focus on key performance indicators (KPIs) such as response time, accuracy, user satisfaction, and system stability.
3. *Collect User Feedback*: Engage users during real-world testing to gather feedback on their experience interacting with the

agent. User feedback is invaluable in identifying gaps, improving usability, and ensuring that the agent meets real-world needs.

4. *Iterate Based on Insights:* Real-world testing provides valuable insights that should be fed back into the development cycle. Use these insights to refine the agent, improve its capabilities, and optimize its performance for future iterations.

Following best practices such as iterative design, agile development, and real-world testing is critical for building agent-based systems that are adaptable, scalable, and resilient. These practices ensure that agents are designed with flexibility, thoroughly tested in real-world conditions, and continuously improved to meet evolving user needs and environmental challenges. By incorporating these approaches into the development lifecycle, developers can create more reliable, efficient, and effective agent systems capable of thriving in dynamic environments.

Conclusion

The success of any agent-based system hinges on a well-defined and thoughtfully scoped task that aligns with real-world

challenges. By precisely scoping problems, setting clear and measurable objectives, and carefully considering constraints, developers lay the groundwork for agents that are not only effective but also efficient and adaptable. Avoiding common pitfalls, such as overly narrow, broad, or vague tasks, ensures that agents are positioned to fully leverage their potential and deliver tangible results. Coupled with a strategic evaluation process and best practices in design and iteration, agent-based systems can be transformative, solving complex problems with precision and scalability. In the next chapter, we'll learn about skills, which is how we provided advanced functionality to our agents and are a critical piece to increasing the usefulness of our autonomous agents.

Chapter 3. User Experience Design for Agentic Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

As agent systems become an integral part of our digital environments—whether through chatbots, virtual assistants, or fully autonomous workflows—the user experience (UX) they deliver plays a pivotal role in their success. While foundation models and agent architectures enable remarkable technical capabilities, how users interact with these agents ultimately

determines their effectiveness, trustworthiness, and adoption. A well-designed agent experience not only empowers users but also builds confidence, minimizes frustration, and ensures clarity in agent capabilities and limitations.

Designing UX for agent systems introduces unique challenges and opportunities. Agents can interact through a variety of modalities, including text, graphical interfaces, speech, and even video. Each modality brings distinct strengths and trade-offs, requiring thoughtful design to match user needs and context. Additionally, agents operate across synchronous (real-time) and asynchronous (delayed-response) experiences, each of which demands specific design principles to ensure seamless and effective interactions.

Another key UX consideration is context retention and continuity—ensuring that agents can remember past interactions, maintain conversation flow, and adapt to user preferences over time. Without these capabilities, even technically advanced agents risk feeling disjointed or unresponsive. Similarly, communicating agent capabilities, limitations, and uncertainty is essential for setting realistic user expectations and preventing misunderstandings. Users must know what an agent can and cannot do, and when they might need to intervene or provide guidance.

Finally, trust and transparency remain foundational to positive user experiences with agent systems. Predictable agent behavior, clear explanations of actions, and safeguards against automation bias all contribute to building relationships where users feel confident relying on agents in high-stakes scenarios.

This chapter explores these core aspects of User Experience Design for Agentic Systems, offering principles, best practices, and actionable insights to help you design interactions that are intuitive, reliable, and aligned with user needs. Whether you're building a chatbot, an AI-powered personal assistant, or a fully autonomous workflow agent, the principles in this chapter will help you create meaningful and effective experiences that users can trust.

Interaction Modalities

Agent systems interact with users through a variety of modalities, each offering unique strengths, limitations, and design considerations. Whether through text, graphical interfaces, speech, or video, the choice of modality shapes how users perceive and interact with agents. Text-based interfaces excel in clarity and traceability, graphical interfaces offer visual richness and intuitive controls, voice interactions provide

hands-free convenience, and video interfaces enable dynamic, real-time communication.

Each modality introduces distinct design challenges, from managing ambiguity in text interactions to ensuring clarity in voice commands or responsiveness in graphical elements. Additionally, agents must seamlessly navigate synchronous (real-time) interactions and asynchronous (delayed-response) experiences, balancing proactive communication with user control. The modality must align not only with the task at hand but also with user expectations and context, ensuring interactions feel natural, productive, and intuitive.

In this section, we'll explore these interaction modalities, examining their key strengths, challenges, and best practices for delivering exceptional user experiences in agent systems.

Text-Based

Text-based interfaces are one of the most common and versatile ways users interact with agent systems, found in everything from customer service chatbots and command-line tools to productivity assistants integrated into messaging platforms. Their widespread adoption can be attributed to their simplicity, familiarity, and ease of integration into existing workflows. Text

interfaces offer a unique advantage: they can support both real-time synchronous conversations and asynchronous interactions, where users can return to the conversation at their convenience without losing context. Additionally, text interactions create a clear and traceable record of exchanges, enabling transparency, accountability, and easier troubleshooting when something goes wrong.

Designing effective text-based agents requires careful attention to clarity, context retention, and error management. Agents should communicate with concise and unambiguous responses, avoiding overly technical jargon or long-winded explanations that may overwhelm the user. Maintaining context across multi-turn conversations is equally important; users should not need to repeat themselves or clarify past instructions. Effective agents are also graceful in failure, providing clear error messages and fallback mechanisms, such as escalating to a human operator or offering alternative suggestions when they cannot fulfill a request. Turn-taking management is another subtle but crucial element—agents must guide conversations naturally, balancing when to ask follow-up questions and when to pause for user input.

However, text interfaces are not without their challenges. Ambiguity in natural language remains a significant hurdle, as

users may phrase requests in unexpected ways, requiring robust intent recognition to avoid misunderstandings. Additionally, text-based agents are often constrained by response length limits—too short, and they risk being cryptic; too long, and they risk overwhelming or frustrating the user. Emotional nuance is another limitation. Without vocal tone, facial expressions, or visual cues, text-based agents must rely on carefully crafted language to ensure they convey empathy, friendliness, or urgency where appropriate.

Despite these challenges, text-based agents shine in scenarios where precision, traceability, and asynchronous communication are valuable. They excel in customer support, where chatbots provide quick answers to frequently asked questions, or in productivity tools, where command-line interfaces help users execute tasks efficiently. They are equally effective in knowledge retrieval systems, answering specific questions or pulling data from structured databases.

When designed thoughtfully, text-based agents are reliable, adaptable, and deeply useful across a wide range of contexts. Their accessibility and ease of deployment make them a cornerstone of agentic user experience design, provided their limitations are acknowledged and mitigated through clear

communication, robust error handling, and a focus on seamless conversational flow.

Graphical Interfaces

Graphical interfaces offer users a visual and interactive way to engage with agent systems, combining text, buttons, icons, and other graphical elements to facilitate communication. These interfaces are particularly effective for tasks requiring visual clarity, structured workflows, or multi-step processes, where pure text or voice interactions may fall short. Common examples include dashboard-based AI tools, graphical chat interfaces, and agent-powered productivity platforms with clickable elements.

The key strength of graphical interfaces lies in their ability to present information visually and reduce cognitive load. Well-designed interfaces can display complex data, status updates, or task progress in an intuitive and digestible format. Visual cues, such as progress bars, color coding, and alert icons, can guide users effectively without requiring lengthy explanations. For example, an agent managing a workflow might use a dashboard to show pending tasks, completed steps, and error notifications, enabling users to quickly understand the system's state at a glance.

However, designing effective graphical agent interfaces comes with its own challenges. Screen real estate is often limited, requiring designers to prioritize what information is displayed and ensure that critical details are not buried in clutter. Agents must also manage interface responsiveness—users expect real-time updates and smooth transitions between states, especially when agents operate asynchronously. Additionally, graphical elements must adapt gracefully across devices and screen sizes, ensuring consistency whether viewed on a desktop, tablet, or mobile phone.

Another critical consideration is the balance between automation and user control. Graphical interfaces often blend agent autonomy with user-driven actions, such as approving agent-suggested decisions or manually overriding recommendations. For example, an agent suggesting a calendar change might display multiple options through buttons, giving users a clear and efficient way to make a final decision.

Graphical interfaces excel in use cases where data visualization, structured interactions, and clear status updates are essential. Examples include task management dashboards, data analytics tools powered by AI agents, or e-commerce product recommendation systems where users can interact with filters and visual previews. They are also particularly effective in

hybrid workflows where agents operate in the background but present updates or options visually for user confirmation.

When implemented thoughtfully, graphical interfaces enable clear, efficient, and satisfying interactions with agents. They reduce ambiguity, improve task clarity, and offer users a tangible sense of control. By focusing on clarity, responsiveness, and intuitive design patterns, graphical interfaces ensure that agent interactions feel smooth, transparent, and aligned with user expectations.

Speech and Voice Interfaces

Speech and voice interfaces offer users a natural and hands-free way to interact with agent systems, leveraging spoken language as the primary mode of communication. From virtual assistants like Amazon Alexa and Apple Siri to customer service voice bots, these interfaces excel in scenarios where manual input is impractical or impossible, such as while driving, cooking, or operating machinery. They also provide an accessible option for users with visual impairments or limited mobility, making agent systems more inclusive.

Historically, latency has been a major barrier for speech and voice interfaces. Processing spoken language in real-time,

including transcribing speech, interpreting intent, and generating appropriate responses, often led to delays that disrupted the conversational flow. However, recent advancements in low-latency speech recognition models and more efficient language processing architectures have dramatically reduced these delays. At the same time, costs associated with deploying speech-enabled agents have decreased, thanks to more affordable cloud-based APIs and improved on-device processing capabilities.

These improvements are opening the door to a wave of new use cases across industries. In healthcare, voice agents can assist doctors with hands-free note-taking during patient consultations. In customer service, speech agents are replacing traditional IVR (Interactive Voice Response) systems with more fluid, human-like conversations. In industrial applications, voice interfaces allow workers to control machinery, log observations, or access manuals without stopping their tasks. Additionally, smart home devices are becoming increasingly capable of handling multi-turn voice interactions, enabling more seamless and natural user experiences.

Despite these advancements, designing effective speech and voice interfaces requires overcoming unique challenges. Latency, though improved, must remain low for interactions to

feel natural. Misinterpretation of user intent due to background noise, accents, or ambiguous phrasing remains a technical hurdle. Moreover, context retention across voice interactions is crucial—users expect agents to “remember” details from earlier in the conversation without having to repeat themselves.

The next few years are likely to see explosive growth in speech and voice applications, driven by falling costs, reduced latency, and continued improvements in natural language understanding. As agents become more adept at maintaining multi-turn voice interactions and seamlessly transitioning between topics, these interfaces will become a dominant modality across smart home devices, mobile applications, customer service platforms, and industrial systems.

When thoughtfully designed, speech and voice interfaces offer unparalleled convenience, accessibility, and flexibility in agent interactions. As technology continues to mature, these systems are poised to become an indispensable part of daily workflows, personal assistants, and enterprise solutions, fundamentally transforming how users interact with AI-powered agents.

Video Based Interfaces

Video-based interfaces are an emerging modality for agent interactions, blending visual, auditory, and sometimes textual elements into a single cohesive experience. These interfaces can range from video avatars that simulate face-to-face conversations to agents embedded in real-time video collaboration tools. As video becomes more pervasive in our digital lives—through platforms like Zoom, Microsoft Teams, and virtual event spaces—agents are finding new ways to integrate into these environments.

One of the core strengths of video interfaces is their ability to combine multiple sensory channels—visual cues, speech, text overlays, and animations—into a richer, more expressive interaction. Video agents can mimic human-like expressions and gestures, adding emotional nuance to their communication. For example, an AI-powered customer service avatar might use facial expressions and hand gestures to reassure a frustrated customer, complementing its spoken responses with visual empathy.

However, video interfaces come with technical and design challenges. High-quality video interactions require significant processing power and bandwidth, which can introduce lag or pixelation, undermining the user experience. The uncanny valley remains a risk—if an agent’s facial expressions, gestures,

or lip-syncing feel slightly off, it can create discomfort rather than engagement. Additionally, privacy concerns are amplified with video agents, as users may feel uneasy about sharing visual data with AI systems.

Looking ahead, video interfaces are poised for significant growth, especially as improvements in rendering, real-time animation, and bandwidth optimization address current limitations. In the near future, expect to see agents embedded seamlessly into virtual meetings, augmented reality (AR) overlays, and digital customer service avatars.

When thoughtfully executed, video interfaces offer an engaging, human-like dimension to agent interactions, enhancing clarity, emotional connection, and overall effectiveness. As technology advances, video-based agents are set to play a larger role in industries such as telehealth, education, remote collaboration, and interactive entertainment, reshaping how humans and agents communicate in immersive digital spaces.

Synchronous vs Asynchronous Agent Experiences

Agent systems can operate in synchronous or asynchronous modes, each offering distinct advantages and challenges. In synchronous experiences, interactions occur in real-time, with immediate back-and-forth exchanges between the user and the agent. These experiences are common in chat interfaces, voice conversations, and real-time collaboration tools, where quick responses are essential for maintaining flow and engagement. In contrast, asynchronous experiences allow agents and users to operate independently, with communication occurring intermittently over time. Examples include email-like interactions, task notifications, or agent-generated reports delivered after a process has completed.

The choice between synchronous and asynchronous designs depends heavily on the nature of the task, user expectations, and operational context. While synchronous agents excel in tasks requiring instant feedback or live decision-making, asynchronous agents are better suited for workflows where tasks may take longer, require background processing, or don't demand the user's constant attention. Striking the right balance between these modes—and managing when agents proactively engage users—can greatly influence user satisfaction and the overall effectiveness of the system. Both are useful and valid patterns, but it is highly recommended to choose which

experiences fall into which category, so that users do not end up waiting for a pinwheel to spin.

Design Principles for Synchronous Experiences

Synchronous agent experiences thrive on immediacy, clarity, and responsiveness. Users expect agents in these settings to respond quickly and maintain conversation flow without noticeable delays. Whether in a live chat, voice call, or real-time data dashboard, synchronous interactions demand low latency and context awareness to avoid frustrating pauses or repetitive questions.

Agents in synchronous environments should prioritize clarity and brevity in their responses. Long-winded explanations or overly complex outputs can break the rhythm of real-time interactions. Additionally, turn-taking mechanics—knowing when to respond, when to wait, and when to escalate—are critical for maintaining a natural and productive conversation flow. Visual cues, like typing indicators or progress spinners, can reassure users that the agent is actively processing their input.

Error handling is equally important in synchronous designs. Agents must gracefully recover from misunderstandings or failures without derailing the interaction. When uncertainty arises, synchronous agents should ask clarifying questions or gently redirect users rather than making risky assumptions. These principles create a smooth, intuitive experience that keeps users engaged without unnecessary friction.

Design Principles for Async Experiences

Asynchronous agent experiences prioritize flexibility, persistence, and clarity over time. These interactions often occur in contexts where immediate responses aren't necessary, such as when agents are processing long-running tasks, preparing detailed reports, or monitoring background events.

Effective asynchronous agents must excel at clear communication of task status and outcomes. Users should always understand what the agent is doing, what stage a task is in, and when they can expect an update. Notifications, summaries, and well-structured reports become key tools for maintaining transparency. For example, an agent generating an analytical report might notify the user when processing begins, provide an estimated completion time, and deliver a concise, actionable summary when finished.

Context management is another critical design principle for asynchronous agents. Because there may be long delays between user-agent interactions, agents must retain and reference historical context seamlessly. Users shouldn't need to repeat information or retrace previous steps when returning to an ongoing task.

Lastly, asynchronous agents must manage user expectations effectively. Clear timelines, progress indicators, and follow-up notifications prevent frustration caused by uncertainty or lack of visibility into an agent's work.

Finding the balance between proactive and intrusive agent behavior

One of the most delicate aspects of agent design—whether synchronous or asynchronous—is determining when and how agents should proactively engage users. Proactivity can be immensely helpful, such as when an agent alerts a user to an urgent issue, suggests an optimization, or provides a timely reminder. However, poorly timed notifications or intrusive behaviors can frustrate users, disrupt their workflow, or even cause them to disengage entirely.

The key to balancing proactivity lies in context awareness and user control. Agents should understand the user's current focus, level of urgency, and communication preferences. For instance, a proactive alert during a high-stakes video meeting might be more disruptive than helpful, while a notification about a completed task delivered via email might be perfectly appropriate.

Agents should also prioritize relevance when proactively reaching out. Notifications and suggestions must add genuine value, solving problems or providing insights rather than adding noise. Additionally, users should have control over notification frequency, channels, and escalation thresholds, enabling them to customize agent behavior to suit their needs.

Striking this balance isn't just about technical capability—it's about empathy for the user's workflow and mental state. Well-designed agents seamlessly weave proactive engagement into their interactions, enhancing productivity and reducing friction without becoming overbearing.

Context Retention and Continuity

One of the most critical aspects of designing effective agent systems is ensuring context retention and continuity across user interactions. Unlike traditional software interfaces, where every interaction often starts fresh, agents are expected to “remember” previous exchanges, preferences, and task history to provide seamless and meaningful experiences. Whether an agent is guiding a user through a multi-step workflow, continuing a paused conversation, or adjusting its behavior based on past interactions, its ability to maintain context directly impacts usability, efficiency, and user trust.

Effective context retention requires agents to manage both short-term and long-term memory effectively. Short-term memory allows an agent to hold details within an ongoing session, such as remembering the specifics of a question or instructions given moments earlier. Long-term memory, on the other hand, enables agents to retain preferences, past interactions, and broader user patterns across multiple sessions, allowing them to adapt over time.

However, context management introduces challenges. Data persistence, privacy concerns, and memory limitations must all be carefully addressed. If an agent loses track of context mid-task, the user experience can feel disjointed, repetitive, and frustrating. Conversely, if an agent retains too much context or

stores unnecessary details, it risks becoming unwieldy or even breaching user privacy.

In this section, we'll explore two key facets of context retention and continuity: maintaining state across interactions and personalization and adaptability—both essential for delivering fluid, intuitive, and user-centric agent experiences.

Maintaining State Across Interactions

State management is the foundation of context continuity in agent systems. For an interaction to feel seamless, an agent must accurately track what has happened so far, what the user intends to achieve, and what the next logical step is. This is particularly important in multi-turn conversations, task handoffs, and workflows with intermediate states where losing context can result in frustration, inefficiency, and abandonment of tasks.

Agents can maintain state through short-term session memory, where details of the ongoing interaction—such as a user's recent commands or incomplete tasks—are temporarily stored until the session ends. In more advanced systems, persistent state management allows agents to resume tasks across

multiple sessions, enabling users to pick up where they left off, even after hours or days have passed.

Effective state retention requires clear session boundaries, data validation, and fallback mechanisms. If an agent forgets context, it should gracefully recover by asking clarifying questions rather than making incorrect assumptions. Additionally, state data must be managed securely and responsibly, especially when it involves sensitive or personally identifiable information.

When done well, maintaining state allows agents to guide users through complex tasks without unnecessary repetition, reduce cognitive load, and create a sense of ongoing collaboration. Whether an agent is helping a user book travel accommodations, troubleshoot a technical issue, or manage a multi-step approval process, effective state management ensures interactions remain smooth, logical, and productive.

Personalization and Adaptability

Personalization goes beyond merely remembering context—it involves using past interactions and preferences to tailor the agent's behavior, responses, and recommendations to individual users. An adaptable agent doesn't just maintain

state; it learns from previous exchanges to deliver increasingly refined and relevant outcomes. Personalization can take multiple forms:

- **Preference Retention:** Remembering user settings, such as notification preferences or commonly chosen options.
- **Behavioral Adaptation:** Adjusting response style or interaction flow based on observed user patterns.
- **Proactive Assistance:** Anticipating user needs and offering suggestions based on past behavior.

For example, an agent assisting with project management might recognize a user's preferred task-tracking style and adapt its notifications or summaries accordingly. Similarly, a customer service agent might adjust its tone and verbosity based on whether the user prefers concise answers or detailed explanations.

However, personalization comes with challenges. Privacy concerns must be carefully managed, with transparent communication about what data is being stored and how it is being used. Additionally, agents must strike a balance between being helpfully adaptive and overly persistent—users should always have the option to reset or override personalized settings.

The best personalization feels invisible yet impactful, where the agent subtly improves the user experience without drawing attention to its adjustments. At its peak, personalization creates an experience where users feel understood and supported, as if the agent is a thoughtful collaborator rather than a mechanical tool.

Communicating Agent Capabilities

One of the most overlooked yet critical aspects of user experience in agent systems is effectively communicating what the agent can and cannot do. Users often approach agents with varying expectations, ranging from overestimating their intelligence and capabilities to underestimating their usefulness. Misaligned expectations can quickly lead to frustration, mistrust, or missed opportunities to fully leverage the agent's strengths. Clear communication about an agent's capabilities, limitations, and operational context is essential for building trust, preventing misuse, and creating productive interactions.

Effective communication begins at the start of an interaction, where agents should set clear boundaries about their scope and functionality. For example, a travel assistant agent might state

upfront that it can book flights and hotels but cannot handle visa applications. Throughout the interaction, agents must also proactively signal their operational boundaries when users make requests that fall outside their capabilities.

Beyond raw capabilities, agents must also communicate their level of confidence in the information they provide or the actions they recommend. An agent might be highly confident in recommending a widely available product but less certain when answering a niche technical question. Communicating this confidence transparently helps users make informed decisions about whether to trust the agent's output or seek further clarification.

Additionally, effective agents know when to ask for help. Instead of making risky assumptions when faced with ambiguous or incomplete inputs, they should politely ask clarifying questions or seek additional user guidance. For example, an agent tasked with booking a flight might ask, *“Did you mean a one-way or round-trip ticket?”* instead of proceeding with an incorrect assumption.

In this section, we'll explore how agents can set realistic expectations, communicate confidence and uncertainty, and ask users for guidance—all essential practices for fostering clarity,

trust, and productive collaboration between humans and agents.

Setting Realistic Expectations

Setting realistic expectations begins with clear upfront communication about what the agent can achieve and where its limitations lie. Users should know the agent's domain expertise, operational boundaries, and any constraints it may have—whether technical, temporal, or contextual. For example, a virtual assistant might clarify, *“I can help schedule meetings, send emails, and set reminders, but I can’t process payments or handle HR requests.”*

Expectations should also be managed throughout the interaction, particularly when the agent encounters tasks beyond its scope. Instead of failing silently or generating inaccurate outputs, the agent should politely explain its limitations and, where possible, suggest an alternative path or escalate the task to a human operator.

Additionally, agents should avoid overpromising capabilities through exaggerated or overly confident messaging.

Transparency builds trust, and it's better for an agent to admit uncertainty than to mislead the user with false confidence.

When users have a clear understanding of what an agent can deliver, they are more likely to interact confidently and productively, leading to a more satisfying experience overall.

Communicating Confidence and Uncertainty

Agents often operate in probabilistic environments, generating outputs based on statistical models rather than deterministic rules. As a result, not every response or action carries the same degree of confidence. Communicating uncertainty effectively is essential for building user trust and helping users make informed decisions.

Confidence levels can be expressed in several ways:

- **Explicit Statements:** “I’m 90% certain this is the correct answer.”
- **Visual Cues:** Icons, color-coded alerts, or confidence meters in graphical interfaces.
- **Behavioral Adjustments:** Offering suggestions rather than firm recommendations when confidence is low.

For example, a medical assistant agent might say, *“I’m not fully certain about this diagnosis. You might want to consult a doctor*

for confirmation.” In contrast, high-confidence outputs can be presented with more assurance, such as, *“This is the most commonly recommended solution for your issue.”*

However, agents must avoid appearing overly confident when uncertainty is high—users are quick to lose trust if an agent confidently delivers an incorrect or misleading response. Similarly, excessive hedging in low-stakes interactions can make an agent appear hesitant or unreliable.

Communicating confidence and uncertainty isn’t just about sharing probabilities; it’s about framing responses in a way that aligns with user expectations and the stakes of the interaction. In critical contexts, transparency is non-negotiable, while in low-stakes settings, confidence can be presented more casually.

Asking for Guidance and Input from Users

No agent, no matter how advanced, can perfectly interpret ambiguous, vague, or conflicting user inputs. Instead of making risky assumptions, agents must know when to ask clarifying questions or seek user guidance. This ability transforms potential errors into opportunities for collaboration.

Effective agents are designed to ask focused, helpful questions when they encounter ambiguity. For example, if a user says, “*Book me a ticket to Chicago,*” the agent might respond, “*Would you like a one-way or round-trip ticket, and do you have preferred travel dates?*” Instead of defaulting to a generic response or making incorrect assumptions, the agent uses the opportunity to refine its understanding.

The way agents ask for guidance also matters. Questions should be clear, polite, and context-aware, avoiding robotic or repetitive phrasing. If the user has already answered part of the question earlier in the conversation, the agent should reference that context rather than starting from scratch.

Additionally, agents should be transparent about why they’re asking for clarification. A simple explanation like, “*I need a bit more information to proceed accurately,*” helps users understand the rationale behind the question.

Finally, agents should avoid asking too many questions at once—this can overwhelm users and make the interaction feel like an interrogation. Instead, they should sequence questions logically, addressing the most critical ambiguities first.

When agents confidently ask for guidance and input, they transform uncertainty into productive collaboration, empowering users to guide the agent toward successful outcomes while maintaining a sense of partnership and shared control.

Failing Gracefully

Failure is inevitable in agent systems. Whether due to incomplete data, ambiguous user input, technical limitations, or unexpected edge cases, agents will encounter scenarios where they cannot fulfill a request or complete a task. However, how an agent handles failure is just as important as how it handles success. A well-designed agent doesn't just fail—it fails gracefully, minimizing user frustration, preserving trust, and providing a clear path forward.

At its core, graceful failure involves acknowledging the issue transparently, offering a helpful explanation, and suggesting actionable next steps. For instance, if an agent cannot find an answer to a query, it might respond with, *“I couldn't find the information you're looking for. Would you like me to escalate this to a human representative?”* instead of producing an incorrect or nonsensical response.

Agents should also be designed to anticipate common points of failure and have predefined fallback mechanisms in place. For example, if a voice-based agent struggles to understand repeated user inputs, it might switch to a text-based option or provide a clear explanation, such as, *“I’m having trouble understanding your request. Could you please try rephrasing it or typing your question instead?”*

In multi-step tasks, state preservation is equally important when an agent encounters failure. Instead of requiring the user to restart from scratch, the agent should retain progress and allow the user to pick up where they left off once the issue is resolved. This prevents unnecessary repetition and frustration.

Another critical aspect of graceful failure is apologetic and empathetic language. When something goes wrong, the agent should acknowledge the failure in a way that feels human and considerate, avoiding cold or overly technical error messages. For example, *“I’m sorry, something went wrong while processing your request. Let me try again or connect you with someone who can help.”*

Additionally, agents should provide clear paths to resolution. Whether it’s offering troubleshooting steps, escalating to a human operator, or directing the user to an alternative

resource, users should always know what options are available to them when the agent encounters a roadblock.

Lastly, agents must learn from their failures whenever possible. Logging failure points, analyzing recurring issues, and feeding these insights back into the development process can help reduce the frequency of similar failures in the future. Agents that improve iteratively based on their failure patterns will become increasingly resilient and reliable over time.

In summary, failing gracefully is about maintaining user trust and minimizing frustration even when things don't go as planned. By being transparent, empathetic, and action-oriented, agents can turn failures into opportunities to strengthen their relationship with users, demonstrating reliability even in moments of imperfection.

Trust and Transparency

Trust is the cornerstone of effective human-agent interactions. Without it, even the most advanced agent systems will struggle to gain user acceptance, regardless of their capabilities. Transparency and predictability are two of the most powerful tools for building and maintaining trust between agents and

users. Users need to understand *what an agent can do*, *why it made a particular decision*, and *what its limitations are*. This clarity fosters confidence, reduces anxiety, and encourages productive collaboration.

Transparency begins with clear communication of agent capabilities and constraints. Users should never have to guess whether an agent can handle a task or if it's operating within its intended scope. When agents provide explanations for their actions—whether it's how they arrived at a recommendation, why they declined a request, or how they interpreted an ambiguous instruction—they give users visibility into their reasoning. This isn't just about building trust; it also helps users refine their instructions, improving the quality of future interactions.

Predictability complements transparency by ensuring that agents behave consistently across different scenarios. Users should be able to anticipate how an agent will respond based on prior interactions. Erratic or inconsistent behavior, even if technically correct, can quickly erode trust. For example, if an agent suggests a cautious approach in one context but appears overly confident in a nearly identical scenario, users may start to question the agent's reliability.

However, transparency does not mean overwhelming the user with unnecessary details. Users don't need to see every step of the agent's reasoning process—they just need enough insight to feel confident in its actions. Striking this balance requires thoughtful interface design, using visual cues, status messages, and brief explanations to communicate what's happening without causing cognitive overload.

When trust and transparency are prioritized, agent systems become more than just tools—they become reliable collaborators. Users feel confident delegating tasks, following agent recommendations, and relying on their outputs in both casual and high-stakes scenarios. In the following sections, we'll explore two key components of trust-building: ensuring predictability and reliability in agent behavior and preventing automation bias.

Building Predictable and Reliable Agents

Predictability and reliability are foundational to trust. Users must be able to count on agents to behave consistently, respond appropriately, and handle errors gracefully. Agents that act erratically, give conflicting outputs, or produce unexpected behavior—even if occasionally correct—can quickly undermine user confidence.

Reliability begins with consistency in agent outputs. If a user asks an agent the same question under the same conditions, they should receive the same response. In cases where variability is unavoidable (e.g., probabilistic outputs from language models), agents should clearly signal when an answer is uncertain or context-dependent.

Agents must also handle edge cases thoughtfully. For example, when they encounter incomplete data, conflicting instructions, or ambiguous user input, they should respond predictably—either by asking clarifying questions, providing a neutral fallback response, or escalating the issue appropriately.

Another critical aspect of reliability is system resilience. Agents should be designed to recover from errors, maintain state across interruptions, and prevent cascading failures. For example, if an agent loses connection to an external API, it should notify the user, explain the issue, and offer a sensible next step rather than silently failing or producing misleading outputs.

Lastly, reliability is about setting and meeting expectations consistently. If an agent claims it can handle a specific task, it must deliver on that promise every time. Misaligned expectations—where agents overpromise and underdeliver—

can cause more damage to user trust than simply admitting limitations upfront.

When agents behave predictably and reliably, they become dependable digital partners, empowering users to trust their outputs, delegate tasks confidently, and rely on them for critical decisions.

Preventing Automation Bias

While trust is essential for productive human-agent collaboration, over-trust can be just as dangerous as mistrust. Automation bias occurs when users place excessive trust in an agent's outputs, even when there are clear signs that something may be wrong. For example, a user might blindly follow an agent's recommendation for scheduling a critical meeting, even if the agent misunderstood an instruction.

Automation bias often arises because agents are designed to sound confident, precise, and authoritative. Users, especially those unfamiliar with AI systems, might assume that agents are infallible. This can lead to errors cascading unnoticed, particularly in high-stakes workflows such as healthcare diagnostics, financial modeling, or technical troubleshooting.

Preventing automation bias requires designing agents that communicate uncertainty honestly. When an agent isn't fully confident in its recommendation or response, it should make that uncertainty visible—either through verbal disclaimers, confidence scores, or visual cues such as status indicators. For example, an agent might say, *“I’m about 70% sure this is the correct answer, but you might want to double-check this step.”*

Another important strategy is to encourage user engagement and critical thinking. Instead of presenting outputs as final or unquestionable, agents can invite users to verify, adjust, or approve certain actions. For example, an agent suggesting an expense report adjustment might ask, *“Does this change look correct to you?”* This approach nudges users to remain actively involved in the decision-making process rather than deferring entirely to the agent.

Training users is also a powerful tool against automation bias. Organizations should educate users about the strengths and limitations of agent systems and encourage them to treat outputs as helpful suggestions rather than final verdicts.

Ultimately, preventing automation bias is about creating an equitable partnership between users and agents, where users remain informed, attentive, and empowered to intervene when

necessary. By being transparent about confidence levels, encouraging verification, and fostering user engagement, agents can build trust without fostering blind reliance. In this balanced relationship, both human and agent strengths are leveraged effectively, leading to better outcomes and increased user satisfaction.

Conclusion

Designing exceptional user experiences for agent systems goes far beyond technical functionality—it requires a deep understanding of how humans interact with technology across different modalities, contexts, and workflows. Whether through text, graphical interfaces, voice, or video, each interaction modality carries its own strengths, trade-offs, and unique design considerations. Successful agent experiences are those where the modality aligns seamlessly with the user's task, environment, and expectations.

Synchronous and asynchronous agent experiences present distinct design challenges, requiring thoughtful approaches to timing, responsiveness, and clarity. Synchronous interactions demand immediacy and conversational flow, while asynchronous interactions excel in persistence, transparency,

and thoughtful notifications. Striking the right balance between proactive assistance and intrusive interruptions remains one of the most delicate aspects of agent design.

Agents must also excel at context retention and personalization, remembering critical details across interactions and adapting intelligently to user preferences. This ability not only reduces cognitive load but also fosters a sense of continuity and collaboration, transforming agents from isolated tools into reliable digital partners.

Equally important is how agents communicate their capabilities, limitations, and uncertainties. Clear expectations, honest confidence signals, and thoughtful clarification questions create trust, reduce frustration, and prevent misunderstandings. Agents must also know how to fail gracefully, guiding users toward alternative solutions without leaving them stranded or confused.

Finally, building trust through predictability, transparency, and responsible design choices ensures that users can rely on agents without falling into the trap of automation bias. Trust is earned not just through success but also through how agents handle ambiguity, failure, and recovery.

In the years ahead, agent systems will continue to evolve, becoming more deeply embedded in our personal and professional lives. The principles outlined in this chapter—focused on clarity, adaptability, transparency, and trust—provide a blueprint for creating agent experiences that are not just functional, but intuitive, engaging, and deeply aligned with human needs. By prioritizing user experience at every stage of development, we can ensure that agents become not just tools, but indispensable partners in our increasingly intelligent digital ecosystems.

Chapter 4. Skills

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Skills are the building blocks that empower AI agents to perform tasks, make decisions, and interact with the environment in meaningful ways. In the context of AI, a skill can be defined as a specific capability or a set of actions that an agent can perform to achieve a desired outcome. These skills range from simple, single-step tasks to complex, multi-step operations that require advanced reasoning and problem-

solving abilities. Especially if you want your agent to make actual changes, instead of just searching for and providing information, skills will be how those changes are executed.

The significance of skills in AI agents parallels the importance of competencies in human professionals. Just as a doctor needs a diverse set of skills to diagnose and treat patients, an AI agent requires a repertoire of skills to handle various tasks effectively. This chapter aims to provide a comprehensive understanding of skills in AI agents, exploring their design, development, and deployment.

AI agents, at their core, are sophisticated systems designed to interact with their environment, process information, and execute tasks autonomously. To do this efficiently, they rely on a structured set of skills. These skills are modular components that can be developed, tested, and optimized independently, then integrated to form a cohesive system capable of complex behavior.

In practical terms, a skill could be as simple as recognizing an object in an image or as complex as managing a customer support ticket from initial contact to resolution. The design and implementation of these skills are critical to the overall functionality and effectiveness of the AI agent. There are

several types of skills that can be provided to an autonomous agent, which we will cover in sequence: hand-crafted skills,

Local Skills

These skills are manually designed and programmed by developers to run locally. They are often based on predefined rules and logic, tailored to specific tasks. Hand-crafted skills are typically used for well-defined problems where the requirements and outcomes are clear. The process involves defining the logic, rules, and decision-making pathways that the AI agent will follow. These manually-crafted skills offer precision, predictability, and simplicity. Hand-crafted local skills allow developers to have complete control over the behavior of the AI agent. Since the logic is explicitly defined, hand-crafted skills tend to be predictable and reliable. This is beneficial in scenarios where consistency and accuracy are paramount.

The challenges of hand-crafted skills include:

Scalability

Designing hand-crafted skills for complex or dynamic tasks can be time-consuming and challenging. As the

complexity of the task increases, the effort required to program and maintain these skills grows exponentially.

Flexibility

Hand-crafted skills are often rigid and may struggle to adapt to new or unforeseen situations. This lack of flexibility can limit the AI agent's ability to handle diverse or evolving tasks.

Maintenance

As the environment or requirements change, hand-crafted skills may need frequent updates and adjustments. This ongoing maintenance can be resource-intensive.

Despite these drawbacks, manually-crafted skills are especially useful in addressing areas of traditional weakness for foundation models. Simple mathematical operations are a great example of this. Unit conversions, calculator operations, calendar changes, operations over maps and graphs, for example, are all areas where hand-crafted skills can substantially improve the efficacy of agentic systems.

Let's look at an example of registering a calculator skill. First, we define our simple calculator function:

```

from langchain_core.runnables import Configurable
from langchain_core.tools import tool

from langchain_openai import ChatOpenAI

# Define tools using concise function definitions
@tool
def multiply(x: float, y: float) -> float:
    """Multiply 'x' times 'y'."""
    return x * y

@tool
def exponentiate(x: float, y: float) -> float:
    """Raise 'x' to the 'y'."""
    return x**y

@tool
def add(x: float, y: float) -> float:
    """Add 'x' and 'y'."""
    return x + y

```

Then, we register it here as an LLM with tools class from LangChain.

```

tools = [multiply, exponentiate, add]

# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model="gpt-4o", tools=tools)

```

```
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools(tools)
```

Now that we've bound the tool, we can ask the LLM questions, and if the tool is helpful for answering the question, the LLM will choose the tools, select the parameters for those tools, and invoke those functions.

```
query = "What is 393 * 12.25? Also, what is 11 + 49?"
messages = [HumanMessage(query)]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)
for tool_call in ai_msg.tool_calls:
    selected_tool = {"add": add, "multiply": multiply}
    tool_msg = selected_tool.invoke(tool_call)

    print(f'{tool_msg.name} {tool_call['args']} {tool_msg.result}')
    messages.append(tool_msg)
```

With those added print statements for visibility, we can see that the LLM invokes two function calls: one each for multiply and add.

```
multiply {'x': 393, 'y': 12.25} Result: 4814.25
add {'x': 11, 'y': 49} 60.0
```

It then considers the output from both when composing the final response.

```
final_response = llm_with_tools.invoke(messages)
print(final_response.content)

( 393 times 12.25 = 4814.25 )
( 11 + 49 = 60 )
```

Maybe you want to enable your agent to browse the open web for additional information. Doing so is as simple as registering a web surfing agent:

```
from langchain_openai import ChatOpenAI
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper
from langchain_core.messages import HumanMessage

api_wrapper = WikipediaAPIWrapper(top_k_results=5)
tool = WikipediaQueryRun(api_wrapper=api_wrapper)

# Initialize the LLM with GPT-4o and bind the tool
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([tool])
```

```
messages = [HumanMessage("What was the most impressive  
achievement of Buzz Aldrin?")]  
  
ai_msg = llm_with_tools.invoke(messages)  
messages.append(ai_msg)  
  
for tool_call in ai_msg.tool_calls:  
    tool_msg = tool.invoke(tool_call)  
  
    print(tool_msg.name)  
    print(tool_call['args'])  
    print(tool_msg.content)  
    messages.append(tool_msg)  
    print()  
  
final_response = llm_with_tools.invoke(messages)  
print(final_response.content)
```

The LLM identifies the object of interest in the query, and searches Wikipedia for the term. It then uses this additional information to generate its final answer when addressing the question.

```
wikipedia  
{  
  'query': 'Buzz Aldrin'  
}  
Page: Buzz Aldrin  
Summary: Buzz Aldrin (; born Edwin Eugene Aldrin
```


One of the most impressive things about Buzz Aldrin

Similar skills can be created to search across team- or company-specific information. By providing your agent with the skills necessary to access the information it needs to handle a task, and the specific skills to operate over that information, you can significantly expand the scope and complexity of tasks that can be automated.

Skill Design Considerations

Designing effective skills involves several key considerations:

Generalization vs. Specialization

Skills can be designed to be highly specialized for a specific task or generalized to handle a range of related tasks. The choice between generalization and specialization depends on the intended use case and the desired flexibility of the AI agent.

Robustness

Skills should be robust enough to handle variations in input and unexpected scenarios. This requires thorough

testing and refinement to ensure reliability in real-world applications.

Efficiency

Efficient skills minimize computational resources and execution time, enhancing the overall performance of the AI agent. Optimization techniques, such as algorithmic improvements and hardware acceleration, play a vital role in achieving efficiency.

Scalability

As the complexity of tasks increases, skills should scale seamlessly to meet the demands. This involves designing skills that can be easily expanded or combined with other skills to address more sophisticated challenges.

API-Based Skills

API-based skills enable autonomous agents to interact with external services, enhancing their capabilities by accessing additional information, processing data, and executing actions that are not feasible to perform locally. These skills leverage Application Programming Interfaces (APIs) to communicate with public or private services, providing a dynamic and scalable way to extend the functionality of an agent.

API-based skills are particularly valuable in scenarios where the agent needs to integrate with various external systems, retrieve real-time data, or perform complex computations that would be too resource-intensive to handle internally. By connecting to APIs, agents can access a vast array of services, such as weather information, stock market data, translation services, and more, enabling them to provide richer and more accurate responses to user queries. These API-based skills have multiple benefits.

By leveraging external services, these skills can dramatically expand the range of tasks an agent can perform. For instance, an agent can use a weather API to provide current weather conditions and forecasts, a financial API to fetch stock prices, or a translation API to offer multilingual support. This ability to integrate diverse external services greatly broadens the agent's functionality, all without having to retrain a model.

Real-time data access is another major benefit of API-based skills. APIs enable agents to access the most current information from external sources, ensuring that their responses and actions are based on up-to-date data. This is particularly crucial for applications that depend on timely and accurate information, such as financial trading or emergency

response systems, where decisions must be made quickly based on the latest available data.

To illustrate the implementation of API-based skills, consider an example where an agent is designed to fetch and display stock market data. This process involves defining the API interaction, handling the response, and integrating the skill into the agent's workflow. By following this approach, agents can seamlessly integrate external data sources, enhancing their overall functionality and effectiveness.

First, we define the function that interacts with the stock market API. Then, we register this function as a skill for our agent, and we can then invoke it just like the previous skills.

```
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper
from langchain_core.messages import HumanMessage
import requests

@tool
def get_stock_price(ticker: str) -> float:
    """Get the stock price for the stock exchange
    api_url = f"https://api.example.com/stocks/{ticker}"
    response = requests.get(api_url)
```

```
    if response.status_code == 200:
        data = response.json()
        return data["price"]
    else:
        raise ValueError(f"Failed to fetch stock price")

# Initialize the LLM with GPT-4o and bind the tool
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([get_stock_price])

messages = [HumanMessage("What is the stock price of Apple?")]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
    tool_msg = get_stock_price.invoke(tool_call)

    print(tool_msg.name)
    print(tool_call['args'])
    print(tool_msg.content)
    messages.append(tool_msg)
    print()

final_response = llm_with_tools.invoke(messages)
print(final_response.content)
```

This shows how an API-based skill can be integrated into an autonomous agent, allowing it to fetch and provide real-time stock prices.

When designing API-based skills for autonomous agents, several key considerations must be addressed to ensure effective and reliable performance. First and foremost, the reliability of external services is crucial. Agents must be able to handle potential failures gracefully, incorporating fallback mechanisms or providing informative error messages to users when services are unavailable.

Security is another critical aspect. Secure communication with external APIs, especially when dealing with sensitive information, is essential. Implementing HTTPS for data transmission and robust authentication and authorization mechanisms helps protect against unauthorized access and ensures data integrity.

We need to also be mindful of API rate limits imposed by external services. Designing the agent to respect these limits prevents service disruptions and maintains continuous access to necessary data. Additionally, data privacy must be a priority. When interacting with external services, it's important to

ensure that all shared and retrieved data complies with relevant data privacy regulations. Sensitive data should be anonymized or obfuscated as needed to protect user privacy.

Effective error handling is also important for managing various failure scenarios, such as network issues, invalid API responses, or service outages. The agent should be capable of recovering from errors and continuing to function without significant interruptions, ensuring a seamless user experience.

API-based skills significantly enhance the capabilities of autonomous agents by leveraging external services. These skills enable agents to access real-time data, perform complex computations, and execute tasks that are beyond their local capabilities, thereby improving their overall functionality and effectiveness.

Plug-in Skills

These skills are modular and can be integrated into the AI agent's framework with minimal customization. They leverage existing libraries, APIs, and third-party services to extend the agent's capabilities without extensive development effort. Plug-in skills enable rapid deployment and scaling of the agent's functionalities. These skills are pre-designed modules that can

be integrated into an AI system with minimal effort, leveraging existing libraries, APIs, and third-party services. The integration of plug-in skills has become a standard offering from leading platforms such as OpenAI, Anthropic's Claude, Google's Gemini, and Microsoft's Phi, providing powerful tools to expand the capabilities of AI agents without extensive custom development, as well as within a growing open-source community.

OpenAI offers a suite of plug-in skills that enable functionalities like natural language understanding, code generation, and data analysis. These plug-ins are designed to be incorporated at the model execution layer, allowing seamless integration into existing workflows. This approach makes it straightforward to add advanced capabilities to AI applications without the need for deep customization. Similarly, Anthropic's Claude focuses on ethical AI and robust performance, providing skills for content moderation, bias detection, and robust decision-making. These ensure that AI systems operate within safe and ethical boundaries, aligning with societal norms.

Google's Gemini platform provides a diverse range of plug-in skills for applications such as natural language processing and computer vision. Leveraging Google's extensive AI research and infrastructure, these plug-ins enable functionalities like image

recognition, speech synthesis, and language translation. Microsoft's Phi platform integrates plug-in skills seamlessly with its suite of productivity tools, enabling tasks such as document processing, data visualization, and workflow automation. This deep integration allows developers to create AI solutions that fit neatly into existing business processes and tools.

One of the significant advantages of plug-in skills is their integration at the model execution layer. This means these skills can be added to AI models with minimal disruption to existing workflows. Developers can simply plug these modules into their AI systems, instantly enhancing their capabilities without extensive customization or development effort. This ease of integration makes plug-in skills an attractive option for rapidly deploying new functionalities in AI applications. However, this ease of use comes with certain limitations. Plug-in skills, while powerful, do not offer the same level of customizability and adaptability as custom-developed solutions. They are designed to be general-purpose tools that can address a broad range of tasks, but they may not be tailored to the specific needs and nuances of every application. This trade-off between ease of integration and customizability is an important consideration for developers when choosing between plug-in skills and bespoke development.

Despite the current limitations, the catalogs of plug-in skills offered by leading platforms are rapidly growing. As these catalogs expand, the breadth of capabilities available through plug-in skills will increase, providing developers with even more tools to enhance their AI agents. This growth is driven by continuous advancements in AI research and the development of new techniques and technologies. In the near future, we can expect these plug-in skill catalogs to include more specialized and advanced functionalities, catering to a wider range of applications and industries. This expansion will facilitate agent development by providing developers with readily available tools to address complex and diverse tasks. The growing ecosystem of plug-in skills will enable AI agents to perform increasingly sophisticated functions, making them more versatile and effective in various domains.

In addition to the offerings from major platforms, there is a rapidly growing ecosystem of tools and plug-in skills that can be incorporated into open-source foundation models. This ecosystem provides a wealth of resources for developers looking to enhance their AI agents with advanced capabilities. Open-source communities are actively contributing to the development of plug-in skills, creating a collaborative environment that fosters innovation and knowledge sharing. One notable example is the Hugging Face Transformers library,

which offers a wide range of pre-trained models and plug-in skills for natural language processing tasks. These skills can be easily integrated into open-source foundation models, enabling functionalities such as text generation, sentiment analysis, and language translation. The open-source nature of this library allows developers to customize and extend these skills to suit their specific needs. Both TensorFlow and PyTorch, two leading deep learning frameworks, have extensive ecosystems of plug-in skills and tools. These ecosystems include pre-trained models, data processing pipelines, and optimization tools that can be integrated into AI applications. The flexibility of these frameworks allows developers to combine plug-in skills with custom development, creating powerful and adaptable AI systems. The open-source AI community is continuously contributing new plug-in skills and enhancements, driven by the collective efforts of researchers, developers, and enthusiasts. Platforms like GitHub host numerous repositories of plug-in skills, ranging from simple utilities to complex models. These contributions enrich the ecosystem and provide valuable resources for developers looking to leverage the latest advancements in AI.

The practical applications of plug-in skills are vast and varied, spanning multiple industries and use cases. By integrating plug-in skills, developers can create AI agents that perform a wide

range of tasks efficiently and effectively. In customer support, plug-in skills can enable AI agents to handle queries, provide solutions, and manage support tickets. Skills like natural language understanding and sentiment analysis can help AI agents understand customer issues and respond appropriately, improving customer satisfaction and reducing response times. In healthcare, plug-in skills can assist AI agents in tasks such as medical image analysis, patient triage, and data management. Skills that leverage computer vision can help identify abnormalities in medical images, while natural language processing skills can assist in managing patient records and extracting relevant information from medical literature. In the finance industry, plug-in skills can enhance AI agents' abilities to analyze market trends, detect fraudulent activities, and manage financial portfolios. Skills like anomaly detection and predictive analytics can provide valuable insights and improve decision-making processes. In education, plug-in skills can support AI agents in personalized learning, automated grading, and content recommendation. Natural language processing skills can help analyze student responses, while recommendation algorithms can suggest relevant study materials based on individual learning needs.

The future of plug-in skills in AI development looks promising, with continuous advancements and growing adoption across

various industries. As the capabilities of plug-in skills expand, we can expect AI agents to become even more capable and versatile. The ongoing research and development efforts by leading platforms and the open-source community will drive innovation, resulting in more powerful and sophisticated tools for AI development. One important area of focus for the future is the interoperability and standardization of plug-in skills. Establishing common standards and protocols for plug-in skills will facilitate seamless integration and interoperability across different AI platforms and systems. This will enable developers to leverage plug-in skills from various sources, creating more flexible and adaptable AI solutions. Efforts are also being made to enhance the customization and adaptability of plug-in skills. Future plug-in skills may offer more configurable options, allowing developers to tailor them to specific use cases and requirements. This will bridge the gap between the ease of integration and the need for customized solutions, providing the best of both worlds.

Skill Hierarchies

Though not necessary for smaller projects or early on in the development of an agentic application, skills can also be organized hierarchically, where complex skills are composed of

simpler sub-skills. Skill hierarchies involve organizing skills in a structured manner where complex tasks are decomposed into simpler sub-skills. Grouping related skills into hierarchies can streamline skill selection and execution, minimizing the risk of semantic collisions—situations where multiple skills overlap in functionality or purpose, leading to conflicts or ambiguities.

By categorizing skills into well-defined groups, developers can ensure that the AI agent selects the appropriate skill for a given task, improving both efficiency and accuracy. This structured approach allows for better management of the agent's capabilities, as each group can be designed to handle specific types of tasks, making the overall system more robust and easier to maintain.

For example, in a customer support AI, skills related to account management, technical support, and billing can be grouped separately. When a user query is processed, the AI can quickly identify the relevant skill group and select the most appropriate skill within that group, avoiding confusion and ensuring a smooth user experience.

The detailed process of skill selection and managing these hierarchies falls under the broader topic of orchestration, which will be covered comprehensively in the subsequent

chapter. Orchestration involves coordinating multiple skills and ensuring they work together seamlessly to achieve the desired outcomes, making it a critical aspect of advanced AI agent development.

Automated Skill Development

Code generation is a technique where AI agents write code autonomously, significantly reducing the time and effort required to create and maintain software applications. This process involves training models on vast amounts of code data, enabling them to understand programming languages, coding patterns, and best practices. This approach has three main

Code generation represents a transformative leap in AI capabilities, particularly when an agent writes its own skills in real-time to solve tasks or interact with new APIs. This dynamic approach enables AI agents to adapt and expand their functionality on-the-fly, significantly enhancing their versatility and problem-solving capacity.

Real-Time Code Generation

Real-time code generation involves an AI agent writing and executing code as needed during its operation. This capability

allows the agent to create new skills or modify existing ones to address specific tasks, making it highly adaptable. For instance, if an AI agent encounters a novel API or an unfamiliar problem, it can generate code to interface with the API or develop a solution to the problem in real-time.

The process begins with the agent analyzing the task at hand and determining the necessary steps to accomplish it. Based on its understanding, the agent writes code snippets, which it then attempts to execute. If the code does not perform as expected, the agent iteratively revises it, learning from each attempt until it achieves the desired outcome. This iterative process of trial-and-error allows the agent to refine its skills continuously, improving its performance and expanding its capabilities autonomously.

Real-time code generation offers several compelling advantages, particularly in terms of adaptability and efficiency. The ability to generate code on-the-fly allows AI agents to quickly adapt to new tasks and environments. This adaptability is crucial for applications requiring dynamic problem-solving and flexibility, such as real-time data analysis, autonomous systems, and complex software integration tasks. By generating code in real-time, AI agents can address immediate needs without waiting for human intervention, significantly speeding

up processes, reducing downtime, and enhancing overall efficiency.

However, real-time code generation also presents several challenges and risks. Quality control is a major concern, as ensuring the quality and security of autonomously generated code is critical. The AI must adhere to best practices and coding standards to prevent bugs, vulnerabilities, and unintended behaviors. Poor quality code can lead to system failures, security breaches, and other significant issues. Security risks are another major challenge, as allowing AI agents to execute self-generated code introduces the potential for malicious actors to exploit this capability to inject harmful code, leading to data breaches, unauthorized access, or system damage. Implementing robust security measures and oversight is essential to mitigate these risks.

Resource consumption is also a critical consideration, as real-time code generation and execution can be resource-intensive, requiring substantial computational power and memory. Ensuring that the AI agent operates efficiently without overloading system resources is crucial. Finally, the autonomous nature of real-time code generation raises ethical and regulatory concerns. Ensuring that the AI agent operates within legal and ethical boundaries is critical, particularly in

sensitive areas such as healthcare, finance, and autonomous vehicles. Addressing these concerns involves careful consideration of the implications of autonomous code generation and implementing appropriate safeguards to ensure responsible and compliant operation.

Imitation Learning

Imitation learning is a technique where AI agents learn to perform tasks by observing and mimicking human behavior. This approach is inspired by how humans and animals learn through imitation and demonstration. Imitation learning is particularly effective for tasks that are difficult to define with explicit rules or for which large datasets of labeled examples are not available.

Imitation learning offers several notable advantages for training AI agents. By mimicking the process of human learning, imitation learning allows AI agents to acquire skills in a manner that resembles how humans learn, making it particularly effective for complex and nuanced tasks. This human-like learning approach simplifies the training process for agents, enabling them to grasp intricate behaviors more naturally.

One of the primary benefits of imitation learning is its efficiency. AI agents can quickly learn new skills by observing demonstrations, bypassing the need for extensive trial-and-error methods or manually labeled data. This rapid acquisition of skills accelerates the training process and reduces the resources required to develop proficient AI systems.

Furthermore, imitation learning is highly versatile, applicable to a broad spectrum of tasks ranging from robotic manipulation to natural language understanding. This versatility makes it a valuable tool in the AI development toolkit, suitable for a wide array of applications.

However, imitation learning also presents several challenges. First, not all teams, companies, or scenarios have the data necessary to attempt imitation learning. The quality of demonstrations is critical; poor or incomplete demonstrations can significantly impair the effectiveness of the learning process, leading to suboptimal performance. Another challenge is ensuring that AI agents can generalize from the observed behavior to new, unseen situations. The ability to adapt learned skills to various contexts and environments is essential for the practical deployment of AI agents. Additionally, scaling imitation learning to handle complex tasks with numerous variations demands substantial computational resources and

sophisticated algorithms, posing a significant challenge for developers.

Several techniques and approaches are employed in imitation learning to address these challenges. Behavior cloning involves training the AI agent to directly mimic the actions of a human demonstrator using supervised learning to map observations to actions based on the demonstrated behavior. Inverse Reinforcement Learning (IRL) aims to infer the underlying reward function that the human demonstrator is optimizing. The AI agent then uses this inferred reward function to guide its own behavior, allowing it to generalize beyond specific demonstrations. Generative Adversarial Imitation Learning (GAIL) combines the principles of generative adversarial networks (GANs) with imitation learning. In this approach, the AI agent learns to generate behavior that is indistinguishable from human demonstrations, while a discriminator model evaluates the quality of the generated behavior.

Skill Learning from Rewards

Skill learning from rewards, also known as reinforcement learning (RL), involves training AI agents to perform tasks by maximizing a reward signal. Unlike imitation learning, where the agent learns from demonstrations, reinforcement learning

agents learn through trial-and-error, receiving feedback in the form of rewards or penalties based on their actions.

Skill learning from rewards, also known as reinforcement learning (RL), provides several significant advantages for developing autonomous AI agents. One of the primary benefits is autonomy, as RL allows AI agents to learn independently without the need for extensive human intervention or labeled data. This capability makes RL particularly valuable for applications where continuous human supervision is impractical. Additionally, RL agents engage in exploration, navigating their environment to discover optimal strategies for achieving their goals. This exploratory nature makes them well-suited for tasks in complex and dynamic environments, where predefined rules may be insufficient.

Optimality is another critical advantage of skill learning from rewards. By focusing on maximizing rewards, RL agents can achieve high levels of performance and efficiency, often surpassing human capabilities in specific tasks. This optimization enables AI agents to perform at peak levels, adapting and improving their strategies based on their experiences and the feedback they receive from their environment.

Despite these advantages, skill learning from rewards also presents several challenges. First is having a high-quality environment with rewards. Some scenarios, such as customer service or finance, can use past data to create a realistic environment with rewards, but this is not possible in all cases. Sample efficiency is a notable issue, as RL algorithms typically require a large number of interactions with the environment to learn effectively. This process can be time-consuming and computationally expensive, posing a significant hurdle for practical applications. Stability is another challenge, as ensuring the stability and convergence of RL algorithms can be difficult, particularly in environments with high variability or noise. Finally, reward design is crucial; creating appropriate reward functions that accurately reflect the desired outcomes is essential for the success of RL agents. Poorly designed rewards can lead to unintended behaviors, undermining the effectiveness of the learning process.

To address these challenges, various techniques and approaches have been developed in skill learning from rewards. Value-based methods, such as Q-learning and Deep Q-Networks (DQNs), involve estimating the value of actions in different states and selecting actions that maximize the expected value. These methods are particularly effective for discrete action spaces. Policy-based methods, including

REINFORCE and Proximal Policy Optimization (PPO), directly optimize the agent's policy by adjusting the parameters of a policy network. These methods are well-suited for continuous action spaces and provide a more direct approach to policy improvement. Actor-critic methods combine the strengths of value-based and policy-based approaches. In this framework, the actor component selects actions, while the critic component evaluates the actions' value, providing feedback to improve the policy. This combination allows for more stable and efficient learning, leveraging the benefits of both approaches.

By utilizing these techniques, reinforcement learning enables the development of autonomous AI agents capable of learning complex skills and adapting to a wide range of environments. The ability to explore, optimize, and operate independently makes RL a powerful tool for advancing AI capabilities and achieving high levels of performance in various applications.

Conclusion

Skills enable AI agents to perform tasks, make decisions, and interact with their environment effectively. These range from simple to complex tasks requiring advanced reasoning. Hand-crafted skills, manually designed by developers, offer precision

but can be time-consuming to maintain. Plug-in skills, provided by platforms like OpenAI and Google's Gemini, allow rapid integration and scalability but lack customizability. Skill hierarchies organize skills into structured groups, enhancing efficiency and reducing conflicts.

Automated skill development, including real-time code generation, imitation learning, and reinforcement learning, allows AI agents to dynamically adapt and refine their abilities. This enhances their versatility and problem-solving capabilities, enabling continuous improvement and autonomous expansion of skills. Building and maintaining the skillset for your agent is one of the most critical ways to give your agent the capabilities to succeed in the task at hand. Now that we know how to build and curate a set of skills that we provide to our agent, we'll move on to consider how we'll enable the agent to make plans, select and parameterize skills, and put these pieces together in order to perform useful work.

Chapter 5. Orchestration

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Now that your agent has a set of skills that can be used, it’s time to start using them to solve real tasks. While simple tasks can be handled by relying on the intrinsic information contained within the models’ weights, and some tasks can be handled with a single skill, more complex tasks will often require multiple skills, with the potential for dependencies between these skills. To handle more complex tasks, agents need to

perform more than one action and do so in a reasonable order, appropriately managing dependencies between the tasks. This chapter will cover orchestration, including skill selection, execution, skill topologies, and planning.

Skill Selection

Before we get to planning, we will consider the critical task of skill selection, because it is the foundation for more advanced planning. Different approaches to skill selection offer unique advantages and considerations, meeting different requirements and environments. We assume a set of skills have already been developed, so if you need a refresher, go back to Chapter 3.

Generative Skill Selection

The simplest approach is Generative Skill Selection. In this case, the skill, its definition, and its description are provided to a foundation model, and the model is asked to select the most appropriate skill for the given context. The output from the foundation model is then compared to the skillset, and the closest one is chosen. This approach is easy to implement, and requires no additional training, embedding, or a skillset hierarchy to use. The main drawback is latency, as it requires

another foundation model call, which can add seconds to the overall response time. It can also benefit from in-context learning, where few-shot examples can be provided to boost predictive accuracy for your problem without the challenge of training or fine-tuning a model.

```
from langchain_core.tools import tool
import requests

@tool def query_wolfram_alpha(expression: str) -> str:
    """ Query Wolfram Alpha to compute mathematical expressions.
    Args: expression (str): The mathematical expression to compute.
    Returns: str: The result of the computation.
    """
    api_url = f"https://api.wolframalpha.com/v1/..."

@tool def trigger_zapier_webhook(zap_id: str, payload: dict) -> str:
    """ Trigger a Zapier webhook to execute a pre-defined action.
    zap_id (str): The unique identifier for the Zapier webhook.
    payload (dict): The data to send to the Zapier webhook.
    str: Confirmation message upon successful trigger.
    Raises: ValueError: If the API request fails.
    """

    url = f"https://hooks.zapier.com/hooks/catch/{zap_id}/..."
    try:
        response = requests.post(zapier_webhook_url, json=payload)
        if response.status_code == 200:
            return "Webhook triggered successfully."
        else:
            raise ValueError(f"API request failed with status code {response.status_code}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"API request failed with error {e}")
```

```

        return f"Zapier webhook '{zap_id}' triggered"

    else:
        raise ValueError(f"Zapier API Error: {e}")
except requests.exceptions.RequestException:
    raise ValueError(f"Failed to trigger Zapier webhook")

@tool def send_slack_message(channel: str, message: str):
    """ Send a message to a specified Slack channel """

    api_url = "https://slack.com/api/chat.postMessage"
    payload = { "channel": channel, "text": message }

    try:
        response = requests.post(api_url, headers=headers, json=payload)
        response_data = response.json()
        if response.status_code == 200 and response_data.get("ok"):
            return f"Message successfully sent to {channel}"
        else:
            error_msg = response_data.get("error", "Unknown error")
            raise ValueError(f"Slack API Error: {error_msg}")
    except requests.exceptions.RequestException:
        raise ValueError(f"Failed to send message to Slack")

# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([get_stock_price, send_slack_message])

# Example usage:
prompt = "What is the stock price of Apple?"
response = llm_with_tools.invoke(prompt)
print(response)

```

```
messages = [HumanMessage("what is the stock price of Tesla?")]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
    tool_msg = get_stock_price.invoke(tool_call)

    print(tool_msg.name)

    print(tool_call['args'])
    print(tool_msg.content)
    messages.append(tool_msg)
    print()

final_response = llm_with_tools.invoke(messages)
print(final_response.content)
```

Semantic Skill Selection

Another approach, Semantic Skill Selection, uses semantic representations to identify and select the relevant skill based on their semantic similarity to the task requirements. Ahead of time, each skill definition and description is embedded using an encoder-only model, such as OpenAI's ada model, Amazon's Titan model, Cohere's Embed model, BERT, or others. These

skills are then indexed in a lightweight vector database. At runtime, the current context is then embedded using the same embedding model, a search is performed on the database, and the top skill is selected. The skill is then parameterized with a text completion model, invoked, and the response is used to compose the response for the user. This is the most common pattern, and is recommended for most use cases. It's typically faster than Generative Skill Selection, performant, and reasonably scalable.

Skill database setup:

```
import os
import requests
import logging
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.messages import HumanMessage
from langchain.vectorstores import FAISS
import faiss
import numpy as np

# Initialize OpenAI embeddings and LLM
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
llm = ChatOpenAI(api_key=OPENAI_API_KEY)

# Tool descriptions
```

```
tool_descriptions = {  
    "query_wolfram_alpha": "Use Wolfram Alpha to",  
    "trigger_zapier_webhook": "Trigger a Zapier w",  
    "send_slack_message": "Send messages to spec",  
}
```

```
# Create embeddings for each tool description
```

```
tool_embeddings = []
```

```
tool_names = []
```

```
for tool_name, description in tool_descriptions.items():
```

```
    embedding = embeddings.embed_text(description)
```

```
    tool_embeddings.append(embedding)
```

```
    tool_names.append(tool_name)
```

```
# Initialize FAISS vector store
```

```
dimension = len(tool_embeddings[0]) # Assuming all embeddings are the same dimension
```

```
index = faiss.IndexFlatL2(dimension)
```

```
# Normalize embeddings for cosine similarity
```

```
faiss.normalize_L2(np.array(tool_embeddings).astype(np.float32))
```

```
# Convert list to FAISS-compatible format
```

```
tool_embeddings_np = np.array(tool_embeddings).astype(np.float32)
```

```
index.add(tool_embeddings_np)
```

```
# Map index to tool functions
```

```
index_to_tool = {}
```

```
    0: "query_wolfram_alpha"
```

```

0: query_with_tool_alpha ,
1: "trigger_zapier_webhook",
2: "send_slack_message"
}

```

```

def select_tool(query: str, top_k: int = 1) -> list:
    """
    Select the most relevant tool(s) based on the query.

    Args:
        query (str): The user's input query.
        top_k (int): Number of top tools to retrieve.

    Returns:
        list: List of selected tool functions.
    """
    query_embedding = embeddings.embed_text(query)
    query_embedding = faiss.normalize_L2(query_embedding.reshape(1, -1))
    D, I = index.search(query_embedding.reshape(1, -1), k=top_k)
    selected_tools = [index_to_tool[idx] for idx in I[0]]
    return selected_tools

```

```

def determine_parameters(query: str, tool_name: str) -> dict:
    """
    Use the LLM to analyze the query and determine the parameters for the tool.

    Args:
        query (str): The user's input query.
        tool_name (str): The selected tool name.
    """

```



```
tool_name (str): The selected tool name.
```

Returns:

```
dict: Parameters for the tool.
```

```
"""
```

```
messages = [  
    HumanMessage(content=f"Based on the user]
```

```
# Call the LLM to extract parameters  
response = llm(messages)
```

```
# Example logic to parse response from LLM  
parameters = {}
```

```
if tool_name == "query_wolfram_alpha":  
    parameters["expression"] = response['exp  
elif tool_name == "trigger_zapier_webhook":  
    parameters["zap_id"] = response.get('zap_  
    parameters["payload"] = response.get('pay  
elif tool_name == "send_slack_message":  
    parameters["channel"] = response.get('cha  
    parameters["message"] = response.get('mes
```

```
return parameters
```

```
# Example user query
```

```
user_query = "Solve this equation:  $2x + 3 = 7$ "
```

```
# Select the top tool
```

```

# Select the top tool
selected_tools = select_tool(user_query, top_k=1)
tool_name = selected_tools[0] if selected_tools else None

if tool_name:

    # Use LLM to determine the parameters based on the query
    args = determine_parameters(user_query, tool_name)

    # Invoke the selected tool
    try:
        # Assuming each tool has an `invoke` method
        tool_result = globals()[tool_name].invoke(**args)
        print(f"Tool '{tool_name}' Result: {tool_result}")
    except ValueError as e:
        print(f"Error invoking tool '{tool_name}': {e}")
    else:
        print("No tool was selected.")

```

If your scenario involves a large number of skills, however, you might need to consider Hierarchical Skill Selection. This is especially true if many of those skills are semantically similar, and you are looking to improve skill selection accuracy at the price of higher latency and complexity. In this pattern, you organize your skills into groups, and provide a description for each group. Your skill selection (either Generative or Semantic) first selects a group, and then performs a secondary search only

among the skills in that group. While this is slower and would be expensive to parallelize, it reduces the complexity of the skill selection task into two smaller chunks, and frequently results in higher overall skill selection accuracy. Crafting and maintaining these skill groups takes time and effort, so this is not recommended as a technique to begin with.

```
import os
import requests

import logging
import numpy as np

from langchain_core.tools import tool
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.messages import HumanMessage,
from langchain.vectorstores import FAISS
import faiss

embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
# Define tool groups with descriptions
tool_groups = {
    "Computation": {
        "description": "Tools related to mathematical computation",
        "tools": []
    },
    "Automation": {
        "description": "Tools that automate workflow tasks"
```

```

        "tools": []
    },
    "Communication": {
        "description": "Tools that facilitate communication"
        "tools": []
    }
}

```

Define Tools

@tool

```

def query_wolfram_alpha(expression: str) -> str:
    api_url = f"https://api.wolframalpha.com/v1/queries"
    try:
        response = requests.get(api_url)
        if response.status_code == 200:
            return response.text
        else:
            raise ValueError(f"Wolfram Alpha API returned status code {response.status_code}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to query Wolfram Alpha: {e}")

```

@tool

```

def trigger_zapier_webhook(zap_id: str, payload: dict):
    """
    Trigger a Zapier webhook to execute a predefined workflow.

    Args:
        zap_id (str): The unique identifier for the Zapier workflow.
    """

```

payload (dict): The data to send to the Zapier API

Returns:

str: Confirmation message upon successful Zapier webhook trigger

Raises:

ValueError: If the API request fails or if the Zapier ID is not provided

"""

zapier_webhook_url = f"https://hooks.zapier.com/hooks/catch/{zap_id}/" + payload

try:

response = requests.post(zapier_webhook_url, data=payload)

if response.status_code == 200:

return f"Zapier webhook '{zap_id}' successfully triggered"

else:

raise ValueError(f"Zapier API Error: {response.status_code} {response.text}")

except requests.exceptions.RequestException as e:

raise ValueError(f"Failed to trigger Zapier webhook: {e}")

@tool

def send_slack_message(channel: str, message: str):

"""

Send a message to a specified Slack channel.

Args:

channel (str): The Slack channel ID or name

message (str): The content of the message

Returns:

```
    str: Confirmation message upon successful
```

```
Raises:
```

```
    ValueError: If the API request fails or i
```

```
"""
```

```
api_url = "https://slack.com/api/chat.postMessage"
```

```
headers = {
```

```
    "Authorization": f"Bearer {SLACK_BOT_TOKEN}"
```

```
    "Content-Type": "application/json"
```

```
}
```

```
payload = {
```

```
    "channel": channel,
```

```
    "text": message
```

```
}
```

```
try:
```

```
    response = requests.post(api_url, headers
```

```
    response_data = response.json()
```

```
    if response.status_code == 200 and respon
```

```
        return f"Message successfully sent to
```

```
    else:
```

```
        error_msg = response_data.get("error"
```

```
        raise ValueError(f"Slack API Error: "
```

```
except requests.exceptions.RequestException a
```

```
    raise ValueError(f"Failed to send message
```

```
# Assign tools to their respective groups
```

```
tool_groups["Computation"]["tools"].append(query_
```

```
tool_groups["Automation"]["tools"].append(trigger
```

```

tool_groups["Communication"]["tools"].append(send)

# -----
# Embed Group and Tool Descriptions
# -----
# Embed group descriptions
group_names = []
group_embeddings = []
for group_name, group_info in tool_groups.items():
    group_names.append(group_name)

    group_embeddings.append(embeddings.embed_text(

# Create FAISS index for groups
group_embeddings_np = np.array(group_embeddings)
faiss.normalize_L2(group_embeddings_np)
group_index = faiss.IndexFlatL2(len(group_embeddings_np))
group_index.add(group_embeddings_np)

# Embed tool descriptions within each group
tool_indices = {} # Maps group name to its FAISS index
for group_name, group_info in tool_groups.items():
    tools = group_info["tools"]
    tool_descriptions = []
    tool_functions = []
    for tool_func in tools:
        description = tool_func.__doc__.strip().split("\n")
        tool_descriptions.append(description)
        tool_functions.append(tool_func)

```

```

    if tool_descriptions:
        tool_embeddings = embeddings.embed_texts(
            tool_descriptions)
        tool_embeddings_np = np.array(tool_embeddings)
        faiss.normalize_L2(tool_embeddings_np)
        tool_index = faiss.IndexFlatL2(len(tool_embeddings_np))
        tool_index.add(tool_embeddings_np)
        tool_indices[group_name] = {
            "index": tool_index,
            "functions": tool_functions,
            "embeddings": tool_embeddings_np
        }

# -----
# Hierarchical Skill Selection
# -----
def select_group(query: str, top_k: int = 1) -> list[str]:
    query_embedding = embeddings.embed_text(query)
    faiss.normalize_L2(query_embedding.reshape(1, D))
    I = group_index.search(query_embedding.reshape(1, D))
    selected_groups = [group_names[idx] for idx in I[0]]
    return selected_groups

def select_tool(query: str, group_name: str, top_k: int = 1) -> list[str]:
    tool_info = tool_indices[group_name]
    query_embedding = embeddings.embed_text(query)
    faiss.normalize_L2(query_embedding.reshape(1, D))
    I = tool_info["index"].search(query_embedding.reshape(1, D))
    selected_tools = [tool_info["functions"][idx] for idx in I[0]]
    return selected_tools

```



```

        return selected_tools

# Initialize the LLM with GPT-4 and set temperature
llm = ChatOpenAI(model_name="gpt-4", temperature=0.1)

selected_groups = select_group(user_query, top_k)
if not selected_groups:
    print("No relevant skill group found for the query")
    return

selected_group = selected_groups[0]
logging.info(f"Selected Group: {selected_group}")
print(f"Selected Skill Group: {selected_group}")

# Step 2: Select the most relevant tool within the group
selected_tools = select_tool(user_query, selected_group)

if not selected_tools:
    print("No relevant tool found within the selected group")
    return

selected_tool = selected_tools[0]
logging.info(f"Selected Tool: {selected_tool}")
print(f"Selected Tool: {selected_tool.__name__}")

# Prepare arguments based on the tool
args = {}

```

```
if selected_tool == query_wolfram_alpha:
    # Assume the entire query is the expression
    args["expression"] = user_query
elif selected_tool == trigger_zapier_webhook:
    # For demonstration, use placeholders
    args["zap_id"] = "123456" # Replace with actual zap_id
    args["payload"] = {"message": user_query}
elif selected_tool == send_slack_message:
    # For demonstration, use placeholders
    args["channel"] = "#general" # Replace with actual channel
    args["message"] = user_query
else:
    print("Selected tool is not recognized.")
    return

# Invoke the selected tool
try:
    tool_result = selected_tool.invoke(args)
    print(f"Tool '{selected_tool.__name__}' result: {tool_result}")
except ValueError as e:
    print(f"Error: {e}")
```

Machine Learned Skill Selection

Machine Learned Skill Selection employs machine learning techniques to automatically learn and select skills based on past

experiences and task feedback. Generic generative and embedding models are often larger, slower, and more expensive than is necessary for skill selection, so by training specific models on task-skill pairs, you can potentially reduce the cost and latency of this part of your agent-based solution. Both historical data and data samples generated by a foundation model can be used to train your skill selection model. Similarly, you could fine-tune a smaller model to improve the classification performance on your skill selection task. The key drawback is it introduces a new model that your team will need to maintain. Carefully consider the costs before choosing to proceed down this path, as it may require extensive training data and computational resources to achieve optimal performance.

Skill Execution

Parametrization is the process of defining and setting the parameters that will guide the execution of a skill in a language model. This process is crucial as it determines how the model interprets the task and tailors its response to meet the specific requirements. Parameters are defined by the skill definition as discussed in more detail in Chapter 3. The current state of the agent, including progress so far, is included as additional

context in the prompt window, and the foundation model is instructed to fill the parameters with appropriate data types to match the expected inputs for the function call. Additional context, such as the current time or the user's location, can be injected into the context window to provide additional guidance for functions that require this type of information. It is recommended to use a basic parser to validate that the inputs meet the basic criteria for the data types, and to instruct the foundation model to correct the pattern if it does not pass this check.

Once the parameters are set, the skill execution phase begins. This involves the actual execution of the skill. Some of these skills can easily be executed locally, while others will be executed remotely by API. During execution, the model might interact with various APIs, databases, or other tools to gather information, perform calculations, or execute actions that are necessary to complete the task. The integration of external data sources and tools can significantly enhance the utility and accuracy of the agent's outputs. Timeout and retry logic will need to be adjusted to the latency and performance requirements for the use case.

Skill Topologies

Today, the majority of chatbot systems rely on Single Skill Execution without planning. This makes sense: it is easier to implement, and has lower latency. If your team is developing its first agent-based system, or if that is sufficient to meet the needs for your scenario, then you can stop there after the following section, Single Skill Execution. For many cases, however, we want our agents to be able to perform complex tasks that require multiple skills. By providing an agent with a sufficient range of skills, you can then enable your agent to flexibly arrange those skills and apply them in correct order to solve a wider variety of problems. In traditional software engineering, the designers had to implement the exact control flow and order in which steps should be taken. Now, we can implement the skills, and define the skills topology in which the agent can operate, then allow the exact composition to be designed dynamically in response to the context and task at hand. This section considers this range of skill topologies and discusses their tradeoffs.

Single Skill Execution

We'll begin with tasks that require precisely one skill. In this case, planning consists of choosing the one skill most appropriate to address the task. Once the skill is selected, it must be correctly parameterized based on the skill definition.

The skill is then executed, and its output is used as an input when composing the final response for the user. This can be seen in Figure 4-1 below (TODO: draft). While this is a minimal definition of a plan, it is the foundation from which we will build more complex patterns.

Parallel Skill Execution

The first increase in complexity comes with skill parallelism. In some cases, it might be worth taking multiple actions on the input. For example, consider you need to look up a record for a patient. If your skillset includes multiple skills that access multiple sources of data, then it will be necessary to execute multiple actions to retrieve data from each of the sources. This increases the complexity of the problem because it is unclear how many skills need to be executed. A common approach is to retrieve a maximum number of skills that might be executed, say 5, using Semantic Skill Selection. Next, make a second call to a foundation model with each of these five skills, and ask it to select the five or fewer skills that are necessary to the problem, filtering down to the skills necessary for the task. Similarly, the foundation model can be called repeatedly with the additional context of which skills have already been selected until it chooses to add no fewer skills. Once selected, these skills are

independently parameterized and executed. After all skills have been completed, their results are passed to the foundation model to draft a final response for the user. Figure 4-2 (TODO) illustrates this pattern.

Chains

The next increase in complexity brings us to chains. Chains refer to sequences of actions that are executed one after another, with each action depending on the successful completion of the previous one. Planning chains involves determining the order in which actions should be performed to achieve a specific goal while ensuring that each action leads to the next without interruption. Chains are common in tasks that involve step-by-step processes or linear workflows.

The planning of chains requires careful consideration of the dependencies between actions, aiming to orchestrate a coherent flow of activity towards the desired outcome. It is highly recommended that a maximum length be set to the skill chains, as errors can compound down the length of the chain. So long as the task is not expected to fan out to multiple branching subtasks, chains provide an excellent tradeoff between adding planning for multiple skills with dependencies, while keeping the complexity relatively low.

Trees

In more complex scenarios, tasks may require branching sequences of actions, where the agent must choose between multiple possible paths at each decision point. Planning trees involve exploring different branches of action possibilities, evaluating the consequences of each choice, and selecting the most promising path towards the goal. Trees are useful for tasks with multiple options or alternative courses of action. This structure enables the natural expansion that is involved in certain tasks, especially when a prior skill returns multiple outputs that need to be considered.

By increasing the skill topology from a chain to a tree, the skill structure contains the state of the execution. Compared to the chain structure, the agent has more options to choose from. In addition to selecting and executing a skill from its current position, the agent can determine that the task has been completed, decide that it is unable to complete the task, or it can traverse to another leaf node on the tree, and proceed from there. This structure reduces the likelihood that subtasks are forgotten by the agent. In this structure, key parameters to tune for your use case include the maximum number of skills per execution and the maximum depth of the tree.

Graphs

Graphs represent interconnected networks of actions, where dependencies between tasks can be more complex and nonlinear. Graphs are an extension of trees, and while they enable the same expansion to multiple items as trees, they also enable topologies that consolidate multiple nodes together. This structure allows for an expressive representation that tracks the flow of information across multiple skill executions.

In addition to the tree structure, the graph structure adds a new action: consolidate. This new action enables the agent to connect the results from multiple previously completed skills. This is invaluable when especially complex reasoning is desirable, and the agent is expected to stitch together findings from multiple previous skills. While graphs are a more flexible and expressive structure, they are more complex to manage and traverse, and open up a new class of errors that the agent can make.

Choosing a Topology

Selecting the appropriate topology is crucial for effectively organizing and executing actions. Linear topologies, such as chains, are suitable for tasks with a sequential flow of actions

where each step leads directly to the next. This topology simplifies the planning process, as actions are executed in a straightforward order without branching or decision points.

Hierarchical topologies, such as trees, are useful for organizing actions into nested levels of abstraction. This topology allows for both high-level strategic planning and detailed, fine-grained control over individual actions. Hierarchical topologies are well-suited for tasks with multiple layers of complexity or tasks that can be decomposed into subtasks with distinct goals. For example, in project management, tasks can be organized hierarchically based on their dependencies and relationships.

Graph topologies offer the most flexibility and expressiveness in representing complex relationships between actions. In this type of graph, actions are interconnected, allowing for nonlinear dependencies and dynamic decision-making. This topology is ideal for tasks with interconnected components, where actions can affect each other in unpredictable ways. Graph topologies are commonly used in tasks such as resource allocation, where the allocation of resources depends on multiple factors and constraints.

Choosing the appropriate topology depends on factors such as the complexity of the task, the degree of interdependence

between actions, and the level of flexibility required in the planning process. By selecting the right topology, autonomous agents can effectively organize their plans, navigate complex environments, and achieve their goals with efficiency and adaptability. As a principle, start simple, and only add complexity to address specific needs in your use case.

Planning

Now that we have discussed a range of skill topologies, it's time to consider how to use them. We'll begin with the simplest approach, then move through several more complex approaches. Note that any skill topology can be used with any planning approach.

Iterative Planning

We begin with a discussion of the simplest type of plan, which is iterative planning. In this approach, the agent chooses an action and executes it. You can think of this as the “unplanned” or “greedy” approach to planning. This has multiple advantages, including simplicity, lower latency, and easier maintainability. This approach can handle many use cases and

is the recommended starting point. For tasks that require a small number of skills, this is probably sufficient.

Zero-Shot Planning

For more complex tasks, it will be necessary to draft a plan before beginning. The more complex the task, the more subtasks it will require. By simply executing one action at a time in a greedy approach, agents will sometimes get caught in loops and fail to make progress toward completion. Taking the time to create a plan, and then to choose actions toward that plan, can increase the overall performance on these tasks that require multiple steps.

The simplest place to start with planning is zero-shot planning, which refers to the ability of an agent to generate plans for tasks it has never encountered before, based solely on its understanding of the task and its environment. This approach requires the agent to possess a robust representation of the task space, including possible actions, their effects, and the relationships between different components. By leveraging this knowledge, the agent can generate plans on the fly, even for novel tasks, without requiring explicit training data or pre-defined solutions. Zero-shot planning is particularly useful in

dynamic environments where tasks may vary over time or where the agent needs to adapt quickly to new challenges.

In-Context Learning with Hand-Crafted Examples

This brings us to hand-crafted examples, where developers design plans for specific tasks manually, based on domain expertise or predefined rules. In this approach, human designers analyze the task requirements, identify relevant actions, and determine the sequence in which these actions should be executed to achieve the desired outcome. Hand-crafted plans are often used for tasks with well-defined structures or known solutions, where human intuition or expertise can guide the planning process effectively. While hand-crafted plans offer precision and reliability, they may lack flexibility and scalability, as they rely heavily on human intervention and may not generalize well to new scenarios. By scoping core, high-value scenarios, developers can provide clear examples. By embedding and indexing these into a few-shot database, the relevant examples can be pulled to guide future plans based on semantically-similar examples.

Plan Adaptation

In scenarios where subsequent skills depend on the output of previous skills, the ability to adapt a plan will be necessary. Plan adaptation enables agents to respond to new information, unexpected events, or deviations from the original plan, ensuring continued progress towards their goals. The leading approach to this type of plan reaction, Reason-Act, or ReAct¹ for short, provides a simple but effective framework for approaching the task of plan adaptation. As the name suggests, the agent alternates between reasoning steps and acting steps. In the reasoning step, the agent is asked to consider what it needs to do to answer the question. The foundation model is then invoked to choose the action, one of which is to complete the execution flow. This enables the agent to repeatedly take actions, such as looking up data, and checking to see if the results from the search are sufficient to meet the task. If not, the agent can choose to continue to search or take additional actions. A further extension of this work is PlanReAct, which adds an additional self-think flow, which includes Chain of Thought reasoning. This combines planning, reasoning, and acting into a cohesive process.

Summary

The success of agents relies heavily on the approach to orchestration, making it important for organizations interested in building agentic systems to invest time and energy into designing the appropriate planning strategy for the use case.

Here are some best practices for designing a planning system:

- Carefully consider the requirements for latency and accuracy for your system, as there is a clear tradeoff between these two factors.
- Determine the typical number of actions required for your scenario's use case. The greater this number, the more complex an approach to planning you are likely to need.
- Assess how much the plan needs to change based on the results from prior actions. If significant adaptation is necessary, consider a technique that allows for incremental plan adjustments.
- Design a representative set of test cases to evaluate different planning approaches and identify the best fit for your use case.
- Choose the simplest planning approach that will meet your use case requirements.

With an orchestration approach that will work well for your scenario, we'll now move onto the next part of the workflow:

memory. It is worth noting that it is worth starting small with well-designed scenarios and simpler approaches to orchestration, and to then gradually move up the scale of complexity as necessary based on the use case.

· Shunyu Yao et al.: ReAct: Synergizing Reasoning and Acting in Language Models. Published as a conference paper at ICLR 2023. <https://arxiv.org/pdf/2210.03629>

Chapter 6. Knowledge and Memory

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Now that your agent has skills and orchestration, it is more than capable of taking actions to do real work. In most cases, though, you will want your agents to know more about your specific problem area than the foundation model would alone, and be able to store knowledge and information over time. You might even want your agent to use this memory to accomplish tasks more effectively. In this chapter, we’ll discuss how

memory can be added to agentic systems to add external knowledge, maintain state across sessions, and perform complex tasks more effectively. Memory can also be an excellent way to get your AI-powered application to better meet the needs of the specific context. Let's dive in.

Memory plays multiple roles in applications with AI Agents. In this context, we are not referring to computer system memory, such as RAM, but to information that is dynamically injected into the prompt to complement the parametric memory of the model. It maintains the state of the interaction, the previous tasks performed and their previous results, and is critical for learning. These approaches are useful for the interactions between humans and LLM-powered applications, the interactions between AI agents, and for domain-specific or organization-specific information. We'll discuss specific architectures soon, but first, we'll focus on the fundamentals of memory for agentic systems.

Foundational Approaches to Memory

We begin by discussing the simplest approaches to memory: relying on a rolling context window for the foundation model,

and keyword-based memory. Despite their simplicity, they are more than sufficient for a wide range of use cases.

Managing Context Windows

We start with the simplest approach to memory: relying on the context window. The context window is a critical resource for developers to use effectively. We want to provide the foundation model with all the information it needs to complex the task, but no more. The context window is all of the information that is provided to the foundation model when the model is called. In the simplest approach, in addition to the current question, all of the remaining context window that is available is filled with the previous interactions in the current session. When that window fills up, only the most recent interactions are included. In some circumstances, we will have more information to provide than we can fit into the context window. When this happens, we need to be careful with how we allocate our limited budget of tokens.

For simple use cases, you can use a rolling context window. In this case, as the interaction with the foundation model progresses, the full interaction is passed into the context window. At a certain point, the context window fills up, the oldest parts of the context are ejected, and replaced with the

most recent context, in a first-in, first-out fashion. This is easy to implement, low in complexity, and will work for many use cases. The primary drawback to this approach is information will be lost, regardless of how relevant or important it is, as soon as enough interaction has occurred to eject it from the current context. With large prompts or verbose foundation model responses, this can happen quickly. Foundation models can also miss important information in large prompts, so highlighting the most relevant context, and placing it close to the end of the prompt can increase the likelihood that it will be used. This standard approach to memory can be incorporated into our LangGraph agent as follows:

```
from typing import Annotated
from typing_extensions import TypedDict

from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState

llm = ChatOpenAI(model="gpt-4o")

def call_model(state: MessagesState):
    response = llm.invoke(state["messages"])
```

```
    return {"messages": response}

builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_edge(START, "call_model")
graph = builder.compile()

# Fails to maintain state across the conversation
input_message = {"type": "user", "content": "hi!"}
for chunk in graph.stream({"messages": [input_message]}):
    chunk["messages"][-1].pretty_print()

input_message = {"type": "user", "content": "what?"}
for chunk in graph.stream({"messages": [input_message]}):
    chunk["messages"][-1].pretty_print()
```

Keyword-Based Memory

The simplest way to begin extracting and organizing larger sessions as they go is through keyword extraction. One task that foundation models have performed very well at is keyword extraction. Given an input block of text, a foundation model can identify the key words and phrases in it. As the interaction occurs, send each response to a foundation model with a keyword extraction prompt. Maintain storage for each

response, as well as a map from the keywords to their original document. When new prompts are received, keywords are extracted from it, and the lookup is consulted. If there are matches, the previous occurrences are included in the prompt. This simple approach can preserve the broader context of interactions that address specific topics over time. An advantage to this approach is its simplicity, while still enabling the preservation of information over time. Some important parameters to choose for this type of memory are

Integrating Keyword-Based Memory with Rolling Context Windows

While both rolling context windows and keyword-based memory have their own advantages, we can also take a hybrid approach that splits the context window between keyword retrieval and rolling context window. Include the j most recent occurrences from the keyword retrieval, then fill the rest of the context window with the k most recent interactions that fit into the context window. But what if we want to include relevant interactions, even if they don't have a keyword match, or occurred far enough back that it wouldn't be in the context window? We'll address that in the next section.

Semantic Memory and Vector Stores

Semantic memory, a type of long-term memory that involves the storage and retrieval of general knowledge, concepts and past experiences, plays a critical role in enhancing the cognitive capabilities of these systems. This allows information and past experiences to be stored and then efficiently retrieved when it is needed to improve performance later on. The leading way to do this is by using vector databases, which enable rapid indexing and retrieval at large scale, enabling agentic systems to understand and respond to queries with greater depth and relevance.

Introduction to Semantic Search

Unlike traditional keyword-based search, semantic search aims to understand the context and intent behind a query, leading to more accurate and meaningful retrieval results. At its core, semantic search focuses on the meaning of words and phrases rather than their exact match. It leverages machine learning techniques to interpret the context, synonyms, and relationships between words. This allows the retrieval system to comprehend the intention and deliver results that are

contextually relevant, even if they don't contain the exact search terms.

The foundation for these approaches are embeddings. These are vector representations of words that capture their meanings based on their usage in large text corpora. By projecting large bodies of text into a dense numeric representation, we can create rich representations that have proven to be very useful for storage and retrieval. Popular models like Word2Vec, GloVe, and BERT have revolutionized how machines understand language by placing semantically similar words closer together in a high-dimensional space. Large language models have further improved the performance of these embedding models across a wide range of types of text by increasing the size of the embedding model and the quantity and variety of data on which they are trained. Semantic search has proven to be an invaluable technique to improve the performance of memory within agentic systems, particularly in retrieving semantically relevant information across documents that do not share exact keywords.

Implementing Semantic Memory with Vector Stores

We begin by generating semantic embeddings for the concepts and knowledge to be stored. These embeddings are typically produced by large language models or other NLP techniques that encode textual information into dense vector representations. These vector representations, or embeddings, capture the semantic properties and relationships of data points in a continuous vector space. For example, a sentence describing a historical event can be converted into a vector that captures its semantic meaning. Once we have this vector representation, we need a place to efficiently store it. That place is a vector database, which are designed specifically to efficiently handle high-dimensional vector representations of data.

Vector stores, such as vectordb, FAISS (Facebook AI Similarity Search) or Annoy (Approximate Nearest Neighbors Oh Yeah), are optimized for storing and searching high-dimensional vectors. These stores allow for fast similarity searches, enabling the retrieval of embeddings that are semantically similar to a given query.

When an agent receives a query or needs to retrieve information, it can use the vector store to perform similarity searches based on the query's embedding. By finding and retrieving the most relevant embeddings from the vector store,

the agent can access the stored semantic memory and provide informed, contextually appropriate responses. These lookups can be performed quickly, providing an efficient way to rapidly search over large volumes of information to improve the quality of actions and responses. This can be implemented as follows:

```
from typing import Annotated
from typing_extensions import TypedDict

from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState

llm = ChatOpenAI(model="gpt-4o")

def call_model(state: MessagesState):
    response = llm.invoke(state["messages"])
    return {"messages": response}

from vectordb import Memory

memory = Memory(chunking_strategy={'mode': 'sliding'})

text = """
Machine learning is a method of data analysis that
uses statistical models to identify patterns in data.

It is a branch of artificial intelligence based on
```

identify patterns and make decisions with minimal

Machine learning algorithms are trained on data sets.
Once an algorithm is trained, it can be used to recognize patterns.
"""

```
metadata = {"title": "Introduction to Machine Learning"}
```

```
memory.save(text, metadata)
```

```
text2 = """
```

Artificial intelligence (AI) is the simulation of human intelligence processes that are programmed to think like humans and mimic their actions.

The term may also be applied to any machine that mimics or automates human intelligence such as learning and problem-solving.

AI research has been highly successful in developing machines that can perform tasks that were previously thought to require human intelligence.
"""

```
metadata2 = {"title": "Introduction to Artificial Intelligence"}
```

```
memory.save(text2, metadata2)
```

```
query = "What is the relationship between AI and machine learning?"
```

```
results = memory.search(query, top_n=3)
```

```
builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_edge(START, "call_model")
graph = builder.compile()

input_message = {"type": "user", "content": "hi!"}
for chunk in graph.stream({"messages": [input_message]}):
    chunk["messages"][-1].pretty_print()

print(results)
```

Retrieval Augmented Generation

Incorporating memory into agentic systems not only involves storing and managing knowledge but also enhancing the system's ability to generate contextually relevant and accurate responses. Retrieval Augmented Generation (RAG) is a powerful technique that combines the strengths of retrieval-based methods and generative models to achieve this goal. By integrating retrieval mechanisms with large language models, RAG allows agentic systems to generate more informed and contextually enriched responses, improving their performance in a wide range of applications.

During retrieval, the system searches a large corpus of documents or a vector store of embeddings to find pieces of information that are relevant to the given query or context. This phase relies on efficient retrieval mechanisms to quickly identify and extract pertinent information.

During generation, the retrieved information is then fed into a generative foundation model, which uses this context to produce a coherent and contextually appropriate response. The generative model synthesizes the retrieved data with its own learned knowledge, enhancing the relevance and accuracy of the generated text.

Retrieval Augmented Generation represents a powerful approach for enhancing the capabilities of agentic systems by combining retrieval-based methods with generative models. By leveraging external knowledge and integrating it into the generation process, RAG enables the creation of more informed, accurate, and contextually relevant responses. As technology continues to evolve, RAG will play a crucial role in advancing the performance and versatility of LLM-powered applications across various domains. This is especially valuable for incorporating domain- or company- specific information or policies to influence the output.

Semantic Experience Memory

While incorporating an external knowledge base with a semantic store is an effective way to incorporate external knowledge into our agent, our agent will start every session from a blank slate, and the context of long-running or complex tasks will gradually drop out of the context window. Both of these issues can be addressed by semantic experience memory.

With each user input, the text is turned into a vector representation using an embedding model. The embedding is then used as the query in a vector search across all of the previous interactions in the memory store. Part of the context window is reserved for the best matches from the semantic experience memory, then the rest of the space is allocated to the system message, latest user input, and the most recent interactions. Semantic experience memory allows agentic systems to not only draw upon a broad base of knowledge but also tailor their responses and actions based on accumulated experience, leading to more adaptive and personalized behavior.

Graph RAG

We now turn to an advanced version of RAG that is more complex to incorporate into your solution, but that is capable of correctly handling a wider variety of questions. Graph Retrieval Augmented Generation (Graph RAG) is an advanced extension of the Retrieval Augmented Generation (RAG) model, incorporating graph-based data structures to enhance the retrieval process. By utilizing graphs, Graph RAG can manage and utilize complex interrelationships and dependencies between pieces of information, significantly enhancing the richness and accuracy of the generated content. This chapter will delve into how Graph RAG works, its implementation, and its applications in various domains.

Using Knowledge Graphs

Graph RAG extends the basic RAG framework by integrating a graph-based retrieval system. This system leverages the power of graph databases or knowledge graphs to store and query interconnected data. In Graph RAG, the retrieval phase doesn't just pull relevant documents or snippets; it analyzes and retrieves nodes and edges from a graph that represent complex relationships and contexts within the data. GraphRAG consists of the following three components:

Knowledge Graph

This component stores data in a graph format, where entities (nodes) and their relationships (edges) are explicitly defined. Graph databases are highly efficient at managing connected data and supporting complex queries that involve multiple hops or relationships.

Retrieval System

The retrieval system in Graph RAG is designed to query the graph database efficiently, extracting subgraphs or clusters of nodes that are most relevant to the input query or context.

Generative Model

Once relevant data is retrieved in the form of a graph, the generative model synthesizes this information to create coherent and contextually rich responses.

Graph Retrieval-Augmented Generation represents a significant leap forward in the capabilities of agentic systems, offering sophisticated tools to handle and generate responses based on complex interconnected data. As this technology evolves, it promises to open new frontiers in AI applications, making systems smarter, more context-aware, and capable of handling increasingly complex tasks. Using knowledge graphs in Graph RAG systems transforms the way information is retrieved and

utilized for generation, enabling more intelligent, contextual, and accurate responses across various applications. We will not cover the details of the algorithm here, but multiple open-source implementations of GraphRAG are now available, and setting them up on your dataset is easier to do. If you have a large set of data you need to reason over, and standard chunking with a vector retrieval is running into limitations, GraphRAG is a more expensive and complex approach that frequently produces better results in practice.

Building Knowledge Graphs

Knowledge graphs are fundamental in providing structured and semantically rich information that enhances the capabilities of intelligent systems, including Graph Retrieval-Augmented Generation (Graph RAG) systems. Building an effective knowledge graph involves a series of steps, from data collection and processing to integration and maintenance. This section will cover the methodology for constructing knowledge graphs that can significantly impact the performance of Graph RAG systems. This process consists of several steps:

1. **Data Collection:** The first step in building a knowledge graph is gathering the necessary data. This data can come from various sources, including databases, text documents,

websites, and even user-generated content. It's crucial to ensure the diversity and quality of sources to cover a broad spectrum of knowledge. For an organization, this may consist of a set of core policies or documents that contain core information to influence the agent.

2. **Data Preprocessing:** Once data is collected, it needs to be cleaned and preprocessed. This step involves removing irrelevant or redundant information, correcting errors, and standardizing data formats. Preprocessing is vital for reducing noise in the data and improving the accuracy of the subsequent entity extraction process.

3. **Entity Recognition and Extraction:** This process involves identifying key elements (entities) from the data that will serve as nodes in the knowledge graph. Common entities include people, places, organizations, and concepts. Techniques such as Named Entity Recognition (NER) are typically used, which may involve machine learning models trained on large datasets to recognize and categorize entities accurately.

4. **Relationship Extraction:** After identifying entities, the next step is to determine the relationships between them. This involves parsing data to extract predicates that connect entities, forming the edges of the graph. Relationship extraction can be challenging, especially in unstructured

data, though foundation models have shown improving efficacy over time.

5. **Ontology Design:** An ontology defines the categories and relationships within the knowledge graph, serving as its backbone. Designing an ontology involves defining a schema that encapsulates the types of entities and the possible types of relationships between them. This schema helps in organizing the knowledge graph systematically and supports more effective querying and data retrieval.
6. **Graph Population:** With the ontology in place, the next step is to populate the graph with the extracted entities and their relationships. This involves creating nodes and edges in the graph database according to the ontology's structure. Databases like Neo4j, OrientDB, or Amazon Neptune can be used to manage these data structures efficiently.
7. **Integration and Validation:** Once the graph is populated, it must be integrated with existing systems and validated to ensure accuracy and utility. This can involve linking data from other databases, resolving entity duplication (entity resolution), and verifying that the graph accurately represents the knowledge domain. Validation might involve user testing or automated checks to ensure the integrity and usability of the graph.

8. Maintenance and Updates: A knowledge graph is not a static entity; it needs regular updates and maintenance to stay relevant. This involves adding new data, updating existing information, and refining the ontology as new types of entities or relationships are identified. Automation and machine learning models can be instrumental in maintaining and updating the knowledge graph efficiently.

Building a knowledge graph is a complex but rewarding endeavor that can significantly enhance the capabilities of Graph RAG systems. By structuring information into an interconnected web of knowledge, these graphs enable intelligent systems to perform sophisticated reasoning, provide contextual responses, and deliver personalized services. These structures make it easy to discover underlying relationships in the data. For instance, it is now possible to search for elements on the graph, then retrieve all the elements that are one or more links away from that node. This provides an efficient way to retrieve relevant context for addressing a task. As AI technology progresses, the methodologies for building, integrating, and maintaining knowledge graphs will continue to evolve, further enhancing their utility in various domains.

Promise and Peril of Dynamic Knowledge Graphs

Dynamic knowledge graphs represent an evolutionary leap in managing and utilizing knowledge in real-time applications. These graphs are continuously updated with new information, adapting to changes in knowledge and context, which can significantly enhance Graph Retrieval-Augmented Generation (Graph RAG) systems. However, the dynamic nature of these graphs also introduces specific challenges that need careful consideration. This section explores the potential benefits and risks associated with dynamic knowledge graphs. As the developer, it is important to apply careful consideration to what should be included in the knowledge graph. Official documentation, publicly available content, and past interactions could all be considered for knowledge graph construction, but it is important to remember that the graph construction process is imperfect, and poor quality or security vulnerabilities in the underlying data can undermine the system.

Dynamic Real-time information processing is greatly enhanced by dynamic knowledge graphs, which can integrate real-time data. This capability is particularly useful in environments

where information is constantly changing, such as news, social media, and live monitoring systems. By ensuring that the system's responses are always based on the most current and relevant information, dynamic knowledge graphs provide a significant advantage.

Adaptive learning is another key feature of dynamic knowledge graphs. They continuously update themselves, learning from new data without the need for periodic retraining or manual updates. This adaptability is crucial for applications in fast-evolving fields like medicine, technology, and finance, where staying updated with the latest knowledge is critical. This helps organizations make informed decisions quickly, which is invaluable in scenarios where decisions have significant implications and depend heavily on the latest information. Knowledge graphs also provide critical information in a structured format that can be effectively operated across and reasoned over, and provide far greater flexibility than vector stores, and are especially valuable for understanding the rich context of an entity. Unfortunately, these benefits come with some important drawbacks:

Complexity in Maintenance

Maintaining the accuracy and reliability of a dynamic knowledge graph is significantly more challenging than

managing a static one. The continuous influx of new data can introduce errors and inconsistencies, which may propagate through the graph if not identified and corrected promptly.

Resource Intensity

The processes of updating, validating, and maintaining dynamic knowledge graphs require substantial computational resources. These processes can become resource-intensive, especially as the size and complexity of the graph grow, potentially limiting scalability.

Security and Privacy Concerns

Dynamic knowledge graphs that incorporate user data or sensitive information must be managed with strict adherence to security and privacy standards. The real-time aspect of these graphs can complicate compliance with data protection regulations, as any oversight might lead to significant breaches.

Dependency and Overreliance

There is a risk of overreliance on dynamic knowledge graphs for decision-making, potentially leading to a lack of critical oversight. Decisions driven solely by automated

insights from a graph might overlook external factors that the graph does not capture.

To harness the benefits of dynamic knowledge graphs while mitigating their risks, several strategies can be employed. Implementing robust validation mechanisms with automated tools and processes is essential for continuously ensuring the accuracy and reliability of data within the graph. Designing a scalable architecture using technologies such as distributed databases and cloud computing helps manage the computational demands of dynamic graphs. Strong security measures, including encryption, access controls, and anonymization techniques, are crucial to ensure that all data inputs and integrations comply with current security and privacy regulations. Additionally, maintaining human oversight in critical decision-making processes mitigates the risks of errors and overreliance on automated systems.

Dynamic knowledge graphs offer substantial promise for enhancing the intelligence and responsiveness of Graph RAG systems, providing significant benefits across various applications. However, the complexities and risks associated with their dynamic nature necessitate careful management and oversight. By addressing these challenges proactively, the potential of dynamic knowledge graphs can be fully realized,

driving forward the capabilities of intelligent systems in an ever-evolving digital landscape.

Working Memory

Working memory plays a crucial role similar to short-term memory in human cognition. It enables agents to hold and manipulate information temporarily for the execution of complex tasks and interactions. This chapter explores the concept of working memory in agentic systems, discussing its functions, implementations, and the challenges associated with integrating it into large language models and other intelligent systems.

Working memory in agentic systems refers to the temporary storage and processing space used to hold immediate data and contextual information needed for task execution. This type of memory is dynamic, rapidly updateable, and crucial for tasks that require comprehension, reasoning, and immediate response, such as conversational understanding, problem-solving, and dynamic decision-making.

Whiteboards

Just as for humans, it is not always possible to solve problems directly. Sometimes, we need to work through a problem step by step, and save our notes as we go. This is exactly what a “whiteboard” is for a foundation model - it serves as a dynamic form of working memory.¹ Conceptually similar to the physical whiteboards used in brainstorming sessions, digital whiteboards in agentic systems provide a flexible, interactive canvas where temporary data can be stored, manipulated, and used collaboratively. This section explores how whiteboards function as an integral part of working memory in intelligent agents, their implementations, and their applications in enhancing system performance.

Whiteboards in agentic systems function as a short-term storage, where multiple streams of data and processes can be dynamically displayed and managed. Agents can interact with and manipulate the data on whiteboards, allowing for dynamic changes based on new inputs or decisions. These systems have also been called scratchpads². This feature is crucial for tasks that require real-time adjustments or updates. It also allows for important information to be extracted and made readily available for reference, which can be incredibly useful when operating over large amounts of data. These have been shown to be especially useful for complex and multi-step

computations, as well as predicting outputs from arbitrary programs.

Note Taking

While closely related to whiteboards, note taking is a distinct approach that can also improve performance for complex tasks. With this technique, the foundation model is prompted to specifically inject notes on the input context without trying to answer the question.³ This mimics the way that we might fill in the margins or summarize a paragraph or section. This note-taking is performed before the question is presented, and then interleaves these notes with the original context when attempting to address the current task. Experiments show good results on multiple reasoning and evaluation tasks, with potential for adaptation to a wider range of scenarios. As we can see in the figure below, in a traditional, vanilla approach, the model is provided with the context and a question, and produces an answer. In chain of thought, it has time to reason about the problem, and only subsequently generate its answer to the question. With the self-note approach, the model generates notes on multiple parts of the context, and then generates a note on the question, before finally moving to generate the final answer.

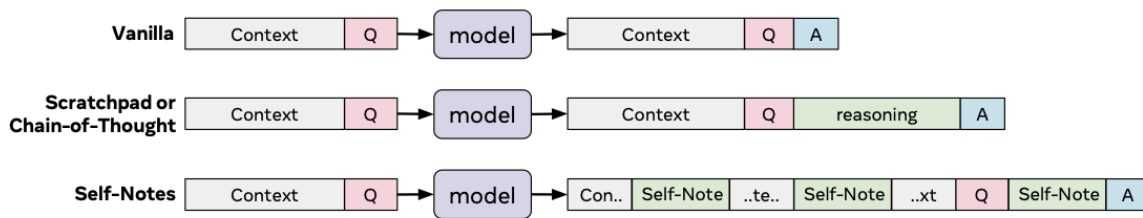


Figure 6-1. Taking Notes on the Context and Question to Improving Answer Quality
(Source: <https://arxiv.org/pdf/2305.00833>)

Conclusion

Memory is critical to the successful operation of agentic systems, and while the standard approach of relying on the context window of recent interactions is sufficient for many use cases, more challenging scenarios can benefit substantially from the investment into a more robust approach. We have explored several approaches here, including semantic memory, GraphRAG, and working memory.

This chapter on memory in agentic systems has delved into various aspects of how memory can be structured and utilized to enhance the capabilities of intelligent agents. From the basic concepts of managing context windows, through the advanced applications of semantic memory and vector stores, to the innovative practices of dynamic knowledge graphs and working memory, we have explored a comprehensive range of

techniques and technologies that play crucial roles in the development of agentic systems.

Memory systems in agentic applications are not just about storing data but about transforming how agents interact with their environment and end-users. By continually improving these systems, we can create more intelligent, responsive, and capable agents that can perform a wide range of tasks more effectively. In the next chapter, we will explore how agents can learn from experience to improve automatically over time.

- Whiteboard of Thought: Thinking Step-by-Step Across Modalities, <https://arxiv.org/pdf/2406.14562>
- Show Your Work: Scratchpads for Intermediate Computation with Language Models, <https://arxiv.org/pdf/2112.00114>
- Learning to Reason with Self-Notes, <https://arxiv.org/abs/2305.00833>

Chapter 7. Learning in Agentic Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the seventh chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

This chapter covers different techniques to approach and integrate learning into agent systems. Adding the capability for agents to learn and improve over time is an incredibly useful addition, but is not necessary when designing agents.

Implementing learning capabilities takes additional design, evaluation, and monitoring, which may or may not be worth

the investment depending on the application. By learning, we mean improving the performance of the agentic system through interaction with the environment. This process allows agents to adapt to changing conditions, refine their strategies, and enhance their overall effectiveness.

Non-parametric learning refers to techniques to change and improve performance automatically without changing the parameters of the models involved. In contrast, parametric learning refers to techniques in which we specifically train or fine-tune the parameters of the foundation model. We will start by looking at nonparametric learning, then we will move on to parametric learning techniques.

Nonparametric Learning

Multiple techniques exist to do this, and we will explore several of the most common and useful approaches.

Non-Parametric Exemplar Learning

The simplest of these techniques is exemplar learning. In this approach, as the agent performs its task, it is provided with a measure of quality, and those examples are used to improve future performance. These examples are used as few-shot

examples for in-context learning. In the simplest version, fixed few shot examples, these examples are hard-coded into the prompt and do not change (the left side of [Figure 7-1](#)).



Figure 7-1. Caption to come

If we have more examples, we can continue adding them into the prompt, but that eventually comes with increases in cost and latency. Also, not all examples might be useful for all inputs. A common way to address this is to dynamically select the most relevant examples to include in the prompt (see on the right side of the figure). These experiences as examples are then stored in a way that makes them accessible for future reference. This typically involves building a memory bank where details of each interaction, such as the context, actions taken, outcomes, and any feedback received, are stored. This database acts much like human memory, where past experiences shape understanding and guide future actions. Each experience provides a data point that the agent can

reference to make better decisions when encountering similar situations. This method enables agents to build a repository of knowledge that can be drawn upon to improve performance.

The agent retrieves information from its database of past cases to solve new problems. Each stored case consists of a problem description, a solution that was applied, and the outcome of that solution. When faced with a new situation, the agent searches its memory to find similar past cases, analyzes the solutions that were applied, and adapts them if necessary to fit the new circumstances. This method allows for high flexibility as the agent can modify its approach based on what has or has not worked in the past, thus continually refining its problem-solving strategies.

When successful examples are saved in persistent storage, then retrieved and provided as examples in the prompt, performance increases significantly on a range of tasks. This is a well-established finding and has been confirmed across a variety of domains¹. In practice, this provides us with a simple, transparent, and lightweight way to rapidly improve the agent performance on given tasks. As the number of successful examples increases, it then becomes wise to retrieve the most relevant successful examples by type, text retrieval, or semantic retrieval. Note that this technique can be applied on the agentic

task execution as a whole, or can be performed independently on subsets of the task.

Reflexion

The technique known as Reflexion takes this a step farther². In this approach, agents adapt based on linguistic feedback. Specifically, they take actions, receive feedback, and maintain their own text in which they regularly reflect on their performance in a memory buffer to improve their performance on subsequent trials. It can accommodate both numerical and free-form feedback, and has shown improvements across coding tasks, sequential decision making, and language reasoning. You can see how this works in [Figure 7-2](#).



Figure 7-2. Caption to come

Despite the significant improvement that Reflexion can add to agents, this approach can be implemented with just a few lines

of code:

```
from typing import Annotated
from typing_extensions import TypedDict
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState
from langchain_core.messages import HumanMessage

llm = ChatOpenAI(model="gpt-4o")

reflections = []

def call_model(state: MessagesState):
    response = llm.invoke(state["messages"])
    return {"messages": response}

reflexion_prompt = f"""You will be given the history of a task that was
placed in an environment and given a task to complete. You will be given the
completing the task. Do not summarize your environment, just list the state.
the strategy and path you took to attempt to complete the task.
Devise a concise, new plan of action that accounts for the feedback.
to specific actions that you should have taken. If you have a new plan,
then devise a plan to achieve C with environment. If you do not have a new plan,
later when you are solving the same task. Give your response in the following format:
[Reflection]
[Plan]
[Action]

Instruction:
i am looking for dairy free and apple variety pack
[Search]
```

Action: search[dairy free and apple variety pack]

Observation:

[Back to Search]

Page 1 (Total results: 50)

[Next >]

[B07HRFJWP8]

Enjoy Life Foods Soft Baked Ovals, Breakfast Bars

\$100.0

[B01KMHY5PG]

Enjoy Life Soft Baked Chewy Bars, Variety Pack, 1

\$21.49

[B008D2X8C4]

Enjoy Life Lentil Chips Variety Pack, Dairy Free

\$100.0

Action: think[The first two items are not what I

Observation: OK.

Action: click[B008D2X8C4]

Observation:

[Back to Search]

[< Prev]

flavor name [dill and sour cream][garlic & parmes

size [0.8 ounce (pack of 24)][4 ounce (pack of 12

Price: \$100.0

Rating: N.A.

[Description]

[Features]

[Features]

[Reviews]

[Buy Now]

Action: think[The item has options 'variety pack'
Observation: OK.

Action: click[variety pack]
Observation: You have clicked variety pack.

Action: click[0.8 ounce (pack of 24)]
Observation: You have clicked 0.8 ounce (pack of

Action: click[Buy Now]

STATUS: FAIL

Plan:

"""

```
def update_memory(trial_log_path: str, env_configs: List[str]):
    """Updates the given env_config with the app
    with open(trial_log_path, 'r') as f:
        full_log: str = f.read()

    env_logs: List[str] = full_log.split('#####\n')
    assert len(env_logs) == len(env_configs), pr
    for i, env in enumerate(env_configs):
```

```


    for i, env in enumerate(env_configs):

        # if unsolved, get reflection and update
        if not env['is_success'] and not env['skipped']:
            if len(env['memory']) > 3:
                memory: List[str] = env['memory']
            else:
                memory: List[str] = env['memory']
            reflection_query: str = _generate_reflection_query(memory)
            reflection: str = get_completion(reflection_query)
            env_configs[i]['memory'] += [reflection]

builder = StateGraph(MessagesState)
builder.add_node("reflexion", call_model)
builder.add_edge(START, "reflexion")
graph = builder.compile()

result = graph.invoke(
    {
        "messages": [
            HumanMessage(
                content=reflexion_prompt
            )
        ]
    }
)
reflections.append(result)
print(result)

```



As we build up these reflections over time, we then inject them into the prompt for subsequent tasks to further improve the agent's performance.

Experiential Learning

Experiential learning takes nonparametric learning a step further³. In this approach, the agent still gathers its experiences into a datastore, but now it applies a new step of aggregating insights across those experiences to improve its future policy. This is especially valuable for reflecting back on past failures, and attempting to develop new techniques to improve performance in similar situations in the future. As the agent extracts insights from its experience bank, it maintains this list of insights over time, and dynamically modifies these insights, promoting the most valuable insights, downvoting the least useful ones, and revising insights based on new experiences.

This work builds on Reflexion by adding a process for cross-task learning. This allows the agent to improve its performance when it moves across different tasks and helps identify good practices that can transfer. In this approach, ExpeL maintains a list of insights that are extracted from past experiences. Over

time, new insights can be added, and existing insights can be edited, upvoted, downvoted, or removed, as can be seen in [Figure 7-3](#).



Figure 7-3. Caption to come

This process begins with a simple effort of asking the foundation model to reflect on the observation returned from the environment with the goal of identifying insights that can lead to better performance on the task in the future.

```
from typing import Annotated
from typing_extensions import TypedDict
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState
from langchain_core.messages import HumanMessage

# Initialize the LLM
llm = ChatOpenAI(model="gpt-4o")

# Function to call the LLM
def call_model(state: MessagesState):
```



```

        response = llm.invoke(state["messages"])
        return {"messages": response}

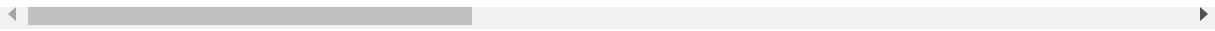
class InsightAgent:
    def __init__(self):
        self.insights = []
        self.promoted_insights = []
        self.demoted_insights = []
        self.reflections = []

    def generate_insight(self, observation):
        # Use the LLM to generate an insight based on the observation
        messages = [HumanMessage(content=f"Generate an insight based on the observation: {observation}")]

        # Build the state graph
        builder = StateGraph(MessagesState)
        builder.add_node("generate_insight", call_generate_insight)
        builder.add_edge(START, "generate_insight")
        graph = builder.compile()

        # Invoke the graph with the messages
        result = graph.invoke({"messages": messages})
        # Extract the generated insight
        generated_insight = result["messages"][-1].content
        self.insights.append(generated_insight)
        print(f"Generated: {generated_insight}")
        return generated_insight

```



This may work well when we have a small number of examples to learn from, but what if we have many examples to learn from? This technique offers a simple but effective way to manage this: the insights generated are regularly reevaluated and adjusted in relative importance to the other rules. For example, a sample prompt to reflect on previous actions to generate new rules to improve performance on future trials could be:

“By examining and contrasting to the successful trial, and the list of existing rules, you can perform the following operations: add, edit, remove, or agree so that the new list of rules is GENERAL and HIGH LEVEL critiques of the failed trial or proposed way of Thought so they can be used to avoid similar failures when encountered with different questions in the future. Have an emphasis on critiquing how to perform better Thought and Action” (ExpeL).

These learned rules are then regularly reevaluated and adjusted in importance relative to the other rules learned from experience. The methodology for evaluating and improving the existing rules is as follows:

“The available operations are: AGREE (if the existing rule is strongly relevant for the task), REMOVE (if one existing rule is contradictory or similar/duplicated to other existing rules), EDIT (if any existing rule is not general enough or can be enhanced, rewrite and improve it), ADD (add new rules that are very different from existing rules and relevant for other tasks). Each needs to CLOSELY follow their corresponding formatting below (any existing rule not edited, not agreed, nor removed is considered copied):

AGREE <EXISTING RULE NUMBER>: <EXISTING RULE>

REMOVE <EXISTING RULE NUMBER>: <EXISTING RULE>

EDIT <EXISTING RULE NUMBER>: <NEW MODIFIED RULE>

ADD <NEW RULE NUMBER>: <NEW RULE>”

This process is a bit more involved, but still relies on manageable logic, and it specifically allows for insights that appeared helpful at points can be dynamically improved by learnings from subsequent experience. This process is illustrated in [Figure 7-4](#) below, in which the model is used to extract insights from pairs of successful and unsuccessful examples, and in which insights are promoted and demoted

over time, distilling out a small list of insights that are used to guide and improve the performance of the agent.



Figure 7-4. Caption to come

In this next section, we see how these rules are actually created, promoted, modified, and removed to enable the agent to improve its performance on the task over time.

```
def promote_insight(self, insight):
    if insight in self.insights:
        self.insights.remove(insight)
        self.promoted_insights.append(insight)
        print(f"Promoted: {insight}")
    else:
        print(f"Insight '{insight}' not found")

def demote_insight(self, insight):
```

```

        if insight in self.promoted_insights:
            self.promoted_insights.remove(insight)
            self.demoted_insights.append(insight)
            print(f"Demoted: {insight}")
        else:
            print(f"Insight '{insight}' not found")

def edit_insight(self, old_insight, new_insight):
    # Check in all lists
    if old_insight in self.insights:
        index = self.insights.index(old_insight)
        self.insights[index] = new_insight
    elif old_insight in self.promoted_insights:
        index = self.promoted_insights.index(old_insight)
        self.promoted_insights[index] = new_insight
    elif old_insight in self.demoted_insights:
        index = self.demoted_insights.index(old_insight)
        self.demoted_insights[index] = new_insight
    else:
        print(f"Insight '{old_insight}' not found")
        return
    print(f"Edited: '{old_insight}' to '{new_insight}'")

def show_insights(self):
    print("\nCurrent Insights:")
    print(f"Insights: {self.insights}")
    print(f"Promoted Insights: {self.promoted_insights}")
    print(f"Demoted Insights: {self.demoted_insights}")

```

```
def reflect(self, reflexion_prompt):
    # Build the state graph for reflection
    builder = StateGraph(MessagesState)
    builder.add_node("reflection", call_model)
    builder.add_edge(START, "reflection")
    graph = builder.compile()
    # Invoke the graph with the reflection prompt
    result = graph.invoke(
        {
            "messages": [
                HumanMessage(
                    content=reflexion_prompt
                )
            ]
        }
    )
    reflection = result["messages"][-1].content
    self.reflections.append(reflection)
    print(f"Reflection: {reflection}")
```

With sufficient feedback, this process provides an efficient way to learn from interactions with the environment and improve performance over time. An added advantage of this approach is its capability to facilitate the agent's gradual adaptation to non-stationary environments. Thus, if your agent needs to gradually adjust its policy to a changing environment, this approach

enables it to do so effectively. Let's now take a look at some example usage.

```
# Example usage:
agent = InsightAgent()
# Generate insights based on observations
insight1 = agent.generate_insight("The sales have
insight2 = agent.generate_insight("Customer comp
# Promote an insight
agent.promote_insight(insight1)
# Demote an insight
agent.demote_insight(insight1)
# Edit an existing insight
agent.edit_insight(insight2, "Refined insight: C
# Display all insights
agent.show_insights()
# Perform reflection using the LLM
reflexion_prompt = "Reflect on the current insight
agent.reflect(reflexion_prompt)
```

As you can see, even a small number of lines of code can enable an agent to continually learn from experience to improve performance on a specific task. These approaches are very practical, affordable, easy to implement, and enable continual adaptation from experience. In some cases though, and

especially when we have a large number of samples to learn from, it can make sense to consider fine-tuning.

Parametric Learning: Fine Tuning

Parametric learning involves adjusting the parameters of a predefined model to improve its performance on specific tasks. When we have evaluation data, we can use it to improve the performance of our system. It often makes sense to start with non-parametric approaches, because they are simpler and faster to implement. Adding examples and insights to the prompt takes time and computational resources, though. When we have a sufficient number of examples, it might be worth considering fine-tuning your models as well to improve your agentic performance on your tasks. Fine-tuning is a common approach where a pre-trained model is adapted to new tasks or datasets by making small adjustments to its parameters.

Fine Tuning Large Foundation Models

Most developers begin building agentic systems with large foundation models such as GPT-4o, Claude Sonnet, Gemini, and other similar classes of models because these offer an exceptional level of performance across a variety of tasks.

These models are pre-trained on extensive, general-purpose datasets, which equip them with a vast amount of linguistic and conceptual knowledge. Fine-tuning these models involves making targeted adjustments to their parameters, tailoring them to specific tasks or domains. This process allows developers to adapt the model's extensive knowledge to specialized applications, boosting its relevance and effectiveness on specific tasks while retaining its general capabilities.

The power of large models lies in their vast number of parameters and complex architectures, which enable them to understand subtle nuances and perform tasks with remarkable accuracy and fluency. Their size and capacity make them versatile enough to handle highly diverse tasks, from language translation and summarization to advanced reasoning and problem-solving. When fine-tuned, these models can apply their broad capabilities to specific challenges with a level of depth that smaller models may struggle to achieve. Fine-tuning essentially narrows the model's focus, refining its responses and aligning its outputs with the particular needs of the target application, while the original model serves as a robust foundation that ensures high-quality performance across tasks.

The flexibility of large foundation models enables developers to tackle complex, multi-faceted tasks that demand sophisticated comprehension and contextual sensitivity. For instance, a fine-tuned GPT-4o model can be used to analyze legal documents, understand nuanced financial reports, or engage in detailed customer support conversations. By adjusting the model for these particular contexts, fine-tuning ensures that it captures the specialized language, conventions, and subtleties specific to the domain. This capacity for domain-specific adaptation is one of the key reasons that large models are frequently chosen for high-stakes applications where accuracy, relevance, and contextual understanding are paramount.

However, the advantages of large models come with significant challenges. The sheer number of parameters—often numbering in the billions—makes fine-tuning computationally intensive. The process requires powerful hardware, typically high-end GPUs, to handle the model's demands. This level of computational power incurs substantial costs, both in terms of time and financial resources. As a result, fine-tuning a large model can be an expensive endeavor, particularly when ongoing updates are required or when the model is deployed in real-time systems. For many organizations, the resource demands of fine-tuning a large model can be a significant

barrier, limiting its accessibility to teams with substantial computational infrastructure.

Additionally, fine-tuning foundation models requires a considerable amount of high-quality, domain-specific data to be effective. Unlike small models, which may perform well on limited data, large models benefit from abundant examples that allow them to learn the finer points of the target domain. This dependence on data means that the preparation and curation of relevant training data is often a major component of the fine-tuning process. For some applications, collecting and processing this data may be challenging, time-consuming, or costly, particularly in specialized fields where labeled data is scarce. Furthermore, careful data handling is crucial to avoid introducing biases or overfitting, which could reduce the model's generalizability. In this process, we take a pre-trained language model, then apply our task-specific dataset, typically using supervised fine-tuning, in order to produce a fine-tuned model that will perform better on our particular dataset.



IMAGE TO COME

Figure 7-5. Caption to come

Despite these challenges, fine-tuning large models remains a powerful approach, especially in cases where high performance is critical and the resources to support such models are available. The unparalleled capacity of large models allows them to perform at exceptional levels when fine-tuned for specific tasks, often surpassing the performance of smaller, task-specific models. This makes them ideal for applications where accuracy, depth of understanding, and nuanced language handling are necessary, such as healthcare diagnostics, legal analysis, or complex technical support.

In addition to task-specific adaptation, fine-tuning large models can yield improvements in multi-tasking and transfer learning. Given their extensive pre-training, large models can often generalize insights from one task to related tasks, enhancing their versatility. Fine-tuning enables developers to optimize this cross-task capability, tailoring the model to perform effectively

across multiple domains or contexts within a single deployment. This adaptability makes large models particularly valuable in multi-functional systems, where they can support a range of tasks with minimal additional training.

The extensive capabilities of large models also contribute to the development of more intelligent, autonomous systems. By fine-tuning these models to handle complex decision-making tasks, developers can create agents capable of reasoning, planning, and acting autonomously. In scenarios requiring advanced problem-solving or decision-making, a fine-tuned large model can deliver insights and solutions that smaller models might not be capable of producing. This makes large models an integral component of sophisticated agentic systems, where they serve not just as assistants but as autonomous entities capable of handling intricate, variable tasks.

In summary, large foundation models offer a powerful solution for applications requiring high accuracy, adaptability, and nuanced understanding. Fine-tuning these models enables developers to harness their extensive pre-trained knowledge while optimizing performance for specialized tasks or domains. While the computational and data requirements are significant, the benefits of fine-tuning large models can justify the investment for applications demanding peak performance and

robust language comprehension, but is only recommended for a small number of use cases.

The Promise of Small Models

In contrast to large foundation models, small models offer a more resource-efficient alternative, making them suitable for many applications where computational resources are limited or response time is critical. While small models inherently have fewer parameters and simpler architectures, they can still be surprisingly effective when finely tuned to a specific task. This adaptability stems from their simplicity, which not only allows for faster adaptation but also enables rapid experimentation with different training configurations. Small models are particularly advantageous in environments where deploying larger, more complex models would be costly, impractical, or excessive given the task requirements.

The lean architecture of small models offers unique advantages in transparency and interpretability. Because they have fewer layers and parameters, it is easier to analyze their decision-making processes and understand the factors influencing their outputs. This interpretability is invaluable in applications where explainability is essential, such as finance, healthcare, and regulatory domains, where stakeholders need clear insights

into how and why decisions are made. For instance, a small model fine-tuned for medical image classification can be more straightforward to debug and validate, providing assurance to medical practitioners who rely on its predictions. In these contexts, smaller models contribute to increased accountability and trust, particularly in high-stakes applications where the reasoning behind decisions must be understandable and accessible.

Small models also enable agile development workflows. Their lightweight structure allows for faster iterations during fine-tuning, which can lead to quicker insights and adjustments. For developers working in agile environments or with limited access to high-performance computing, small models provide a flexible, responsive solution. They are ideal for tasks requiring continuous or incremental learning, where models must be frequently updated with new data to maintain relevance.

Moreover, small models can be deployed effectively in real-time systems, such as embedded devices, mobile applications, or IoT networks, where low latency is essential. In these applications, the reduced computational footprint of small models enables efficient processing without compromising the overall system's responsiveness.

Another key advantage of small models is their accessibility, both in terms of cost and availability. Many high-performing small models are open-source and freely available, including models like LLaMA and Phi, which can be modified to suit various use cases. This accessibility lowers barriers for organizations and developers who may not have the budget or infrastructure to support large-scale models. Small models allow these teams to experiment, innovate, and deploy machine learning solutions without incurring significant operational costs. This democratization of machine learning technology enables more organizations to harness the benefits of AI, contributing to a more inclusive development ecosystem.

In terms of performance, fine-tuned small models can achieve results comparable to those of larger models on specific, narrowly defined tasks. For example, a small model fine-tuned for sentiment analysis within a particular domain, such as financial reports, can achieve high accuracy because it specializes in recognizing patterns specific to that context. When applied to well-defined tasks with clear data boundaries, small models can match, or even surpass, the performance of larger models by focusing all of their capacity on the relevant aspects of the task. This efficiency is particularly valuable in applications with high accuracy demands but limited data,

where small models can be customized to perform effectively without overfitting.

In addition to their efficiency, small models support a sustainable approach to AI development. Training and deploying large models consume significant energy and computational resources, which contribute to environmental impacts. Small models, however, require substantially less energy for training and inference, making them a more sustainable choice for applications where resource consumption is a concern. Organizations prioritizing environmental sustainability can integrate small models as part of their green AI strategies, contributing to reduced carbon footprints without compromising on innovation.

The promise of small models extends to settings where frequent updates or retraining are needed. In scenarios where the data landscape changes rapidly—such as social media sentiment analysis, real-time fraud detection, or personalized recommendations—small models can be quickly retrained or fine-tuned with new data, adapting rapidly to changing patterns. This ability to frequently update without high retraining costs makes small models ideal for applications where adaptability is crucial. Additionally, small models can be deployed in federated learning environments, where data

privacy concerns require models to be trained across decentralized data sources. In these settings, small models can be efficiently fine-tuned on edge devices, enabling privacy-preserving AI solutions.

Fine-tuning small models offers a promising approach for various applications where large models may be impractical. Their adaptability, interpretability, cost-effectiveness, and support for sustainable development make small models a compelling choice for targeted, high-performance applications. By focusing on relevant features within specific domains, small models can deliver impressive results while enabling greater flexibility in deployment and maintenance. The availability of open-source small models further empowers developers to create customized, efficient agentic systems without the overhead of large-scale infrastructure, making AI accessible and sustainable across a broad range of industries.

Function Calling Fine Tuning

Function calling fine-tuning is an advanced technique that focuses on enhancing an agent's ability to interact effectively with external tools, APIs, or functions. As agents become more integrated into complex systems, the need for precise and reliable function calls increases. Fine-tuning models specifically

for function calling enables agents to understand when and how to invoke functions, parse arguments correctly, and handle responses appropriately.

Function calling fine-tuning equips an agent with a deep understanding of the structure and expected parameters of available functions or APIs. This includes knowledge of function names, argument types, and return formats, which enables the agent to accurately match user requests with the appropriate functions. By comprehending the API schema, the agent can seamlessly integrate with external systems and execute tasks based on the user's instructions. Fine-tuning also involves training the model to make contextual decisions about when to invoke a function.

For example, if a user inquires about the weather, the agent might decide to call a weather API to retrieve the forecast. However, if the query is hypothetical or does not actually require data retrieval, the agent can choose to refrain from making the call. This contextual decision-making ensures that function calls are meaningful and relevant, preserving system resources and improving response accuracy.

Another essential aspect of function calling fine-tuning is the model's ability to parse and validate arguments accurately. The

agent must be able to extract the relevant parameters from user inputs, align them with the expected API parameters, and verify that they are in the correct format. This step reduces the likelihood of errors during function calls, as arguments are validated before the call is made. In addition to argument parsing, fine-tuning trains the model to handle errors and exceptions that may arise during function execution. If a function call fails due to a missing or incorrect parameter, for instance, the agent can recognize the issue and either prompt the user for additional information or offer alternative solutions. This ability to recover gracefully from errors ensures that the agent remains responsive and helpful, even in situations where function calls do not initially succeed.

Fine-tuning for function calls offers several benefits, starting with enhanced reliability. By reducing the likelihood of function call errors, the agent becomes more dependable in its task execution, ensuring that actions are completed accurately and without unnecessary interruptions. This reliability is coupled with increased versatility, as a functionally fine-tuned agent can perform a broader range of actions, from retrieving data to controlling connected devices. This expanded capability makes the agent useful in a wider variety of applications, where it can fulfill complex requests that go beyond basic interactions.

Ultimately, this enhanced versatility leads to an improved user experience. With the ability to automate tasks, retrieve precise information, and perform complex actions seamlessly, the agent provides users with smooth and efficient interactions, improving overall satisfaction.

Implementing function calling fine-tuning requires careful data preparation. Developing a fine-tuning dataset involves creating examples of function invocations, expected inputs, and outputs, covering a wide range of potential scenarios and variations in user requests. This data diversity is essential for training a robust model capable of handling real-world variations in input.

Additionally, deciding on the representation of functions within the model is crucial for successful training. Structured prompts or special tokens can be employed to help the model identify and process function-related text accurately, enabling it to make function calls as part of its responses. Lastly, security and validation are key considerations. Function calling introduces risks if not properly controlled, so implementing safeguards such as whitelisting specific functions and validating input parameters helps prevent the agent from executing unintended or potentially harmful actions. This approach ensures that the

agent can perform function calls safely and responsibly, maintaining system integrity and protecting user data.

Conclusion

Learning in agentic systems encompasses a variety of approaches, each offering unique advantages for improving AI performance and adaptability. Non-parametric learning allows agents to learn from experiences without fixed models, emphasizing flexibility and real-world applicability. Parametric learning, through fine-tuning, enhances the specialization of pre-trained models, balancing efficiency and performance. Transfer learning leverages existing knowledge to expedite and improve training on new tasks. Together, these methodologies form a robust framework for developing intelligent, adaptable, and efficient AI agents.

· <https://arxiv.org/abs/2005.14165>

· Noah Shinn et al.: Reflexion: Language Agents with Verbal Reinforcement Learning. Preprint. Under review. <https://arxiv.org/pdf/2303.11366>

· Andrew Zhao et al.: ExpeL: LLM Agents Are Experiential Learners. AAAI Conference on Artificial Intelligence 2024. <https://arxiv.org/abs/2308.10144>

Chapter 8. From One Agent to Many

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the eighth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Scaling systems from a single agent to multiple agents can significantly enhance their ability to solve complex tasks. However, this scalability introduces new challenges that require careful consideration. In this chapter, we explore the advantages and drawbacks of transitioning from one agent to many, discuss how to determine the optimal number of agents for various scenarios, examine different coordination

strategies, and review frameworks that facilitate multi-agent system development.

How Many Agents Do I Need?

Determining the optimal number of agents is crucial for effective system performance. This decision hinges on understanding the complexity of the tasks, the environment, and the interactions between agents. The goal is to balance the benefits of multi-agent collaboration against the potential complications it introduces.

Single-Agent Scenarios

Single-agent systems are suitable for tasks that are well-defined, relatively simple, or isolated. In these scenarios, a solitary agent is responsible for performing tasks and making decisions independently. The primary benefits include:

- **Simplicity:** Easier implementation and management.
- **Lower Resource Requirements:** Less computational overhead.
- **Direct Control:** Clear understanding of the agent's behavior.

However, single-agent systems may struggle with tasks requiring diverse skill sets, parallel processing, or adaptability to complex environments. They are best suited for straightforward tasks or environments with minimal complexity. If a single agent is adequate for the use case and performance requirements, there is no need to add further complexity to the system. For most use cases, though, the key bottleneck arises when the number of skills and responsibilities increases. When an agent is expected to choose the correct skill from a set, performance degrades as the potential number of skills increases. At this point, it often makes sense to encapsulate multiple skills into larger groupings, and then choose from the skillset first, and then subsequently choose the skill within the skillset. When the number of skills exceeds the ability to correctly choose a correct skill, then decomposing the skills to distinct agents with appropriate responsibilities can improve reliability and performance.

Multi-Agent Scenarios

In multi-agent systems, multiple agents collaborate to achieve shared goals, an approach that is especially advantageous when tasks are complex and require varied skill sets, parallel processing, or adaptability to dynamic environments. A key

benefit of multi-agent systems is specialization: each agent can be assigned specific roles or areas of expertise, allowing the system to leverage each agent's strengths effectively. This division of labor enables agents to focus on defined aspects of a task, which improves efficiency and ensures that specialized skills are applied where they are most needed. By distributing skills and responsibilities across agents, multi-agent systems address the limitations faced by single-agent systems, especially when tasks require expertise across different domains or when the number of skills required exceeds what a single agent can manage reliably.

Parallel processing is another significant advantage. In multi-agent systems, tasks can be split and processed concurrently by different agents, reducing overall execution time and increasing throughput. This parallelism is essential for applications where real-time performance is critical, such as monitoring systems, data processing pipelines, or autonomous vehicle coordination. By operating simultaneously on different parts of a task, agents can accelerate progress and meet the demands of time-sensitive environments more effectively.

Multi-agent systems also offer robustness through redundancy. When multiple agents are available to complete a task, the system can tolerate individual agent failures without disrupting

overall functionality. Redundancy ensures that if one agent fails or encounters an obstacle, other agents can take over its responsibilities, allowing the system to maintain stable performance even in unpredictable conditions. This enhanced fault tolerance makes multi-agent systems particularly valuable in mission-critical applications, where uninterrupted operation is essential.

Adaptability is another core advantage, as multi-agent systems can respond dynamically to changing conditions. By coordinating their actions, agents can reallocate roles and responsibilities as needed, adapting to new information or environmental changes in real time. This adaptability allows the system to remain efficient and effective in complex and unpredictable scenarios, where static, single-agent approaches may struggle to keep up.

However, multi-agent systems are not without challenges. With multiple agents interacting, the complexity of coordination increases, requiring sophisticated communication and synchronization mechanisms to ensure agents work harmoniously. Communication overhead is another challenge, as agents must frequently exchange information to stay aligned and avoid duplicating efforts. This need for communication can slow down the system and introduce additional resource

demands, especially in large-scale applications. Additionally, conflicts between agents may arise if they pursue overlapping goals or fail to prioritize effectively, necessitating protocols for conflict resolution and resource allocation.

In sum, while multi-agent systems offer powerful advantages in handling complex, multi-faceted tasks, they also require careful planning to manage the additional complexity and coordination requirements they introduce. By assigning agents distinct roles, enabling parallel processing, and incorporating adaptability and redundancy, multi-agent systems can achieve high levels of performance, reliability, and flexibility, particularly in scenarios where a single-agent approach would fall short.

Principles for Adding Agents

When expanding a system by adding more agents, a strategic approach is essential to ensure the system remains efficient, manageable, and effective. The following principles serve as guidelines for optimizing agent-based design and functionality.

Task Decomposition

Task Decomposition is a foundational principle, emphasizing the importance of breaking down complex tasks into smaller, manageable subtasks. By decomposing

tasks, each agent can focus on a specific aspect of the workload, simplifying its responsibilities and improving efficiency. Clear task boundaries reduce overlap and redundancy, ensuring that each agent's contribution is valuable and that no effort is wasted. This decomposition not only enhances individual agent performance but also makes the system easier to coordinate and scale.

Specialization

Specialization allows agents to be assigned roles that match their strengths, thereby maximizing the system's collective capabilities. When each agent is tasked with activities that align with its specific functions, the system operates with greater precision and effectiveness.

Specialized agents are more adept at handling particular types of work, which translates to improved performance and faster task execution overall. By designing agents with distinct responsibilities, the system can leverage diverse expertise to address complex or multidisciplinary tasks.

Parsimony

Parsimony is a guiding principle that encourages adding only the minimal number of agents necessary to achieve

the desired functionality and performance. This principle emphasizes simplicity and efficiency, reminding developers that each agent added to the system introduces additional communication overhead, coordination complexity, and resource demands. By adhering to parsimony, developers avoid unnecessary agent proliferation, which can lead to increased maintenance burdens and potential performance bottlenecks. Parsimony requires careful assessment of each agent's role and a disciplined approach to agent allocation, ensuring that each addition provides clear value to the system. Before adding an agent, developers should consider whether its responsibilities could be fulfilled by existing agents or by enhancing current capabilities. This focus on simplicity results in a streamlined, more manageable system that performs effectively without excessive redundancy. Ultimately, parsimony promotes an efficient, lean multi-agent system that maximizes functionality while minimizing the risks and costs associated with complexity.

Coordination

Coordination is critical for the harmonious operation of multi-agent systems. To maintain alignment among

agents, robust communication protocols must be established, facilitating efficient information sharing and reducing the risk of conflicts. Coordination mechanisms should also include protocols for conflict resolution, particularly when agents have overlapping tasks or resource requirements. When agents can exchange information seamlessly and resolve issues autonomously, the system is more resilient and adaptable, capable of responding efficiently to dynamic scenarios.

Robustness

Robustness is essential for enhancing fault tolerance and resilience. Redundancy involves adding agents that can take over if others fail, providing backup support that ensures uninterrupted operation. In high-stakes environments, redundancy is invaluable for maintaining system stability and reliability. Robustness also encompasses designing agents and workflows that can withstand unexpected disruptions, such as network failures or agent downtime. By embedding redundancy and robustness into the system, developers can ensure that it remains functional even in adverse conditions.

Efficiency

Efficiency helps in assessing the trade-offs between adding agents and the potential complexity or resource demands that come with them. Each additional agent increases computational requirements and coordination overhead, so it is crucial to weigh the advantages of expanded functionality against these costs. By carefully evaluating the costs and benefits of each agent addition, developers can make informed decisions that balance system performance, resource efficiency, and scalability.

By following these principles, developers can determine the optimal number and configuration of agents required to achieve the desired balance of performance, efficiency, and complexity. This thoughtful approach enables the creation of multi-agent systems that are both capable and sustainable, maximizing the benefits of additional agents while minimizing potential downsides.

Multi-Agent Coordination

Effective coordination among agents is critical for the success of multi-agent systems. Various coordination strategies can be employed, each with its advantages and challenges. This section explores five primary coordination strategies.

Democratic Coordination

In democratic coordination, each agent within the system is given equal decision-making power, with the goal of reaching consensus on actions and solutions. This approach is characterized by decentralized control, where no single agent is designated as the leader. Instead, agents collaborate and share information equally, contributing their unique perspectives to collectively arrive at a decision. The key strength of democratic coordination is its robustness; because no agent holds a dominant role, the system has no single point of failure. This means that even if one or more agents experience failures, the overall system can continue functioning effectively. Another advantage is flexibility: when agents collaborate openly, they can quickly adapt to changes in their environment by updating their collective input. This adaptability is essential in dynamic settings where responsiveness to new information is crucial.

Moreover, democratic coordination promotes equity among agents, ensuring that all participants have an equal voice, which can lead to fairer and more balanced outcomes.

However, democratic coordination comes with its own set of challenges. The process of reaching a consensus often requires extensive communication between agents, leading to significant

communication overhead. As each agent must contribute and negotiate their perspective, the decision-making process can also be slow, potentially causing delays in environments where quick responses are necessary. Furthermore, implementing a democratic coordination protocol is often complex, as it requires well-defined communication and conflict-resolution mechanisms to facilitate consensus-building. Despite these challenges, democratic coordination is particularly well-suited for applications that prioritize fairness and robustness, such as distributed sensor networks or collaborative robotics, where each agent's contribution is valuable and consensus is essential for system success.

Manager Coordination

Manager coordination adopts a more centralized approach, where one or more agents are designated as managers responsible for overseeing and directing the actions of subordinate agents. In this model, managers take on a supervisory role, making decisions, distributing tasks, and resolving conflicts among agents under their guidance. One of the primary advantages of manager coordination is its streamlined decision-making. Since managers have the authority to make decisions on behalf of the group, the system

can operate more efficiently, bypassing the lengthy negotiation process required in democratic systems. This centralization also allows managers to clearly assign tasks and responsibilities, ensuring that agents focus on specific objectives without duplicating efforts or causing conflicts. Additionally, manager coordination simplifies communication pathways, as subordinate agents primarily communicate with their designated manager rather than with every other agent, reducing coordination complexity.

However, the reliance on managers introduces certain vulnerabilities. A single point of failure exists because if a manager agent fails or is compromised, the entire system may experience disruptions. Additionally, scalability becomes a concern as the system grows; managers can become bottlenecks if they cannot handle the increased volume of tasks or interactions required in larger networks. Finally, the centralized nature of decision-making in manager coordination can reduce adaptability, as managers may not always be able to make the most informed decisions based on real-time changes within each subordinate's environment. This type of coordination is particularly effective in structured, hierarchical settings like manufacturing systems or customer support centers, where centralized control allows for optimized workflows and quicker conflict resolution.

Hierarchical Coordination

Hierarchical coordination takes a multi-tiered approach to organization, combining elements of both centralized and decentralized control through a structured hierarchy. In this system, agents are organized into multiple levels, with higher-level agents overseeing and directing those below them while allowing subordinate agents a degree of autonomy. This approach provides significant scalability benefits, as the hierarchical structure enables coordination responsibilities to be distributed across multiple levels. By doing so, the system can manage a large number of agents more efficiently than a fully centralized model. The layered design also introduces redundancy, as tasks can be managed at different levels, improving fault tolerance. Clear lines of authority within the hierarchy streamline operations, with higher-level agents handling strategic decisions and lower-level agents focusing on tactical execution.

Despite these advantages, hierarchical coordination presents its own challenges. The complexity of designing a hierarchical system can be substantial, as each level must be carefully structured to ensure smooth coordination between layers. Communication delays can arise due to the need for information to propagate through multiple levels before

reaching all agents, which can slow down responsiveness to urgent changes. Additionally, decision-making at higher levels may introduce latency, as lower-level agents may need to wait for instructions before acting. Despite these challenges, hierarchical coordination is well-suited for large, complex systems such as supply chain management or military operations, where different levels of coordination can handle both high-level planning and on-the-ground execution.

Actor-Critic Approaches

Actor-critic approaches, originally developed in the context of reinforcement learning, involve a dual-agent setup where one agent, the “actor,” makes decisions, while another agent, the “critic,” evaluates those decisions and provides feedback. This approach allows agents to learn and adapt over time, improving their performance through feedback loops. A key advantage of actor-critic methods is that they enable adaptive learning, allowing agents to refine their behavior based on evaluative feedback from the critic. This framework supports decentralized decision-making, as each actor can make choices independently while still benefiting from the critic’s feedback. Due to its decentralized nature, the actor-critic model scales well, making it suitable for systems with large numbers of

agents that need to operate autonomously but benefit from structured feedback to enhance their performance.

However, implementing actor-critic approaches can be challenging. The underlying algorithms are often sophisticated, requiring considerable computational resources and a nuanced understanding of reinforcement learning techniques. Ensuring that all actors align with the system's overarching goals can also be difficult, especially as each actor seeks to optimize its own decisions based on feedback that may be influenced by local conditions rather than global objectives. Stability is another concern, as learning processes may become erratic or even destabilize the system if the feedback signals are noisy or inconsistent. Nevertheless, actor-critic approaches are highly effective in dynamic environments, such as financial trading systems or adaptive traffic control, where agents benefit from the ability to learn and improve in response to changing conditions.

Automated Design of Agent Systems

Automated Design of Agentic Systems (ADAS) represents a transformative approach to agent development, shifting away from hand-crafted architectures and toward systems that can design, evaluate, and iteratively improve themselves. As

articulated by Shengran Hu, Cong Lu, and Jeff Clune¹, the central idea of ADAS is that, rather than manually constructing each component of an agent, we can enable a higher-level “meta-agent” to automatically create, assess, and refine agentic systems. This approach opens up a new research frontier, one that could yield agents capable of adapting to complex, shifting environments and continually improving their own capabilities without direct human intervention. ADAS builds on the idea that, historically, hand-designed solutions in machine learning have often been replaced by learned or automated alternatives, suggesting that agentic systems, too, may benefit from this transition.

In ADAS, Foundation Models serve as flexible, general-purpose modules within an agent’s architecture. These models, which already power strategies such as Chain-of-Thought reasoning, Self-Reflection, and Toolformer-based agents, form a base upon which more specialized or task-specific capabilities can be layered. Yet, ADAS seeks to advance beyond these traditional approaches by enabling agents to invent entirely new structures and modules autonomously. The versatility of Foundation Models provides an ideal starting point, but ADAS leverages automated processes to push beyond pre-defined capabilities, enabling agents to evolve novel prompts, control flows, and tool use. These building blocks are not static; rather,

they are generated dynamically by the meta-agent, which can continuously experiment with new designs in response to changing requirements or opportunities for improvement.

The backbone of ADAS is the concept of defining agents through code. By utilizing programming languages, which are Turing-complete, this framework theoretically allows agents to invent any conceivable structure or behavior. This includes complex workflows, creative tool integrations, and innovative decision-making processes that a human designer may not have foreseen. The power of ADAS lies in this code-based approach, which treats agents not as static entities but as flexible constructs that can be redefined, modified, and optimized over time. The potential of this approach is vast: in principle, a meta-agent could develop an endless variety of agents, continually refining and combining elements in pursuit of higher performance across diverse tasks.

Central to ADAS is the Meta Agent Search (MAS) algorithm, which demonstrates how a meta-agent can autonomously generate and refine agentic systems. In MAS, the meta-agent acts as a designer, writing code to define new agents and testing these agents against an array of tasks. Each successful design is archived, forming a continuously growing knowledge base that informs the creation of future agents. MAS operates through an

iterative cycle: the meta-agent evaluates each agent's performance, learns from the outcomes, and applies that knowledge to generate new agents with potentially superior capabilities. The meta-agent is thus both a creator and a curator, balancing exploration of new designs with exploitation of successful patterns. This process mirrors the evolution of biological systems, where successful traits are preserved and iteratively modified to adapt to new challenges.

The results of MAS reveal an intriguing property of agents designed through ADAS: they tend to maintain high levels of performance even when applied to new domains and models. For instance, agents developed in specific coding environments have demonstrated surprising efficacy when transferred to other fields, such as scientific or mathematical problem-solving. This robustness across domains suggests that agents created through MAS are not merely optimized for one-off tasks; rather, they embody more general principles and adaptive structures that enable them to excel even when the specifics of the environment change. This cross-domain transferability reflects a fundamental advantage of automated design: by generating agents that are inherently flexible, MAS produces solutions that can generalize more effectively than those designed for narrow, specialized contexts.

ADAS holds significant promise, yet its development requires careful consideration of both ethical and technical dimensions. The potential to automate the design of ever-more-powerful agents introduces questions about safety, reliability, and alignment with human values. While MAS offers a structured and exploratory approach, it is crucial to ensure that the evolving agents adhere to ethical standards and do not develop unforeseen behaviors that could be misaligned with human intentions. Ensuring that these systems are beneficial necessitates a balance between autonomy and constraint, allowing agents the freedom to innovate while guiding them to operate within safe and predictable bounds.

The trajectory of ADAS suggests a future where agentic systems can autonomously adapt, improve, and tackle an expanding range of tasks with minimal human intervention. As ADAS advances, the ability of agents to develop more sophisticated designs will likely become a cornerstone of AI research, providing tools that can address increasingly complex, evolving challenges. In this way, ADAS offers a glimpse into a future of intelligent systems capable of self-improvement and innovation, embodying a shift from static, pre-designed agents to adaptive, autonomous systems that grow alongside our expanding needs.

Multi-Agent Frameworks

Several frameworks facilitate the development and deployment of multi-agent systems. Selecting the appropriate framework depends on the specific needs and constraints of the project.

Do-It-Yourself (DIY)

A do-it-yourself (DIY) approach to multi-agent systems offers the ultimate flexibility, allowing developers to build a system tailored to their specific needs without the constraints or assumptions of pre-built frameworks. This approach is ideal for highly specialized applications or experimental research where the particularities of agent interaction, communication protocols, and coordination mechanisms need to be closely controlled and fine-tuned. By developing a custom multi-agent infrastructure, developers can optimize performance for specific hardware configurations, integrate unique communication standards, or test novel agent behaviors that existing frameworks may not support.

However, the DIY approach is not without challenges. Building a robust multi-agent system from scratch requires a deep understanding of distributed systems, network protocols, and

software engineering best practices. The lack of built-in support for common tasks, such as agent lifecycle management or fault tolerance, means that every aspect of the system—from the architecture to the communication protocols—must be carefully designed and implemented. Despite these challenges, the DIY approach is invaluable for pushing the boundaries of multi-agent systems and exploring new possibilities in agent coordination.

LangGraph

LangGraph is a multi-agent framework focused on language-based interactions, leveraging natural language as a primary means of communication and decision-making among agents. In LangGraph, agents communicate through structured language exchanges, making it particularly well-suited for applications that rely on rich, context-sensitive information exchange, such as collaborative problem-solving, instructional systems, and conversational AI.

The framework provides tools to facilitate language-based negotiation, goal setting, and knowledge sharing among agents, making it possible for agents to dynamically adjust their strategies based on the evolving context of their language interactions. LangGraph also includes features for tracking

conversational history and context, enabling agents to maintain coherence in their interactions over extended dialogues. While LangGraph's language-centered approach offers powerful capabilities for collaborative applications, it can be limited in scenarios where real-time performance or direct action coordination is essential. Nevertheless, LangGraph's unique focus on language provides a valuable platform for exploring multi-agent systems in domains where communication and interpretability are as important as raw computational power.

Autogen

Autogen stands out as a leading framework for managing complex multi-agent interactions, especially in environments where agents must work together to achieve goals through coordinated actions and decision-making. Autogen's strength lies in its versatility, providing built-in support for various types of agent coordination, such as task sharing, parallel processing, and hierarchical management. The framework is highly extensible, allowing developers to define custom agent behaviors, decision-making protocols, and reward structures. Autogen also offers robust logging and monitoring tools, which facilitate debugging and provide insights into agent behavior patterns, making it particularly valuable for research

applications. Additionally, Autogen's modular architecture enables easy integration with machine learning models, reinforcement learning algorithms, and other AI components, allowing agents to incorporate advanced learning capabilities into their interactions. Autogen's extensive support for multi-agent workflows, coupled with its flexible configuration options, makes it an ideal choice for both research-oriented projects and real-world applications where coordination efficiency and agent scalability are paramount.

Crew AI

CrewAI is a framework designed to orchestrate role-playing, autonomous AI agents, fostering collaborative intelligence to tackle complex tasks. By enabling agents to assume specific roles, share goals, and operate cohesively, CrewAI facilitates the development of sophisticated multi-agent systems.

The framework offers a range of features to streamline the creation and management of AI workflows. Developers can utilize CrewAI's framework or UI Studio to build multi-agent automations, whether coding from scratch or leveraging no-code tools and templates. Once built, these "crews" can be deployed confidently with powerful tools for different deployment types, including autogenerated user interfaces.

CrewAI also provides monitoring capabilities to track the performance and progress of each crew, enabling continuous improvement through testing and training tools. ([CrewAI](#))

CrewAI's modular architecture supports seamless integration with various large language models (LLMs) and cloud platforms, offering flexibility in deployment. The framework's emphasis on role-based task allocation allows agents to focus on specialized aspects of a workload, enhancing efficiency and collaboration. Additionally, CrewAI's support for real-time communication protocols ensures that agents can update one another on task progress and make necessary adjustments dynamically.

By providing a structured yet flexible approach to multi-agent orchestration, CrewAI serves as a valuable resource for developers and researchers interested in exploring the dynamics of agent coordination and execution. Its comprehensive suite of tools and features makes it an ideal choice for both research-oriented projects and real-world applications where coordination efficiency and agent scalability are paramount.

Swarm

Swarm is an experimental educational framework developed by OpenAI's Solutions team, designed to explore ergonomic and lightweight multi-agent orchestration. It focuses on making agent coordination and execution highly controllable and easily testable. Swarm achieves this through two primary abstractions: Agents and handoffs. An Agent encompasses instructions and tools, and can at any point choose to hand off a conversation to another Agent. These primitives are powerful enough to express rich dynamics between tools and networks of agents, allowing developers to build scalable, real-world solutions while avoiding a steep learning curve.

Swarm is entirely powered by OpenAI's Chat Completions API and is stateless between calls. It is not intended for production use and does not have official support. The primary goal of Swarm is to showcase the handoff and routines patterns explored in OpenAI's "Orchestrating Agents: Handoffs & Routines" cookbook. It is not meant as a standalone library and is primarily for educational purposes.

Swarm's `client.run()` function is analogous to the `chat.completions.create()` function in the Chat Completions API—it takes messages and returns messages without saving state between calls. Importantly, it also handles Agent function

execution, handoffs, context variable references, and can take multiple turns before returning to the user.

By providing a structured yet flexible approach to multi-agent orchestration, Swarm serves as a valuable resource for developers and researchers interested in exploring the dynamics of agent coordination and execution.

Conclusion

The transition from single-agent to multi-agent systems offers significant advantages in addressing complex tasks, enhancing adaptability, and increasing efficiency. Yet, as we've explored in this chapter, the scalability that comes with adding more agents brings challenges that demand careful planning. Deciding on the optimal number of agents requires a nuanced understanding of task complexity, potential task decomposition, and the cost-benefit balance of multi-agent collaboration.

Coordination is critical to success in multi-agent systems, and a variety of coordination strategies—such as democratic, manager-based, hierarchical, actor-critic approaches, and automated design with ADAS—provide different trade-offs between robustness, efficiency, and complexity. Each

coordination strategy offers unique advantages and limitations, suited to particular scenarios, and careful selection can significantly enhance a system's effectiveness and reliability.

Choosing an appropriate framework is equally important. Custom-built solutions allow for maximum flexibility, but established frameworks like Autogen, Crew AI, and Swarm offer structured, scalable architectures and powerful tools for managing agent interactions and task orchestration. Selecting the right framework ensures not only that the multi-agent system can handle current requirements but also that it remains adaptable for future expansion and unforeseen challenges.

By understanding these factors and applying them thoughtfully, developers can create multi-agent systems that are not only robust and capable but also prepared to meet the demands of increasingly complex, dynamic tasks in real-world applications. This strategic approach enables multi-agent systems to evolve as powerful solutions that drive meaningful advancements across various domains.



Your gateway to knowledge and culture. Accessible for everyone.



z-library.sk

z-lib.gs

z-lib.fm

go-to-library.sk



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>