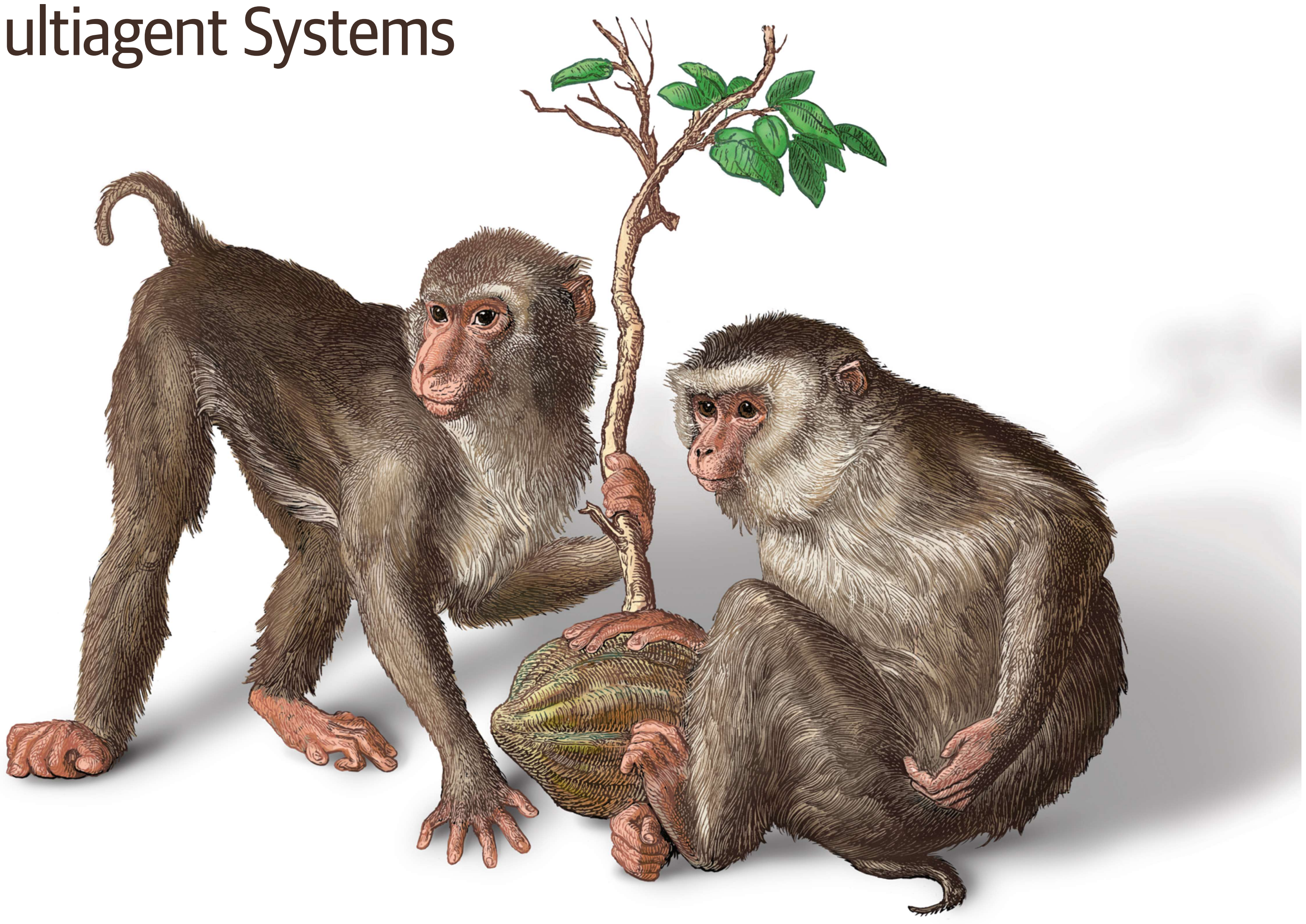
O'REILLY®

Building Applications with AI Agents

Designing and Implementing Multiagent Systems



Michael Albada

"The best single-volume introduction to building AI agent systems—you can read hundreds of papers or this one book."

Arun Rao
Ex-Meta GenAI group, adjunct professor at UCLA

Building Applications with AI Agents

Generative AI has revolutionized how organizations tackle problems, accelerating the journey from concept to prototype to solution. As the models become increasingly capable, we have witnessed a new design pattern emerge: AI agents. By combining tools, knowledge, memory, and learning with advanced foundation models, we can now sequence multiple model inferences together to solve ambiguous and difficult problems. From coding agents to research agents to analyst agents and more, we've already seen agents accelerate teams and organizations. While these agents enhance efficiency, they often require extensive planning, drafting, and revising to complete complex tasks, and deploying them remains a challenge for many organizations, especially as technology and research rapidly develops.

This book is your indispensable guide through this intricate and fast-moving landscape. Author Michael Albada provides a practical and research-based approach to designing and implementing single- and multiagent systems. It simplifies the complexities and equips you with the tools to move from concept to solution efficiently.

- **Understand the distinct features of foundation model-enabled AI agents**
- **Discover the core components and design principles of AI agents**
- **Explore design trade-offs and implement effective multiagent systems**
- **Design and deploy tailored AI solutions, enhancing efficiency and innovation in your field**

Michael Albada is a seasoned machine learning engineer with expertise in deploying large-scale solutions for major tech firms including Uber, ServiceNow, and Microsoft. He holds degrees from Stanford University, the University of Cambridge, and Georgia Tech, specializing in machine learning.

DATA

US $79.99 CAN $99.99
ISBN: 978-1-098-17650-1

57999

9 781098 176501

O'REILLY®

Praise for *Building Applications with AI Agents*

Finally, a book about really scaling AI into the human workforce. Michael does a great job leveraging his expertise at scalable organizations like Uber and Microsoft to teach any technical leader in a small and medium business how to really create scalable agentic solutions for their transformation.

—Birju Shah, professor of product management and AI at Kellogg School of Management, Northwestern University, former head of Uber AI product team

A sharp, practical guide, Building Applications with AI Agents equips leaders to move from generative AI hype to real-world systems. It distills complex concepts into actionable strategies, bridging vision and execution for organizations seeking measurable efficiency and competitive edge.

—Amanda Cheng, partner of Founders Bay

As a clinician working at the intersection of medicine and technology, I found this to be an essential read for anyone building AI agents—clear, practical, and rich with insight into tools, orchestration, and design patterns relevant to healthcare use cases like intake, triage, and workflow integration.

—Carrie Ho, MD, assistant professor, hematologist/oncologist, UCSF

This is the book I wish every team had before deploying agents, a clear, rigorous approach to architecture, safety, and measurement that accelerates delivery and reduces risk.

—Brad Sarsfield, senior director, Microsoft Security AI Research & Development

The best single-volume introduction to building AI agent systems—you can read hundreds of papers or this one book.

—Arun Rao, ex-Meta GenAI group, adjunct professor at UCLA

Building Applications with AI Agents

Designing and Implementing Multiagent Systems

Michael Albada

O'REILLY®

Building Applications with AI Agents
by Michael Albada

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

Acquisitions Editor: Nicole Butterfield
Development Editor: Shira Evans
Production Editor: Ashley Stussy
Copyeditor: nSight, Inc.
Proofreader: Piper Content Partners
Indexer: nSight, Inc.
Cover Designer: Karen Montgomery
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

September 2025: First Edition

Revision History for the First Edition
2025-09-16: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098176501* for release details.

978-1-098-17650-1

[LSI]

Table of Contents

Preface

When I first started connecting language models, tools, orchestration, and memory together into what we now call an agent, I was surprised by how capable this design pattern was, and just how much confusion there was about this topic. During my time building agents and sharing my findings on incident investigation, threat hunting, vulnerability detection, and more, I found that this latest design pattern enabled us to solve whole new classes of problems, but also came with many practical hurdles to making them reliable for real-world applications. Engineers, scientists, product managers, and leadership all wanted to know more. "How do I get my agent to work?" "I can get my agent to work some of the time, but how do I get it to work most or all of the time?" "How do I choose a model for my use case?" "How do I design good tools for my agent?" "What kind of memory do I need?" "Should I use RAG?" "Should I build a single-agent or multiagent system?" "What architecture should I use?" "Do I need to fine-tune?" "How do I enable agents to learn from experience and improve over time?"

While there are many blog posts and research papers that focus on specific aspects of the topic of designing agent systems, I realized there were a lack of accessible, holistic, trustworthy guides for this. I couldn't find the book that I wanted to share with my colleagues, so I set out to write it.

Through in-depth discussions, I've helped teams navigate the complexities of AI agents, considering their unique goals, constraints, and environments. AI agent systems are intricate, blending autonomy, decision making, and interaction in ways that traditional software doesn't. They're data-driven, adaptive, and involve multiple components like perception, reasoning, action, and learning, all while interfacing with users, tools, and other agents. Complicating matters, the foundation models that power these agents are probabilistic and stochastic by nature, making evaluation and testing more challenging.

This book takes a comprehensive approach to building applications with AI agents. It covers the entire lifecycle, from conceptualization to deployment and maintenance, illustrated with real-world case studies, supported by references, and reviewed by practitioners in the field. Sections on advanced topics—like agent architectures, tool integration, memory systems, orchestration, multiagent coordination, measurement, monitoring, security, and ethical considerations—are further refined by expert input.

Writing this book has been a journey of discovery for me as well. The initial drafts sparked conversations that challenged my views and introduced new ideas. I hope this process continues as you read it, bringing your own insights. Feel free to share any feedback you might have for this book via Twitter (X), LinkedIn, my personal website, or any other channels that you can find.

What This Book Is About

This book provides a practical framework for building robust applications using AI agents. It addresses key challenges and offers solutions to questions such as:

- What defines an AI agent, and when should I use one? How do agents differ from traditional machine learning (ML) systems?
- How do I design agent architectures for specific use cases, including scenario selection, and core components like tools, memory, planning, and orchestration?
- What are effective strategies for agent planning, reasoning, execution, tool selection, and topologies like chains, trees, and graphs?
- How can I enable agents to learn from experience through nonparametric methods, fine-tuning, and transfer learning?
- How do I scale from single-agent to multiagent systems, including coordination patterns like democratic, hierarchical, or actor-critic approaches?
- How do I evaluate and improve agent performance with metrics, testing, and production monitoring?
- What tools and frameworks are best for development, deployment, and securing agents against risks?
- How do I ensure agents are safe, ethical, and scalable, with considerations for user experience (UX), trust, bias, fairness, and regulatory compliance?

The content draws from established engineering principles and emerging practices in AI agents, with case studies (such as customer support, personal assistants, legal, advertising, and code review agents) and discussions on trade-offs to help you tailor solutions to your needs.

What This Book Is Not

This book isn't an introduction to AI or ML basics. It assumes familiarity with concepts like neural networks, natural language processing, and basic programming in languages like Python. If you're new to these, pointers to resources are provided, but the focus is on applied agent building.

It's also not a step-by-step tutorial for specific tools, as technologies evolve rapidly. Instead, it offers guidance on evaluating and selecting tools, with pseudocode and examples to illustrate concepts. For hands-on implementation, online tutorials and documentation are recommended, including frameworks like LangChain and AutoGen.

Who This Book Is For

This book is for engineers, developers, and technical leaders aiming to build AI agent-based applications. It's geared toward roles like AI engineers, software developers, ML engineers, data scientists, and product managers with a technical bent. You might relate to scenarios like the following:

- You're tasked with building an autonomous system for decision support, or interactive services.
- You have a working agent prototype and you want to harden it and get it ready for production.
- Your team struggles with agent reliability—handling failures, adapting to dynamic environments, or orchestrating complex tasks—and you want systematic approaches including orchestration, memory, and learning from experience.
- You're integrating agents into existing workflows and seek best practices for scalability, multiagent coordination, UX design, measurement, validation, monitoring, and security.

You can also benefit if you're a tool builder identifying gaps in the agent ecosystem, a researcher exploring applications, or a job seeker preparing for AI agent roles.

Navigating This Book

The chapters follow the lifecycle of building an AI agent application, organized into three main sections.

The first three chapters cover core concepts, design principles, and essential components:

- Chapter 1 introduces agents, their promise, use cases, how they compare to traditional ML, and recent advancements.
- Chapter 2 provides an overview of designing agent systems, including scenario selection, core components (model selection, tools, memory, planning), design trade-offs, architecture patterns (single-agent, multiagent, modular), and best practices.
- Chapter 3 focuses on UX design, covering interaction modalities (text, graphical, speech, video), synchronous versus asynchronous experiences, context retention, communicating capabilities, trust, and key UX principles.

The next five chapters focus on creating, orchestrating, and scaling agents:

- Chapter 4 dives into tools, including design (local, API-based, plug-in, hierarchies) and automated tool development (code generation, imitation learning, tool learning from rewards).
- Chapter 5 covers orchestration, with fundamentals (parameterization, tool selection, execution), tool selection methods (generative, semantic, hierarchical, machine-learned), tool topologies (decomposition, single/parallel/sequential execution, chains, trees, graphs), and planning strategies (incremental execution, zero-shot, few-shot, ReAct).
- Chapter 6 explores memory, including foundational approaches (context windows, keyword-based), semantic memory and vector stores (semantic search, RAG, experience memory), GraphRAG (knowledge graphs), and working memory (whiteboards, note-taking).
- Chapter 7 addresses learning from experience, with nonparametric learning (experiences as examples, exploration/exploitation, reflection), parametric learning (fine-tuning large/small models), and transfer learning.
- Chapter 8 discusses scaling from one agent to many, including when to use multiagents, coordination (democratic, manager, hierarchical, actor-critic, automated design), and frameworks such as LangChain.

The final five chapters address validation, monitoring, security, improvement, and human-agent integration:

- Chapter 9 covers measurement and validation, with key objectives (accuracy, robustness, efficiency, etc.), evaluation sets, unit tests (tools, planning, memory, learning), integration tests (end-to-end, consistency, hallucinations), limitations, and deployment preparation.
- Chapter 10 focuses on production monitoring, including causes of failures, agent metrics (system health, automated/human evaluation, feedback), distribution shifts, and monitoring at scale (analytics, alerting, logging).

- Chapter 11 explores improvement loops, with feedback pipelines (issue detection, human review, refinement, prioritization), experimentation (shadow deployments, A/B testing, adaptive, gating), and continuous learning (in-context, offline retraining, online reinforcement).
- Chapter 12 addresses protecting agent systems, covering unique risks, securing LLMs (model selection, defenses, red teaming, fine-tuning), data protection (privacy, provenance), securing agents (safeguards, external/internal protections), and governance/compliance.
- Chapter 13 discusses humans and agents, with ethical principles (oversight, transparency, fairness, explainability, privacy), building trust/oversight, addressing bias, and accountability/regulatory considerations.

Feel free to skip sections you're familiar with—the book is modular by design.

Note: I often use "we" to refer to you (the reader) and me, fostering a collaborative learning vibe.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://oreil.ly/building-applications-with-ai-agents-supp*.

If you have a technical question or a problem using the code examples, please email *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Building Applications with AI Agents* by Michael Albada (O'Reilly). Copyright 2025 Advance AI LLC, 978-1-098-17650-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/building-applications-with-ai-agents-1e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Acknowledgments

As a first-time author, it's humbling to discover just how many people it takes to write a book, and it's thanks to the contributions of many wonderful people that this book has come to fruition.

This book took over a year to write, and I'm especially grateful to the technical reviewers who carved out their valuable time to share their detailed feedback, perspective, and insight. Nuno Campos has brought invaluable expertise on all things agents and LangChain and pointed me to concepts I had missed. Prashanth Josyula held the writing and the code examples to a high bar of technical rigor and brought deep technical expertise. Megan MacLennan has been my technical writing expert, helping ensure accessibility and relevance to a wide audience. Early drafts are always imperfect, and I'm deeply grateful to my technical reviewers for tolerating my blunders and oversights with grace. Thank you for all of your patience and your invaluable suggestions.

I also want to offer a special thanks to Anthony Wainman, who has been a thought partner from the earliest stages of this book, and offered invaluable guidance on the structure, content, examples, and so much more.

This book wouldn't have been possible without the fantastic team at O'Reilly, especially my development editor, Shira Evans, who helped shepherd the project. Many thanks to Melissa Potter for providing early feedback and reviews, and my production editors Ashley Stussy and Gregory Hyman. Nicole Butterfield has been invaluable in turning concepts into reality.

I also want to thank everyone who read the early release version of the book and offered suggestions and encouragement, including Tiago Dufau de Vargas, Jenny Song, Leonidas Askianakis, Karthik Rao, and Drew Hoskins.

I owe so much to my brilliant current and former colleagues at Microsoft, ServiceNow, and Uber, especially Olcay Cirit, Dawn Woodard, Sameera Poduri, Zoubin Ghahramani, Piero Molino, Pablo Bellver, Jaikumar Ganesh, Jay Stokes, Marc-Alexandre Cote, Chi Wang, Anush Sankaran, Amir Abdi, Tong Wang, Antonios Matakos, Max Golovanov, Abe Starosta, Francis Beckert, Malachi Jones, Taylor Black, Ryan Sweet, Lital Badash, Amir Pirogovsky, Alexander Stojanovic, Brad Sarsfield,

Chang Kawaguchi, Jure Leskovic, Chiyu Zhang, Andrew Zhao, Matthieu Lin, and many, many more. Thank you for your wisdom, your insight, your patience, your mentorship, and your many suggestions.

I would like to thank Luke Miratrix, who introduced me to statistics and taught me how to code. I would also like to thank my core academic mentors Lisa Schmitt, Lise Shelton, James Sheehan, Finbarr Livesey, Matthew Sommer, James Ward, Charles Isbell, Michael Littman, Zsolt Kira, and Constantine Dovrolis for shaping my thinking in ways big and small.

This book is, in many ways, a distillation of lessons I've learned throughout my life and career, and I am grateful to many more people than I can name here. I am deeply grateful to have the opportunity to release this book out into the world, and I truly hope it serves you well.

CHAPTER 1

Introduction to Agents

We are witnessing a profound technological transformation driven by autonomous agents—intelligent software systems capable of independent reasoning, decision making, and interacting effectively within dynamic environments. Unlike traditional software, autonomous agents interpret contexts, adapt to changing scenarios, and perform sophisticated actions with minimal human oversight.

Defining AI Agents

Autonomous agents are intelligent systems designed to independently analyze data, interpret their environment, and make context-driven decisions. As the popularity of the term "agent" grows, its meaning has become diluted, often applied to systems lacking genuine autonomy. In practice, agency exists on a spectrum. True autonomous agents demonstrate meaningful decision making, context-driven reasoning, and adaptive behaviors. Conversely, many systems labeled as "agents" may simply execute deterministic scripts or tightly controlled workflows. Designing genuinely autonomous, adaptive agents is challenging, prompting many teams to adopt simpler approaches to achieve quicker outcomes. Therefore, the key test of a true agent is whether it demonstrates real decision making rather than following static scripts.

The rapid evolution of autonomous agents is primarily driven by breakthroughs in foundation models and reinforcement learning. While traditional use cases with foundation models have focused on generating human-readable outputs, the latest advances enable these models to generate structured function signatures and parameter selections. Orchestration frameworks can then execute these functions—enabling agents to look up data, manipulate external systems, and perform concrete actions. Throughout this book, we will use the term "agentic system" to describe the full supporting functionality that enables an agent to run effectively, including the tools, memory, foundation model, orchestration, and supporting infrastructure.

With a growing range of protocols such as Model Context Protocol (discussed in Chapter 4) and Agent-to-Agent Protocol (discussed in Chapter 8), these agents will be able to use remote tools and collaborate with other agents to solve problems. This unlocks enormous opportunities for sophisticated automation—but it also brings a profound responsibility to design, measure, and manage these systems thoughtfully, ensuring their actions align with human values and operate safely in complex, dynamic environments.

The Pretraining Revolution

While traditional ML is an incredibly powerful technique, it is usually limited by the quantity and quality of the dataset. ML practitioners will typically tell you that they spend the majority of their time not training models, but on collecting and cleaning datasets that they can use for training. The incredible success of generative models that have been trained on large volumes of data have shown that single models can now adapt to a wide range of tasks without any additional training. This upends years of practice. To build an application that used ML previously required hiring an ML engineer or data scientist, having them collect data, and then deploying that model. With the latest developments in large, pretrained generative models, high-quality models that will work reasonably well for many use cases are now available through a single call to a hosted model without any training or hosting required. This dramatically lowers the cost and complexity of building applications enabled with ML and AI.

Recent advancements in large language models (LLMs) such as GPT-5, Anthropic's Claude, Meta's Llama, Google's Gemini Ultra, and DeepSeek's V3 have increased the performance on a range of difficult tasks even further, widening the scope of problems solvable with pretrained models. These foundation models offer robust natural language understanding and content generation capabilities, enhancing agent functionality through:

Natural language understanding
: Interpreting and responding intuitively to user inputs

Context-aware interaction
: Maintaining context for relevant and accurate responses over extended interactions

Structured content generation
: Producing text, code, and structured outputs essential for analytical and creative tasks

While these models are very capable on their own, they can also be used to make decisions within well-scoped areas, adapt to new information, and invoke tools to accomplish real work. Integration with sophisticated orchestration frameworks enables these models to interact directly with external systems and execute practical tasks. These models are capable of:

Contextual interpretation and decision making
: Navigating ambiguous situations without exhaustive preprogramming

Tool use
: Calling other software to retrieve information or take actions

Adaptive planning
: Planning and executing complex, multistep actions autonomously

Information summarization
: Rapidly processing extensive documents, extracting key insights, thereby aiding legal analysis, research synthesis, and content curation

Management of unstructured data
: Interpreting and responding intelligently to unstructured texts such as emails, documents, logs, and reports

Code generation
: Writing and executing code and writing unit tests

Routine task automation
: Efficiently handling repetitive activities in customer service and administrative workflows, freeing human workers to focus on more nuanced tasks

Multimodal information synthesis
: Performing intricate analyses of image, audio, or video data at scale

This enhanced flexibility enables autonomous agents to effectively handle complex and dynamic scenarios that static ML models typically cannot address.

Types of Agents

As the term "agent" has gained popularity, its meaning has broadened to encompass a wide range of AI-enabled systems, often creating confusion about what truly constitutes an AI agent. *The Information* categorizes agents into seven practical types, reflecting how these technologies are being applied today:

Business-task agents
: These agents automate predefined business workflows, such as UiPath's robotic process automation, Microsoft Power Automate's low-code flows, or Zapier's app integrations. They execute sequences of deterministic actions, typically triggered by events, with minimal contextual reasoning.

Conversational agents
: This category includes chatbots and customer service agents that engage users through natural language interfaces. They are optimized for dialogue management, intent recognition, and conversational turn-taking, such as virtual assistants embedded in customer support platforms.

Research agents
: Research agents conduct information gathering, synthesis, and summarization tasks. They scan documents, knowledge bases, or the web to provide structured outputs that assist human analysts. Examples include Perplexity AI and Elicit.

Analytics agents
: Analytics agents, such as Power BI Copilot or Glean, focus on interpreting structured datasets and generating insights, dashboards, and reports. They often integrate tightly with enterprise data warehouses, enabling users to query complex data in natural language.

Developer agents
: Tools like Cursor, Windsurf, and GitHub Copilot represent coding agents, which assist developers by generating, refactoring, and explaining code. They integrate deeply into IDE workflows to augment software development productivity.

Domain-specific agents
: These agents are tuned for specialized professional domains, such as legal (Harvey), medical (Hippocratic AI), or finance agents. They combine domain-specific knowledge with structured workflows to deliver targeted, expert-level assistance.

Browser-using agents
: These agents navigate, interact with, extract information from, and take actions on websites without human interaction. As opposed to traditional robotic process automation, which follows prescripted steps, modern browser-using agents combine language understanding, visual perception, and dynamic planning to adapt on the fly.

In addition to these seven types of agents, voice and video agents are important and also expected to increase in adoption in the coming years:

Voice agents
: Powered by end-to-end speech understanding and generation, these agents are enabling conversational automation in areas like customer service, appointment scheduling, and even real-time order processing.

Video agents
: These agents present users with avatar-based video responses, combining lip-synced speech, facial expression, and gesture. They're emerging rapidly in sales,

training, customer onboarding, marketing, and virtual presence tools—enabling scalable, personalized video interactions without manual production.

Importantly, the number and variety of agent types is growing rapidly, and we will likely see new kinds of agents emerge across many domains as the field and its underlying technologies evolve. In this book, our emphasis is on the core category of agents built around language models, particularly those using text and code. While we touch on business task automation, voice, and video, we'll primarily explore agents built around language models—their architectures, reasoning, and UX—in subsequent chapters.

Now that we've discussed the evolving types of agents, the next critical question becomes: which model should you choose to power your agent? Model selection is a complex and rapidly changing domain. As discussed in the next section, you'll need to balance factors like task complexity, modality support, latency and cost constraints, and integration requirements to make the right choice for your agent.

Model Selection

Today, we are fortunate to have a proliferation of powerful models available from both commercial providers and the open source community. OpenAI, Anthropic, Google, Meta, and DeepSeek each offer state-of-the-art foundation models with impressive general-purpose capabilities. At the same time, open-weight models like Llama, Mistral, and Gemma are pushing the boundaries of what can be achieved with local or fine-tuned deployments. Even more striking is the rapid advancement of small- and medium-sized models. New techniques for distillation, quantization, and synthetic data generation are enabling compact models to inherit surprising levels of capability from their larger counterparts.

This explosion of choice is good news: competition is driving faster innovation, better performance, and lower costs. But it also creates a dilemma—how do you choose the right model for your agentic system? The truth is, there isn't a one-size-fits-all answer. In fact, one of the most reasonable starting points is simply to use the latest general-purpose model from a leading provider like OpenAI or Anthropic. As you can see in Table 1-1, these models offer strong performance out of the box, require little customization, and will take you surprisingly far for many applications. GPT-5 mini (Aug 2025) leads overall with the highest mean score (0.819), closely followed by o4-mini (0.812) and o3 (0.811). Proprietary and open-access models like Qwen3, Grok 4, Claude 4, and Kimi K2 also show competitive results.

Table 1-1. HELM Core Scenario leaderboard (August 2025). Comparative benchmark performance of the top 10 models across reasoning and evaluation tasks: MMLU-Pro, GPQA, IFEval, WildBench, and Omni-MATH.

Model	Mean score	MMLU-Pro—COT correct	GPQA—COT correct	IFEval—IFEval Strict Acc	WildBench—WB Score	Omni-MATH—Acc
GPT-5 mini (2025-08-07)	0.819	0.835	0.756	0.927	0.855	0.722
o4-mini (2025-04-16)	0.812	0.82	0.735	0.929	0.854	0.72
o3 (2025-04-16)	0.811	0.859	0.753	0.869	0.861	0.714
GPT-5 (2025-08-07)	0.807	0.863	0.791	0.875	0.857	0.647
Qwen3 235B A22B Instruct 2507 FP8	0.798	0.844	0.726	0.835	0.866	0.718
Grok 4 (0709)	0.785	0.851	0.726	0.949	0.797	0.603
Claude 4 Opus (20250514, extended thinking)	0.78	0.875	0.709	0.849	0.852	0.616
gpt-oss-120b	0.77	0.795	0.684	0.836	0.845	0.688
Kimi K2 Instruct	0.768	0.819	0.652	0.85	0.862	0.654
Claude 4 Sonnet (20250514, extended thinking)	0.766	0.843	0.706	0.84	0.838	0.602

That said, they aren't always the most efficient choice. For many tasks—especially those that are well-defined, low-latency, or cost-sensitive—much smaller models can provide near-equivalent performance at a fraction of the cost. This has led to a growing trend: automated model selection. Some platforms now route simpler queries to fast, inexpensive small models, reserving the large, expensive models for more complex reasoning. This dynamic test-time optimization is proving effective, and it hints at a future where multimodel systems become the norm.

The key takeaway is that you can spend enormous effort optimizing model selection for marginal gains—but unless your scale or constraints demand it, starting simple is fine. Over time, it's often worth experimenting with smaller models, fine-tuning, or adding retrieval to improve performance and reduce costs. Just remember: the future is almost certainly multimodel, and designing for flexibility now will pay off later.

From Synchronous to Asynchronous Operations

Traditional software systems typically execute tasks synchronously, moving step-by-step and waiting for each action to finish before starting the next. While this approach is straightforward, it can lead to significant inefficiencies—especially when waiting on external inputs or processing large volumes of data.

In contrast, autonomous agents are designed for asynchronous operation. They can manage multiple tasks in parallel, swiftly adapt to new information, and prioritize actions dynamically based on changing conditions. This asynchronous processing dramatically enhances efficiency, reducing idle time and optimizing the use of computational resources.

The practical implications of this shift are substantial. For example:

- Emails can arrive with reply drafts already prepared.
- Invoices can come with pre-populated payment details.
- Software engineers might receive tickets accompanied by code to solve them and unit tests to assess them.
- Customer support agents can be provided with suggested responses and recommended actions.
- Security analysts can receive alerts that have already been automatically investigated and enriched with relevant threat intelligence.

In each case, agents are not just speeding up routine workflows—they are changing the nature of work itself. This evolution transforms human roles from task executors to task managers. Rather than spending time on repetitive or mechanical steps, individuals can focus on strategic oversight, review, and high-value decision making—amplifying human creativity and judgment while letting agents handle the operational details. These agents make it much easier for human roles to be proactive rather than reactive.

Practical Applications and Use Cases

The versatility of autonomous agents opens up a myriad of applications across different industries. To keep this book grounded in clear, specific use cases, I have seven real-world example agents with evaluation systems available in the public GitHub repo supporting this book. We will frequently turn back to these examples as we explore the key aspects of agent systems:

Customer support agent
: Customer support is one of the most prevalent applications for autonomous agents. These agents handle common inquiries, process refunds, update orders, and escalate complex issues to human representatives, providing 24/7 support while enhancing customer satisfaction and reducing operational costs.

Financial services agent
: In banking and financial services, agents assist with account management, loan processing, fraud investigation, and investment portfolio rebalancing. They streamline customer service, accelerate transaction processing, and improve security by detecting suspicious activities in real time.

Healthcare patient intake and triage agent

These agents support frontline healthcare operations by registering new patients, verifying insurance, assessing symptoms to prioritize care, scheduling appointments, managing medical histories, and coordinating referrals, thereby improving workflow efficiency and patient outcomes.

IT help desk agent

IT help desk agents manage user access, troubleshoot network and system issues, deploy software updates, respond to security incidents, and escalate unresolved issues to specialists. They enhance productivity by resolving common technical problems swiftly.

Legal document review agent

Legal agents assist attorneys and paralegals by reviewing contracts, conducting legal research, performing client intake and conflict checks, managing discovery, assessing compliance, calculating damages, and tracking deadlines. This helps to streamline workflows and improve accuracy in legal operations.

Security Operations Center (SOC) analyst agent

SOC analyst agents investigate security alerts, gather threat intelligence, query logs, triage incidents, isolate compromised hosts, and provide updates to security teams. They accelerate incident response and strengthen organizational security posture.

Supply chain and logistics agent

In supply chain management, agents optimize inventory, track shipments, evaluate suppliers, coordinate warehouse operations, forecast demand, manage disruptions, and handle compliance requirements. These capabilities help maintain resilience and efficiency across global networks.

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.

Workflows and Agents

In many real-world projects, choosing between a simple script, a deterministic workflow, a traditional chatbot, a retrieval-augmented generation (RAG) system, or a full-blown autonomous agent can be the difference between an elegant solution and an overengineered, hard-to-maintain mess. To make this choice clearer, consider four key factors: the variability of your inputs, the complexity of the reasoning required, any performance or compliance constraints, and the ongoing maintenance burden.

First, when might you choose not to use a foundation model—or any ML component at all? If your inputs are fully predictable and every possible output can be described in advance, a handful of lines of procedural code are often faster, cheaper, and far easier to test than an ML–based pipeline. For example, parsing a log file that always follows the format "YYYY-MM-DD HH:MM:SS—message" can be handled reliably with a small regular-expression-based parser in Python or Go. Likewise, if your application demands millisecond-level latency—such as an embedded system that must react to sensor data in real time—there simply isn't time for a language model API call. In such cases, traditional code is the right choice. Finally, regulated domains (medical devices, aeronautics, certain financial systems) often require fully deterministic, auditable decision logic—black-box neural models won't satisfy certification requirements. If any of these conditions hold—deterministic inputs, strict performance or explainability needs, or a static problem domain—plain code is almost always preferable to a foundation model.

Next, consider deterministic or semiautomated workflows. Here, the logic can be expressed as a finite set of steps or branches, and you know ahead of time where you might need human intervention or extra error handling. Suppose you ingest invoices from a small set of vendors and each invoice arrives in one of three known formats: CSV, JSON, or PDF. You can build a workflow that routes each format to its corresponding parser, checks for mismatches, and halts for a human review if any fields fail a simple reconciliation—no deep semantic understanding is required. Likewise, if your system must retry failed steps with exponential backoff or pause for a manager's approval, a workflow engine (such as Airflow, AWS Step Functions, or a well-structured set of scripts) offers clearer control over error paths than an LLM could. Deterministic workflows make sense whenever you can enumerate all decision branches in advance and you need tight, auditable control over each branch. In such scenarios, workflows scale more naturally than large, ad hoc scripts but still avoid the complexity and cost of running an agentic pipeline.

Traditional chatbots or RAG systems occupy the next tier of complexity: they add natural language understanding and document retrieval but stop short of autonomous, multistep planning. If your primary need is to let users ask questions about a knowledge base—say, searching a product manual, a legal archive, or corporate wikis—a RAG system can embed documents into a vector store, retrieve relevant passages in response to a query, and generate coherent, context-aware answers. For instance, an internal IT help desk might use RAG to answer "How do I reset my VPN credentials?" by fetching the latest troubleshooting guide and summarizing the relevant steps. Unlike autonomous agents, RAG systems do not independently decide on follow-up actions (like filing a ticket or scheduling a callback); they simply surface information. A traditional chatbot or RAG approach makes sense when the task is primarily question-answering over structured or unstructured content, with limited need for external API calls or decision orchestration. Maintenance costs are lower than for agents—your main overhead lies in keeping document embeddings up to date and refining prompts—but you sacrifice the agent's ability to plan multistep workflows or learn from feedback loops.

Finally, we reach autonomous agents—situations where neither simple code, nor rigid workflows, nor RAG suffice because inputs are unstructured, novel, or highly variable, and because you require dynamic, multistep planning or continuous learning from feedback. Consider a customer support center that receives free-form emails with issues ranging from "my laptop battery is swelling and might erupt" to "I keep getting billed for services I didn't order." A rule-based workflow or a RAG-powered FAQ lookup would shatter under such open-ended variety, but an agent powered by a foundation model can parse intent, extract relevant entities, consult a knowledge base, draft an appropriate response, and even escalate to a human if necessary—all without being told every possible branch in advance. Similarly, in supply chain management, an agent that ingests real-time inventory data, supplier lead times, and sales forecasts can replan shipment schedules dynamically; a deterministic workflow would require constant manual updates to handle new exceptions.

Agents also excel when many subtasks must run in parallel—such as a security operations agent that simultaneously queries threat intelligence APIs, scans network telemetry, and performs sandbox analysis on suspicious binaries. Because agents operate asynchronously and reprioritize based on real-time data, they avoid the brittle "one-step-at-a-time" nature of workflows or RAG systems. To justify the higher compute and maintenance costs of running a foundation model, you need this level of contextual reasoning, parallel task orchestration, or ongoing self-improvement—scenarios where rigid code, workflows, or chatbots would be too brittle or expensive to maintain.

Table 1-2. Distinguishing workflows and agents from traditional code

Characteristic	Traditional code	Workflow	Autonomous agent
Input structure	Fully predictable schemas	Mostly predictable with finite branches	Highly unstructured or novel inputs
Explainability	Full transparency; easily auditable	Explicit branch-by-branch audit trail	Black-box components requiring additional tooling
Latency	Ultra-low latency	Moderate latency	Higher latency
Adaptability and learning	None	Limited	High (learning from feedback)

Every path carries trade-offs. Pure code is cheap and fast but inflexible; workflows offer control but break down when inputs grow wildly variable; traditional chatbots or RAG are great for question-answering over documents but cannot orchestrate multistep actions; and agents are powerful but demanding—both in terms of cloud compute and engineering effort to monitor, tune, and govern. Before choosing, ask: are my inputs unstructured or unpredictable? Do I need multistep planning that adapts to intermediate results? Can a document retrieval system suffice for my users' information needs, or must the system decide and act autonomously? Will I want this system to improve itself over time with minimal human intervention? And can I tolerate the latency and maintenance burden of a foundation model?

In short, if your task is a fixed, deterministic transformation, write some simple code. If there are a handful of known branches and you require explicit error-handling checkpoints, use a deterministic workflow. If you primarily need natural language question-answering over a corpus, choose a traditional chatbot or RAG architecture. But if you face high variability, open-ended reasoning, dynamic planning needs, or continual learning requirements, invest in an autonomous agent. Making this choice thoughtfully ensures that you get the right balance of simplicity, performance, and adaptability—so your solution remains both effective and maintainable as requirements evolve.

Principles for Building Effective Agentic Systems

Creating successful autonomous agents requires an approach that prioritizes scalability, modularity, continuous learning, resilience, and future-proofing:

Scalability
: Ensure that agents can handle growing workloads and diverse tasks by utilizing distributed architectures, cloud-based infrastructure, and efficient algorithms that support parallel processing and resource optimization. Example: a customer support agent that processes 10 tickets per minute may crash or hang when traffic spikes to 1,000 if not backed by autoscaling infrastructure.

Modularity
: Design agents with independent, interchangeable components connected through clear interfaces. This modular approach simplifies maintenance, promotes flexibility, and facilitates rapid adaptation to new requirements or technologies. Example: a poorly modular agent that hardcodes all its tools in its agent service would require a full redeployment anytime a small addition or modification is needed to a tool.

Continuous learning
: Equip agents with mechanisms to learn from experience, such as in-context learning. Integrate user feedback to refine agent behaviors and maintain

performance relevance as tasks evolve. Example: agents that ignore feedback loops may keep making the same mistakes—like misclassifying contract clauses or failing to escalate critical support issues.

Resilience
: Develop robust resilience architectures capable of gracefully handling errors, security threats, timeouts, and unexpected conditions. Incorporate comprehensive error handling, stringent security measures, and redundancy to ensure reliable and continuous agent operations. Example: agents without retry or fallback logic may crash entirely when a single API call fails, leaving the user waiting and confused.

Future-proofing
: Build agent systems around open standards and scalable infrastructure, fostering a culture of innovation to adapt quickly to emerging technologies and evolving user expectations. Example: tightly coupling your agent to one proprietary vendor's prompt format can make switching models painful and limit experimentation.

Adhering to these principles enables organizations to develop autonomous agents that remain effective and relevant, adapting seamlessly to technological advancements and changing operational environments.

Organizing for Success in Building Agentic Systems

The widespread availability of foundation models via simple API calls has spurred extensive experimentation with agent systems across many organizations. Teams frequently embark on independent proofs of concept, leading to valuable discoveries and innovative ideas. However, this ease of experimentation often results in fragmentation—overlapping projects, duplicated efforts, and unfinished experiments become scattered throughout the organization. Conversely, premature standardization could stifle creativity and trap organizations into rigid frameworks or vendor-specific solutions. Achieving success requires balancing flexibility for experimentation with sufficient alignment for scalability and coherence.

In the early phases of agent development, organizations should actively encourage exploratory efforts, permitting teams to test various architectures, workflows, and models freely. Over time, as successful patterns and best practices become apparent, strategic alignment becomes critical. Implementing a "one standard per large group" strategy can effectively balance this need. Within specific departments or functional areas, teams can standardize around common tools and methodologies, streamlining collaboration without restricting broader organizational innovation.

Another essential aspect of success is avoiding vendor lock-in by adopting open standards, such as OpenAPI, and embracing modular system designs. These practices help ensure flexibility and reduce dependency on any single technology or provider, facilitating future adaptability.

Effective knowledge sharing is also crucial. Lessons learned from both successful and unsuccessful experiments should be communicated widely via internal forums, shared repositories, and comprehensive documentation. This collaborative approach accelerates organizational learning, minimizes redundant efforts, and promotes collective improvement.

Lastly, governance frameworks should remain lightweight and flexible, emphasizing guiding principles over rigid mandates. A streamlined governance structure enables teams to innovate confidently while remaining aligned with overarching organizational objectives.

Organizing successfully around agentic systems is fundamentally iterative. Organizations must continually reassess their strategies to maintain a dynamic balance between exploration and standardization. By cultivating an environment that values experimentation, collaborative learning, and open standards, organizations can effectively transition agentic systems from isolated experiments into scalable, transformative solutions that are deeply integrated into their operational processes.

Agentic Frameworks

Numerous frameworks currently exist for developing autonomous agents, each addressing critical functionalities such as skills integration, memory management, planning, orchestration, experiential learning, and multiagent coordination. This list is certainly not exhaustive, but leading frameworks include the following.

LangGraph

Strengths
: Modular orchestration framework based on directed graphs whose nodes contain discrete units of logic (often foundation model calls) and whose edges manage the flow of data through complex, potentially cyclic workflows; strong developer ergonomics; native support for asynchronous workflows and retries

Trade-offs
: Requires custom logic for advanced planning and memory; less built-in support for multiagent collaboration

Best for
: Teams building robust, single-agent or light multiagent systems with explicit, inspectable flow control

AutoGen

Strengths
: Powerful multiagent orchestration; dynamic role assignment; flexible messaging-based interaction between agents

Trade-offs
: Can be heavyweight or complex for simple use cases; more opinionated around agent interaction patterns

Best for
: Research and production systems involving dialogue between multiple agents (e.g., manager-worker, self-reflection loops)

CrewAI

Strengths
: Easy to learn and use; quick setup for prototyping; useful abstractions like “crew” and “tasks”

Trade-offs
: Limited customization and control over orchestration internals; less mature than LangGraph or AutoGen for complex workflows

Best for
: Developers who want to get started quickly on practical, human-centric agents like assistants or support agents

OpenAI Agents Software Development Kit (SDK)

Strengths
: Deep integration with OpenAI’s tool ecosystem; secure and easy-to-use function calling, memory primitives, and tool routing

Trade-offs
: Tightly coupled to OpenAI’s infrastructure; may be less flexible or portable for custom agent stacks or open source toolchains

Best for
: Teams already using the OpenAI API and looking for a fast way to build secure, tool-using agents with minimal scaffolding

While each framework offers unique advantages and limitations, continuous innovation and competition in this space are expected to drive further evolution. For early prototypes, CrewAI or OpenAI Agents SDK can get you running quickly. For scalable, production-grade systems, LangGraph and AutoGen provide more control and sophistication. These frameworks are also not necessary, and many teams choose to build directly against the model provider APIs. This book primarily focuses on LangGraph, chosen for its straightforward yet powerful approach to agent system development. Through detailed explanations, practical examples, and real-world scenarios, we demonstrate how LangGraph effectively addresses the complexity and dynamics required by modern intelligent agents.

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 ML 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.

CHAPTER 2

Designing Agent Systems

Most practitioners don't begin with a grand design document when building agent systems. They start with a messy problem, a foundation model API key, and a rough idea of what might help. This chapter is your quick start to get you up and running. We'll cover each of the following topics in more depth through the rest of the book, and many will get their own chapter, but this chapter will give you an overview of how to design an agentic system, all grounded in a specific example of managing customer support for an ecommerce platform.

Our First Agent System

Let's start with the problem we're solving. Every day, your customer-support team fields dozens or hundreds of emails asking to refund a broken mug, cancel an unshipped order, or change a delivery address. For each message, a human agent has to read free-form text, look up the order in your backend, call the appropriate API, and then type a confirmation email. This repetitive two-minute process is ripe for automation—but only if we carve off the right slice. When we realize that humans type keys and click buttons, often following rules and guidelines, we see that many of these same patterns can be performed by well-designed systems that rely on foundation models. We want our agent to take a raw customer message plus the order record, decide which tool to call (`issue_refund`, `cancel_order`, or `update_address_for_order`), invoke that tool with the correct parameters, and then send a brief confirmation message. That two-step workflow is narrow enough to build quickly, valuable enough to free up human time, and rich enough to showcase intelligent behavior. We can build a working agent for this use case in just a few lines of code:

```
from langchain.tools import tool
from langchain_openai.chat_models import ChatOpenAI
from langchain.schema import SystemMessage, HumanMessage, AIMessage
from langchain_core.messages.tool import ToolMessage
from langgraph.graph import StateGraph

# -- 1) Define our single business tool
@tool
def cancel_order(order_id: str) -> str:
    """Cancel an order that hasn't shipped."""
    # (Here you'd call your real backend API)
    return f"Order {order_id} has been cancelled."

# -- 2) The agent "brain": invoke LLM, run tool, then invoke LLM again
def call_model(state):
    msgs = state["messages"]
    order = state.get("order", {"order_id": "UNKNOWN"})

    # System prompt tells the model exactly what to do
    prompt = (
        f'''You are an ecommerce support agent.
        ORDER ID: {order['order_id']}
        If the customer asks to cancel, call cancel_order(order_id)
        and then send a simple confirmation.
        Otherwise, just respond normally.'''
    )
    full = [SystemMessage(prompt)] + msgs

    # 1st LLM pass: decides whether to call our tool
    AIMessage = ChatOpenAI(model="gpt-5", temperature=0)(full)
    out = [first]

    if getattr(first, "tool_calls", None):
        # run the cancel_order tool
        tc = first.tool_calls[0]
        result = cancel_order(**tc["args"])
        out.append(ToolMessage(content=result, tool_call_id=tc["id"]))

        # 2nd LLM pass: generate the final confirmation text
        AIMessage = ChatOpenAI(model="gpt-5", temperature=0)(full + out)
        out.append(second)

    return {"messages": out}

# -- 3) Wire it all up in a StateGraph
def construct_graph():
    g = StateGraph({"order": None, "messages": []})
    g.add_node("assistant", call_model)
    g.set_entry_point("assistant")
    return g.compile()

graph = construct_graph()

if __name__ == "__main__":
```

```
example_order = {"order_id": "A12345"}
convo = [HumanMessage(content="Please cancel my order A12345.")]
result = graph.invoke({"order": example_order, "messages": convo})
for msg in result["messages"]:
    print(f"{msg.type}: {msg.content}")
```

Great—you now have a working "cancel order" agent. Before we expand our agent, let's reflect on *why* we started with such a simple slice. Scoping is always a balancing act. If you narrow your task too much—say, only cancellations—you miss out on other high-volume requests like refunds or address changes, limiting real-world impact. But if you broaden it too far—"automate every support inquiry"—you'll drown in edge cases like billing disputes, product recommendations, and technical troubleshooting. And if you keep it vague—"improve customer satisfaction"—you'll never know when you've succeeded.

Instead, by focusing on a clear, bounded workflow—canceling orders—we ensure concrete inputs (customer message + order record), structured outputs (tool calls + confirmations), and a tight feedback loop. For example, imagine an email that says, "Please cancel my order #B73973 because I found a cheaper option elsewhere." A human agent would look up the order, verify it hasn't shipped, click "Cancel," and reply with a confirmation. Translating this into code means invoking `cancel_order(order_id="B73973")` and sending a simple confirmation message back to the customer.

Now that we have a working "cancel order" agent, the next question is: does it actually work? In production, we don't just want our agent to run—we want to know how well it performs, what it gets right, and where it fails. For our cancel order agent, we care about questions like:

- Did it call the correct tool (`cancel_order`)?
- Did it pass the right parameters (the correct order ID)?
- Did it send a clear, correct confirmation message to the customer?

In our open source repository, you'll find a full evaluation script to automate this process:

- Evaluation dataset
- Batch evaluation script

Here's a minimal, simplified version of this logic for how you might test your agent directly:

```
# Minimal evaluation check
example_order = {"order_id": "B73973"}
convo = [HumanMessage(content='''Please cancel order #B73973.
    I found a cheaper option elsewhere.''')]
```

```
result = graph.invoke({"order": example_order, "messages": convo})

assert any("cancel_order" in str(m.content) for m in result["messages"],
    "Cancel order tool not called")
assert any("cancelled" in m.content.lower() for m in result["messages"],
    "Confirmation message missing")

print("✅ Agent passed minimal evaluation.")
```

This snippet ensures that the tool was called and the confirmation was sent. Of course, real evaluation goes deeper: you can measure tool precision, parameter accuracy, and overall task success rates across hundreds of examples to catch edge cases before deploying. We'll dive into evaluation strategies and frameworks in depth in Chapter 9, but for now, remember: an untested agent is an untrusted agent.

Because both steps are automated using `@tool` decorators, writing tests against real tickets becomes trivial—and you instantly gain measurable metrics like tool recall, parameter accuracy, and confirmation quality. Now that we've built and evaluated a minimal agent, let's explore the core design decisions that will shape its capabilities and impact.

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 tools, 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—the foundation model, tools, and memory—and explores how they interact to form a cohesive agent system. Figure 2-1 shows the core components of an agent system.

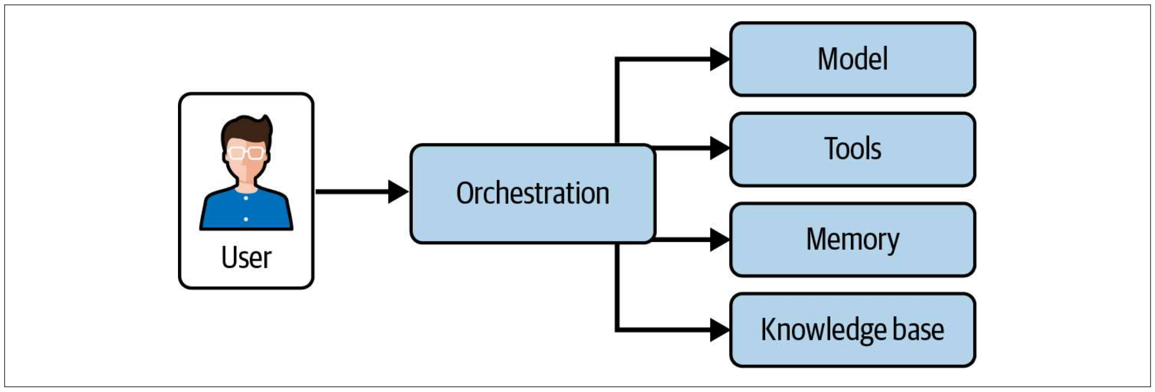


Figure 2-1. Core components of an agent system.

Model Selection

At the heart of every agent-based system lies the model that drives the agent's decision-making, interaction, and learning capabilities. Selecting the right model is foundational: it determines how the agent interprets inputs, generates outputs, and adapts to its environment. This decision influences the system's performance, scalability, latency, and cost. Choosing an appropriate model depends on the complexity of the agent's tasks, the nature of the input data, infrastructure constraints, and the trade-offs between generality, speed, and precision.

Broadly speaking, model selection starts with assessing task complexity. Large foundation models—such as GPT-5 or Claude Opus 4.1—are well suited for agents operating in open-ended environments, where nuanced understanding, flexible reasoning, and creative generation are essential. These models offer impressive generalization and excel at tasks involving ambiguity, contextual nuance, or multiple steps. However, their strengths come at a cost: they require significant computational resources, often demand cloud infrastructure, and introduce higher latency. They are best reserved for applications like personal assistants, research agents, or enterprise systems that must handle a wide range of unpredictable queries.

In contrast, smaller models—such as distilled ModernBERT variants or Phi-4—are often more appropriate for agents performing well-defined, repetitive tasks. These models run efficiently on local hardware, respond quickly, and are less expensive to deploy and maintain. They work well in structured settings like customer support, information retrieval, or data labeling, where precision is needed but creativity and flexibility are less important. When real-time responsiveness or resource constraints are critical, smaller models may outperform their larger counterparts simply by being more practical.

An increasingly important dimension in model selection is modality. Agents today often need to process not just text, but also images, audio, or structured data. Multimodal models, such as GPT-5 and Claude 4.1, enable agents to interpret and combine diverse data types—text, visuals, speech, and more. This expands the agent's utility in domains like healthcare, robotics, and customer support, where decisions rely on integrating multiple forms of input. In contrast, text-only models remain ideal for purely language-driven use cases, offering lower complexity and faster inference in scenarios where additional modalities provide little added value.

Another key consideration is openness and customizability. Open source models, such as Llama and DeepSeek, provide developers with full transparency and the ability to fine-tune or modify the model as needed. This flexibility is particularly important for privacy-sensitive, regulated, or domain-specific applications. Open source models can be hosted on private infrastructure, tailored to unique use cases, and deployed without licensing costs—though they do require more engineering overhead. By contrast, proprietary models like GPT-5, Claude, and Cohere offer powerful capabilities via API and come with managed infrastructure, monitoring, and performance optimizations. These models are ideal for teams seeking rapid development and deployment, though customization is often limited and costs can scale quickly with usage.

The choice between using a pretrained general-purpose model or a custom-trained model depends on the specificity and stakes of the agent's domain. Pretrained models—trained on broad internet-scale corpora—work well for general language tasks, rapid prototyping, and scenarios where domain precision is not critical. These models can often be lightly fine-tuned or adapted through prompting techniques to achieve strong performance with minimal effort. However, in specialized domains—such as medicine, law, or technical support—custom-trained models can provide significant advantages. By training on curated, domain-specific datasets, developers can endow agents with deeper expertise and contextual understanding, leading to more accurate and trustworthy outputs.

Cost and latency considerations often tip the scales in real-world deployments. Large models deliver high performance but are expensive to run and may introduce response delays. In cases where that is untenable, smaller models or compressed versions of larger models provide a better balance. Many developers adopt hybrid strategies, where a powerful model handles the most complex queries and a lightweight model handles routine tasks. In some systems, dynamic model routing ensures that each request is evaluated and routed to the most appropriate model based on complexity or urgency—enabling systems to optimize both cost and quality.

The Center for Research on Foundation Models at Stanford University has released the Holistic Evaluation of Language Models, providing rigorous third-party performance measurement across a wide range of models. In Table 2-1, a small selection of language models are shown along with their performance on the Massive Multitask Language Understanding (MMLU) benchmark, a commonly used general assessment of these models' abilities. These measurements are not perfect, but they provide us with a common ruler with which to compare performance. In general, we see that larger models perform better, but inconsistently (some models perform better than their size would suggest). Significantly more computation resources are required to obtain high performance.

Table 2-1. Selected open weight models by performance and size

Model	Maintainer	MMLU	Parameters (billion)	VRAM (full precision model in GB)	Sample hardware required
Llama 3.1 Instruct Turbo	Meta	56.1	8	20	RTX 3090
Gemma 2	Google	72.1	9	22.5	RTX 3090

Model	Maintainer	MMLU	Parameters (billion)	VRAM (full precision model in GB)	Sample hardware required
NeMo	Mistral	65.3	12	24	RTX 3090
Phi-3	Microsoft	77.5	14.7	29.4	A100
Qwen1.5	Alibaba	74.4	32	60.11	A100
Llama 3	Meta	79.3	70	160	4xA100

Conversely, this means moderate performance can be obtained at a small fraction of the cost. As you'll see in Table 2-1, models up to roughly 14 billion parameters can be run on a single consumer-grade graphics processing unit (GPU), such as NVIDIA's RTX 3090 with 24 GB of video RAM. Above this threshold, though, you will probably want a server-grade GPU such as NVIDIA's A100, which comes in 40 GB and 80 GB varieties. Models are called "open weight" when the architecture and weights (or parameters) of the model have been released freely to the public, so anyone with the necessary hardware can load and use the model for inference without paying for access. We will not get into the details of hardware selection, but these select open weight models show a range of performance levels at different sizes. These small, open weight models continue to improve at a rapid pace, bringing increasing amounts of intelligence into smaller form factors. While they might not work well for your hardest problems, they can handle easier, more routine tasks at a fraction of the price. For our example ecommerce support agent, a small fast model suffices—but if we expanded into product recommendations or sentiment-based escalation, a larger model could unlock new capabilities.

Now let's take a look at several of the large flagship models. Note that two of these models, DeepSeek-v3 and Llama 3.1 Instruct Turbo 405B, have been released as open weight models but the others have not. That said, these large models typically require at least 12 GPUs for reasonable performance, but they can require many more. These large models are almost always used on servers in large data centers. Typically, the model trainers charge for access to these models based on the number of input and output tokens. The advantage of this is that the developer does not need to worry about servers and GPU utilization but can begin building right away. Table 2-2 shows the model costs and performance on the same MMLU benchmark.

Table 2-2. Selected large models by performance and cost

Model	Maintainer	MMLU	Relative price per million input tokens	Relative price per million output tokens
DeepSeek-v3	DeepSeek	87.2	2.75	3.65
Claude 4 Opus Extended Thinking	Anthropic	86.5	75	125
Gemini 2.5 Pro	Google	86.2	12.5	25
Llama 3.1 Instruct Turbo 405B	Meta	84.5	1	1
o4-mini	OpenAI	83.2	5.5	7.33

Model	Maintainer	MMLU	Relative price per million input tokens	Relative price per million output tokens
Grok 3	xAI	79.9	15	25
Nova Pro	Amazon	82.0	4	5.33
Mistral Large 2	Mistral	80.0	10	10

In Table 2-2, prices are shown as a multiple of the price per million tokens on Llama 3.1, which was the least expensive at the time of publishing. At the time of publishing, Meta is charging $0.20 per million input tokens and $0.60 per million output tokens. You might also notice that performance does not directly correlate to price. Also know that performance on benchmarks offers useful guidance, but your mileage may vary in how these benchmarks align with your particular task. When possible, compare the model for your task and find the model that provides you with the best price per performance.

Ultimately, model selection is not a onetime decision but a strategic design choice that must be revisited as agent capabilities, user needs, and infrastructure evolve. Developers must weigh trade-offs between generality and specialization, performance and cost, simplicity and extensibility. By carefully considering the task complexity, input modalities, operational constraints, and customization needs, teams can choose models that enable their agents to act efficiently, scale reliably, and perform with precision in the real world.

Tools

In agent-based systems, *tools* are the fundamental capabilities that enable agents to perform specific actions or solve problems. Tools 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 tools.

Designing Capabilities for Specific Tasks

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

Local tools

These are actions that the agent performs based on internal logic and computations without external dependencies. Local tools 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 tools
: API-based tools enable agents to interact with external services or data sources. These tools enable 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.

Model Context Protocol (MCP)
: MCP-based tools enable agents to provide structured, real-time context to language models using the Model Context Protocol, a standardized schema for passing external knowledge, memory, and state into the model's prompt. Unlike traditional API calls that require full round-trip execution, MCP enables agents to inject rich, dynamic context—such as user profiles, conversation history, world state, or task-specific metadata—directly into the model's reasoning process without invoking separate tools. They are particularly effective in reducing redundant tool use, preserving conversational state, and injecting real-time situational awareness into model behavior.

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

Tool Integration and Modularity

Modular design is critical for tool development. Each tool should be designed as a self-contained module that can be easily integrated or replaced as needed. This approach enables 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 tools for handling simple queries and later have more complex tools (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, enabling 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. We'll discuss memory in much more detail in Chapter 6.

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 enable 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, enabling 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 enable 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, enabling 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 ecommerce 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.

Orchestration

Orchestration is what turns isolated capabilities into end-to-end solutions: it's the logic that composes, schedules, and supervises a series of skills so that each action flows into the next and works toward a clear objective. At its core, orchestration evaluates possible sequences of tool or skill invocations, forecasts their likely outcomes, and picks the path most likely to succeed in multistep tasks—whether that's plotting an optimal delivery route that balances traffic, time windows, and vehicle availability, or assembling a complex data-processing pipeline.

Because real-world conditions can change in an instant—new information arrives, priorities shift, or resources become unavailable—an orchestrator must continuously monitor both progress and environment, pausing or rerouting workflows as needed to stay on course. In many scenarios, agents build plans incrementally: they execute a handful of steps, then reassess and update the remaining workflow based on fresh results. A conversational assistant, for example, might confirm each subtask's outcome before planning the next, dynamically adapting its sequence to ensure responsiveness and robustness.

Without a solid orchestration layer, even the most powerful skills risk running at cross-purposes or stalling entirely. We'll dig into the patterns, architectures, and best practices for building resilient, flexible orchestration engines in Chapter 5.

Design Trade-Offs

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

Performance: Speed/Accuracy Trade-Offs

A key trade-off 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, it becomes critical to manage computational resources, particularly GPUs. 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 enables 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, giving 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 for ensuring 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 enables GPU tasks to be processed in parallel without waiting for previous tasks to be completed, 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, enabling the system to meet growing demands efficiently.

To scale GPU-intensive systems effectively, it 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 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 trade-offs 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 enable agents to learn from their environment and improve performance over time, increasing their robustness.

Costs: Balancing Performance and Expense

Cost is an often-overlooked but critical trade-off 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 (ML) 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, ML 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 versus 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 up-front infrastructure costs, establishing a more scalable, pay-as-you-go model.

Open source models and tools
: Utilizing open source ML libraries and frameworks can help minimize software development costs while still delivering high-quality agents.

Designing agent systems involves balancing several critical trade-offs. Prioritizing performance may require sacrificing some accuracy, while scaling to a multiagent 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 multiagent architectures—and discusses their advantages, challenges, and appropriate use cases. We'll discuss this in far more detail in Chapter 8.

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 and narrow tasks, this architecture is best suited for workloads that are 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. This makes them a fit for customer service chatbots, general-purpose assistants, and code generation agents. We'll discuss single-agent and multiagent architectures much more in Chapter 8.

Multiagent Architectures: Collaboration, Parallelism, and Coordination

In multiagent 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. Multiagent 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 multiagent 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
Multiagent 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 multiagent systems highly scalable and capable of managing larger and more complex environments.

Redundancy and resilience
Because 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, multiagent 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, multiagent systems can become disorganized and inefficient.

Increased complexity
While multiagent 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, multiagent 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 with 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 multiagent systems offer robust solutions for complex tasks, their efficiency challenges mean that careful resource management is crucial.

Multiagent 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. Multiagent 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*, *evaluation strategy*, and *real-world testing*—that contribute to creating adaptable, efficient, and reliable agent systems.

Iterative Design

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 it 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 enables 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 enables 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:

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.

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.

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.

Evaluation Strategy

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. We'll cover measurement and validation in far more depth in Chapter 9.

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), ensuring the system meets the domain's accuracy and compliance requirements

For agent systems, particularly those powered by ML 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
: Create 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

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 enables 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 for ensuring 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 enables 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 ecommerce 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:

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.

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.

Collect user feedback
: Engage users during real-world testing to gather feedback on their experiences when interacting with the agent. User feedback is invaluable in identifying gaps, improving usability, and ensuring that the agent meets real-world needs.

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

You don't need a 30-page plan to start building a good agent system—but a little foresight goes a long way. As we saw with our ecommerce support agent, picking a tractable slice—like canceling orders—lets you build something small, testable, and immediately useful. Define what success looks like, avoid vague or over-scoped ambitions, and focus on delivering clear value quickly.

Effective agent systems are more than a sum of their parts. They depend on strong architecture, disciplined engineering, and tight feedback loops. Choosing the right structural pattern sets the stage for scalability and resilience, while iterative development and robust evaluation ensure your agents improve over time. Best practices like phased rollouts and real-world testing turn promising prototypes—like our simple cancel order agent—into reliable systems that can be trusted in production.

In Chapter 3, we shift focus to the human side of the equation—how to design agent experiences that are clear, responsive, and intuitive for the people who rely on them. Ultimately, no matter how powerful your system architecture, its success depends on how it lands in human hands.

CHAPTER 3

User Experience Design for Agentic Systems

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. The field of agent UX is evolving at an unprecedented pace. New interface paradigms, modality combinations, and user interaction models are emerging almost monthly. This chapter provides foundational design principles that remain relevant even as the specific technologies and capabilities continue to advance rapidly. 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.

Table 3-1. Placeholder

Modality	Prevalence	Example use cases	Ideal situations
Text	Very common	Customer service chatbots, productivity assistants	When clear, asynchronous, or searchable communication is needed
Graphical user interfaces (GUI)	Common	Workflow orchestration dashboards, AI coding assistants like Cursor	When visual structure, context management, or multistep workflows are important
Speech/voice	Less common	Siri, smart home assistants (Alexa, Google Home), call center automation	When hands-free interaction or natural conversation is required
Video	Rare	Virtual tutors, therapy avatars, interactive learning agents	When visual demonstration, rich expression, or immersive learning is needed

Another key UX consideration is how the context is managed over time. Some generative AI applications have no memory or learning, so have precisely the information you present them with in exactly that session. This requires users to copy and paste information into the prompt. More modern applications automatically manage this context for you. For example, Cursor uses the integrated development environment to intelligently identify code to include in each model inference. Some applications retain memory over time, enabling agents to 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 and clear explanations of actions contribute to building relationships where users feel confident relying on agents in high-stakes scenarios.

This chapter explores these core aspects of UX 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.

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

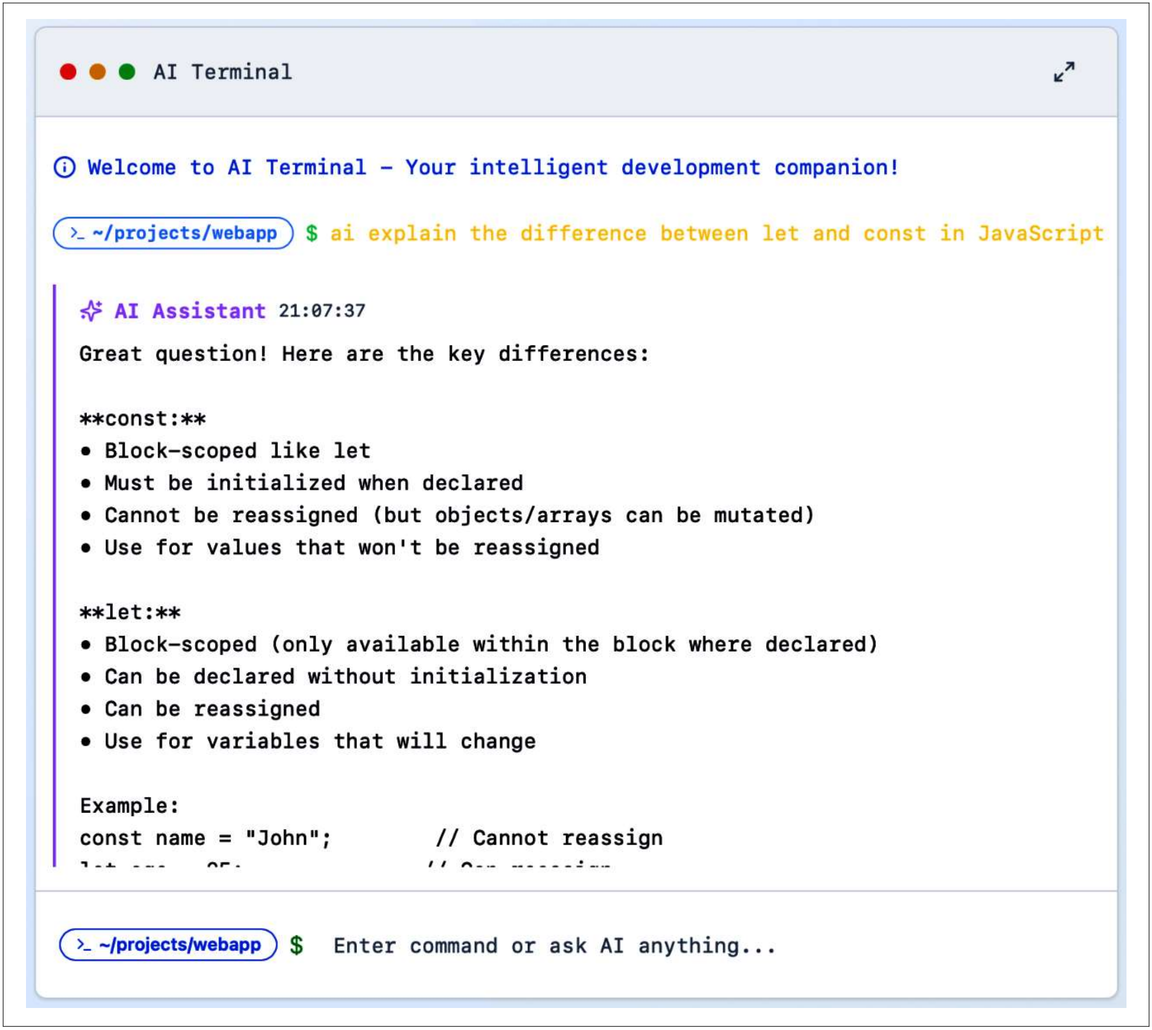
Text-Based

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 synchronous conversations (in real time) 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.

In recent years, the text-based modality has undergone a renaissance driven by the integration of advanced AI capabilities within terminal environments. Tools like Warp, Claude Code, and Gemini CLI illustrate this shift vividly. Warp reimagines the traditional developer terminal by integrating natural language command translation, intelligent autocompletion, and context-aware explanations, turning the command line into a collaborative, AI-augmented workspace. To illustrate this trend, Figure 3-1 shows an example of an AI-enabled terminal interface inspired by modern tools like Claude Code and Gemini CLI. This demonstration captures how developers can interact with the terminal using natural language prompts to generate, run, and debug commands seamlessly, without memorizing complex syntax or flags.

Similarly, Claude Code and Gemini CLI extend natural language interactions to code generation, execution, and file manipulation directly within terminal workflows, enabling developers to perform complex tasks by simply describing their goals in plain English. This figure highlights how AI is revitalizing the humble terminal, transforming it from a tool accessible only to those with deep command-line expertise into an approachable, powerful gateway for both novice and expert users to interact with systems through natural language.

This trend reflects a broader rethinking of what text-based interfaces can achieve. The incredible natural language understanding capabilities of modern foundation models are making ordinary text-based interactions more powerful than ever before. Where traditional terminals required precise syntax knowledge and memorization of command flags, AI terminals now act as conversational partners, interpreting user intent, suggesting best practices, and even debugging errors in real time. This shift is democratizing access to powerful systems operations, scripting, and data workflows, making the terminal "new again" as an accessible, intelligent gateway for both novice and expert users.



```
AI Terminal

ⓘ Welcome to AI Terminal - Your intelligent development companion!

~/projects/webapp $ ai explain the difference between let and const in JavaScript

AI Assistant 21:07:37
Great question! Here are the key differences:

**const:**
• Block-scoped like let
• Must be initialized when declared
• Cannot be reassigned (but objects/arrays can be mutated)
• Use for values that won't be reassigned

**let:**
• Block-scoped (only available within the block where declared)
• Can be declared without initialization
• Can be reassigned
• Use for variables that will change

Example:
const name = "John";        // Cannot reassign

~/projects/webapp $  Enter command or ask AI anything...
```

Figure 3-1. AI-enabled terminal interface. A demonstration of an AI-augmented terminal, where natural language inputs are interpreted into executable commands. Such interfaces transform the traditional command line into an intelligent conversational partner for system operations and development workflows.

However, a key limitation of text-based interfaces is discoverability. Users often do not know what capabilities the agent supports or how to phrase commands effectively. Unlike graphical interfaces—where options, buttons, and menus visually indicate what actions are possible—text-based interfaces require users to guess or recall available functionalities. This lack of affordances can lead to confusion, underutilization of agent capabilities, and user frustration when their requests fall outside the agent's supported scope. For example, a user might ask a support chatbot to modify an order detail that the system does not support, receiving an opaque rejection rather than guidance toward what is possible.

Designing effective text-based agents therefore requires strategies to enhance discoverability. Agents should proactively communicate their supported functions, either through onboarding messages, periodic capability reminders, or dynamic suggestions during conversation. For instance, an agent can respond to a greeting not only with "How can I help you today?" but also with "I can help you cancel orders, check delivery status, or update your account details." This approach ensures users understand the agent's operational boundaries, reducing trial-and-error interactions.

Beyond discoverability, text-based design 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 multiturn 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.

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. For example, text-based agents might be ideal for chat interfaces over messaging apps—like Slack, Teams, and WhatsApp for scalable communications with customers or employees—or text-heavy workloads like customer service, claims processing, or textual research tasks. Their accessibility and ease of deployment make them a cornerstone of agentic UX design—provided their limitations (particularly around discoverability) are mitigated through clear communication of capabilities, 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 multistep 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. Humans primarily rely on visual input and can process visual information more quickly and easily than text-based information. 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, 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. Tools like LangSmith, n8n, Arize, and AutoGen are beginning to illustrate agent workflows visually, making them easier to understand, debug, and reason about; we are likely to see much more of this visual orchestration in the future. To see how these graphical orchestration interfaces are emerging in practice, Figure 3-2 shows an example of a modern agent workflow builder. Tools like this illustrate agent actions, tool calls, conditionals, and outputs as connected visual nodes, enabling developers and operators to easily understand, debug, and optimize complex agentic flows without stepping through raw code alone.

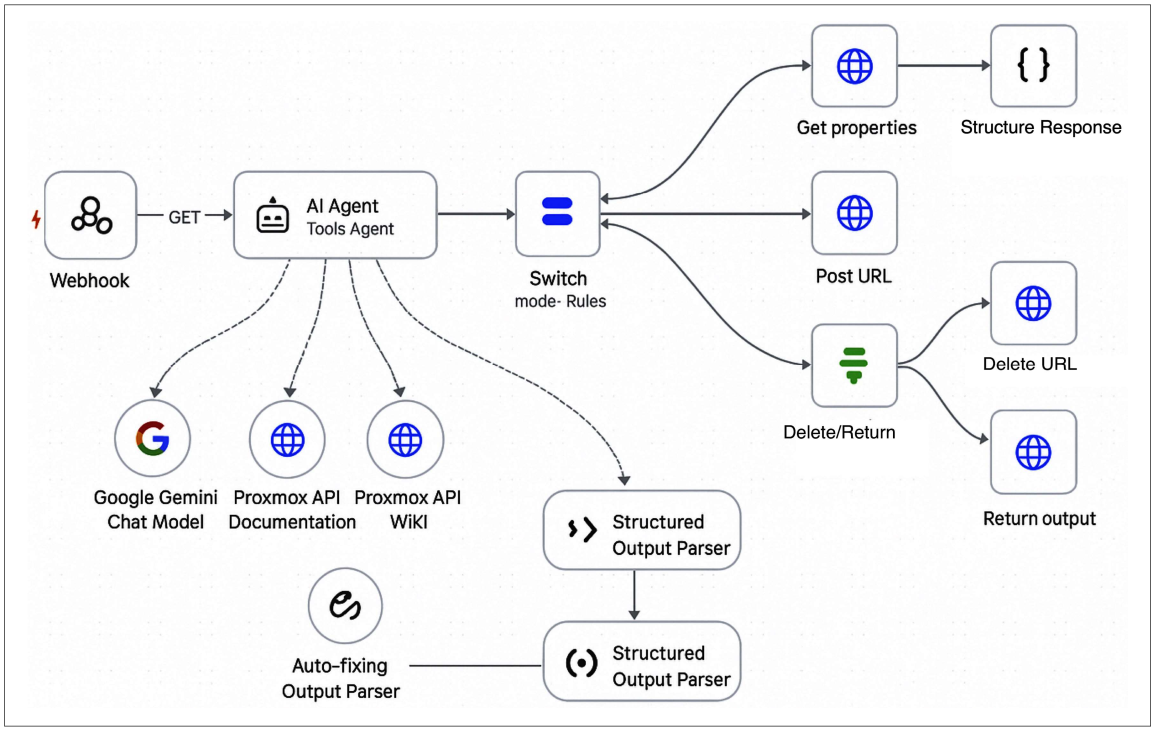
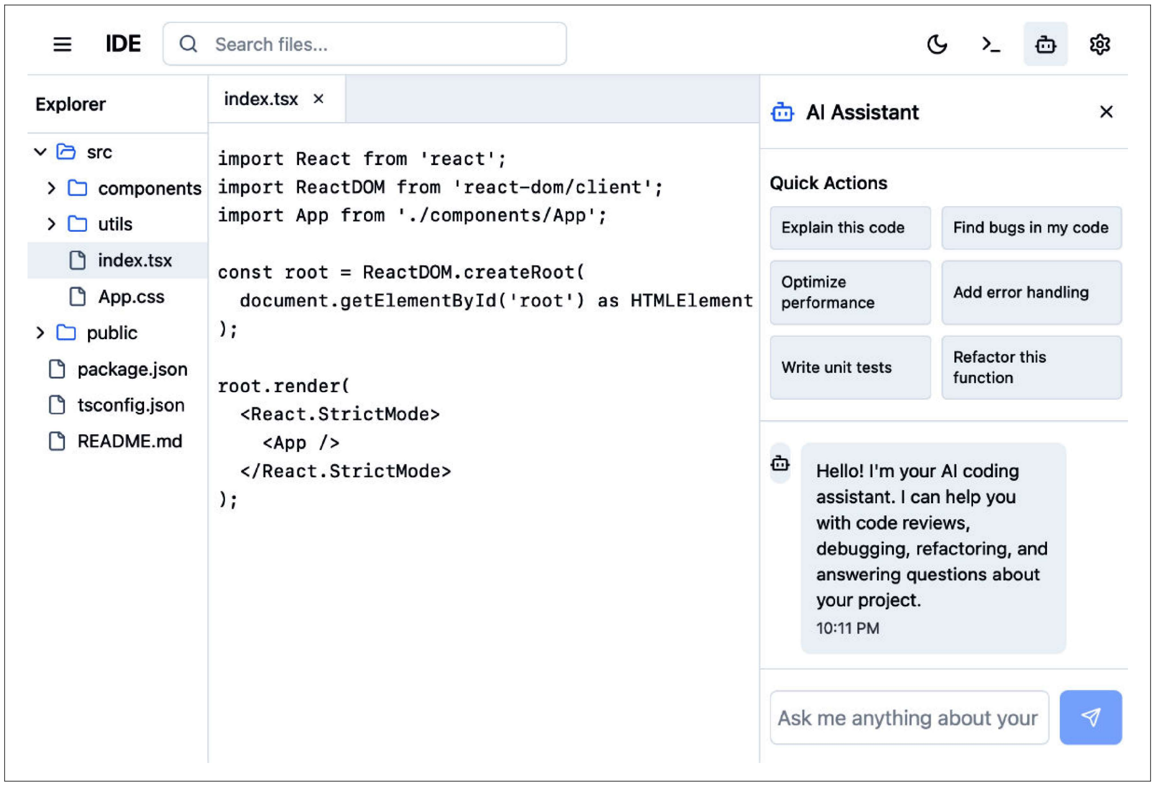


Figure 3-2. Visual orchestration of an agent workflow in n8n.io. This interface displays an AI agent integrated with multiple tools, models, and structured parsing components arranged in a node-based workflow. Such visual designs make it easier to build, manage, and iterate on multistep agent pipelines at scale.

Similarly, Figure 3-3 shows a modern AI-enabled IDE interface, similar to tools like Cursor, Windsurf, Cline, and many more. These environments integrate natural language understanding directly into the coding workflow, enabling developers to ask questions, generate code, refactor functions, and receive explanations or performance optimizations—all within a single, streamlined graphical interface.



```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './components/App';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);

root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Hello! I'm your AI coding assistant. I can help you with code reviews, debugging, refactoring, and answering questions about your project.

10:11 PM

Figure 3-3. AI-enabled IDE interface. An integrated development environment (IDE) enhanced with AI capabilities, combining traditional file explorers and code editors with natural language assistant panels that provide explanations, debugging suggestions, and autogenerated code improvements.

Together, these examples illustrate the rapid evolution of graphical agentic UX. As these interfaces mature, they will redefine what productive, AI-enabled tools look like —not just for developers, but for every knowledge-intensive profession.

A growing frontier in graphical agent interfaces is the emergence of generative UIs. Instead of relying solely on static dashboards or predesigned layouts, generative UIs dynamically create interface elements, data visualizations, or structured outputs based on user queries. For example, Perplexity AI not only provides textual answers but also generates structured knowledge cards, reference lists, and data tables tailored to the question asked. Similarly, AI coding copilots generate entire forms, config files, or UI components based on user intent.

Generative UIs combine the flexibility of natural language with the clarity and discoverability of graphical layouts, enabling agents to create rich, context-specific interfaces on demand. This expands the usefulness of graphical agents from predefined workflows to open-ended tasks where visual structuring enhances understanding. However, designing generative UIs introduces new challenges: ensuring the generated elements are usable and aesthetically coherent, and that they do not overwhelm users with poorly organized or excessive information. Careful design patterns, layout constraints, and prioritization logic are critical to keep generative UIs effective and user-friendly.

Designing effective graphical agent interfaces also comes with traditional challenges. Screen real estate is limited, requiring prioritization of displayed information to ensure critical details are not buried in clutter. Agents must 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, ecommerce product recommendation systems with filters and visual previews, and generative UI systems that dynamically produce structured outputs tailored to user questions. They are particularly effective in hybrid workflows where agents operate in the background but present updates or options visually for user confirmation.

When implemented thoughtfully, graphical and generative 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, intuitive design patterns, and the emerging potential of generative UI capabilities, graphical interfaces ensure that agent interactions feel smooth, transparent, and aligned with user expectations.

Graphical interfaces excel in use cases where data visualization, structured interactions, and clear status updates are essential. Recent years have seen enormous growth in tools like Lovable, Cursor, Windsurf, and GitHub Copilot, which offer high-quality GUIs that manage context and complex multistep operations with remarkable fluidity. These tools are redefining what productive, agent-enabled interfaces look like for developers. It is time to think just as hard about what the next generation of

AI-enabled, agentic UX will be for other professions—lawyers, accountants, insurance professionals, product managers, and knowledge workers. The future of work may not revolve around documents, spreadsheets, and slide decks, but around interactive, agent-driven interfaces purpose-built for decision making, analysis, and creation.

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's Alexa and Apple's 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 conversational flow and made voice interfaces feel clunky or robotic. However, the past two years have seen astonishing advances in this space. New low-latency speech recognition models, combined with more efficient language processing architectures, have dramatically reduced delays. Equally important, the *fluidity* and *capability* of voice AI systems have improved, enabling more natural-sounding interactions that can handle interruptions, mid-sentence corrections, and shifts in conversation topic.

Graceful handling of interruptions is a particularly important aspect of voice interface design. Human conversations are rarely linear monologues; people interrupt themselves to clarify, change direction, or refine a request mid-sentence. Effective voice agents must mirror this conversational flexibility, allowing users to interrupt commands without confusion, revise their inputs seamlessly, and resume where they left off without forcing a complete restart. For example, a user might say, "Book me a table for—oh wait, make that tomorrow instead," and a well-designed agent will adapt fluidly to incorporate the correction without requiring the user to start the command again. This capability not only makes interactions feel more natural but also builds trust and reduces frustration, as users feel the agent is responsive to their real communication patterns rather than demanding rigid, computer-like inputs.

Another major leap has been the integration of tool use into voice agent workflows. Modern voice agents are no longer limited to parsing commands and returning static answers. Instead, they can now pull in external context, update records, and take real-time actions—such as scheduling appointments, changing system configurations, or placing orders—based on dynamic conversational inputs. This ability to combine natural voice interaction with structured backend operations is transforming what voice agents can achieve.

Despite these impressive technological advances, it is important to note that voice interfaces remain a frontier technology. It is true that they have entered mainstream use in smart speakers and simple assistants. However, fully conversational, multiturn, context-aware voice agents with action-taking capabilities are not yet widely deployed across industries. Many enterprises are only beginning to explore voice interfaces for customer service, healthcare, logistics, and field operations.

A key consideration in deploying voice interfaces is understanding the speed at which humans process spoken versus written information. Humans typically speak at 150–180 words per minute, whereas reading speeds average 250–300 words per minute, with skimming speeds exceeding 500 words per minute. This means spoken interfaces are inherently slower for dense or complex information, where text-based interfaces enable faster comprehension and easier reference. However, voice excels in scenarios where hands-free convenience, natural interaction, and immediate contextual responsiveness outweigh these speed constraints.

The following example demonstrates a minimal FastAPI server using the OpenAI Realtime Voice API. It streams microphone audio from a browser to the agent and plays back the assistant's audio responses in real time. Notably, it handles interruptions gracefully: if the user starts speaking mid-response, it immediately truncates the assistant's output to keep the conversation natural. This compact implementation shows the core architecture for building low-latency, interruption-aware voice interfaces with agents:

```python
import os, json, base64, asyncio, websockets
from fastapi import FastAPI, WebSocket
from dotenv import load_dotenv

load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
VOICE          = "alloy"                  # GPT-4o voice
PCM_SR         = 16000                    # sample-rate we'll use client-side
PORT           = 5050

app = FastAPI()

@app.websocket("/voice")
async def voice_bridge(ws: WebSocket) -> None:
    """
    1. Browser opens ws://host:5050/voice
    2. Browser streams base64-encoded 16-bit mono PCM chunks: {"audio": "<b64>"}
    3. We forward chunks to OpenAI Realtime (`input_audio_buffer.append`)
    4. We relay assistant audio deltas back to the browser the same way
    5. We listen for 'speech_started' events and send a truncate if
       user interrupts
    """
```

```
await ws.accept()

openai_ws = await websockets.connect(
    "wss://api.openai.com/v1/realtime?" +
        "model=gpt-4o-realtime-preview-2024-10-01" (split across two lines)
    extra_headers={
        "Authorization": f"Bearer {OPENAI_API_KEY}",
        "OpenAI-Beta" : "realtime=v1"
    },
    max_size=None, max_queue=None # unbounded for demo simplicity
)

# initialize the realtime session
await openai_ws.send(json.dumps({
    "type": "session.update",
    "session": {
        "turn_detection": {"type": "server_vad"},
        "input_audio_format": f"pcm_{PCM_SR}",
        "output_audio_format": f"pcm_{PCM_SR}",
        "voice": VOICE,
        "modalities": ["audio"],
        "instructions": "You are a concise AI assistant."
    }
}))

last_assistant_item = None          # track current assistant response
latest_pcm_ts       = 0             # ms timestamp from client
pending_marks       = []

async def from_client() -> None:
    """Relay microphone PCM chunks from browser → OpenAI."""
    nonlocal latest_pcm_ts
    async for msg in ws.iter_text():
        data = json.loads(msg)
        pcm = base64.b64decode(data["audio"])
        latest_pcm_ts += int(len(pcm) / (PCM_SR * 2) * 1000)
        await openai_ws.send(json.dumps({
            "type": "input_audio_buffer.append",
            "audio": base64.b64encode(pcm).decode("ascii")
        }))

async def to_client() -> None:
    """Relay assistant audio + handle interruptions."""
    nonlocal last_assistant_item, pending_marks
    async for raw in openai_ws:
        msg = json.loads(raw)

        # assistant speaks
        if msg["type"] == "response.audio.delta":
            pcm = base64.b64decode(msg["delta"])
            await ws.send_json({"audio":
                base64.b64encode(pcm).decode("ascii")})
                last_assistant_item = msg.get("item_id")

            # user started talking → cancel assistant speech
            started = "input_audio_buffer.speech_started"
            if msg["type"] == started and last_assistant_item:
                await openai_ws.send(json.dumps({
                    "type": "conversation.item.truncate",
                    "item_id": last_assistant_item,
                    "content_index": 0,
                    "audio_end_ms": 0   # stop immediately
                }))
                last_assistant_item = None
                pending_marks.clear()

    try:
        await asyncio.gather(from_client(), to_client())
    finally:
        await openai_ws.close()
        await ws.close()

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("realtime_voice_minimal:app", host="0.0.0.0", port=PORT)
```

Looking ahead, we are likely to see significant adoption of advanced voice interfaces in the coming years, driven by falling costs, reduced latency, improved speech recognition, and better orchestration with backend tools. In healthcare, voice agents can assist doctors with hands-free note-taking during patient consultations. In customer service, they are replacing rigid interactive voice response (IVR) systems with fluid, humanlike conversations that resolve issues end to end. In industrial applications, workers can control machinery, log observations, or access manuals without stopping their tasks.

Ultimately, voice interfaces are most effective for short, hands-free tasks, quick queries, and action-oriented workflows, rather than for dense information consumption or complex decision making that requires rapid skimming or side-by-side comparison.

When thoughtfully designed, speech and voice interfaces offer unparalleled convenience, accessibility, and flexibility in agent interactions. As these technologies continue to mature and integrate deeply with backend tools and knowledge systems, they are poised to become indispensable in 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. While many of these experiences are still in the uncanny valley, the rapid pace of improvement suggests that this technology is getting closer to prime time, and more teams will begin building experiences around it.

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 humanlike 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, humanlike 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.

Combining Modalities for Seamless Experiences

While each interaction modality—text, graphical interfaces, voice, and video—has its own strengths and limitations, the most compelling agentic experiences often combine multiple modalities into a single, cohesive user journey. Users don't think in terms of modality boundaries; they simply want to achieve their goals as effortlessly and naturally as possible. The ability to move seamlessly across modalities—maintaining state and context throughout—is a hallmark of great agent system design.

For example, a user might begin interacting with an agent via voice while driving, continue the conversation on their phone through text while walking into a meeting, and later review a graphical dashboard summarizing results on their laptop. In another scenario, a voice assistant might read out a summary of an analytics report before emailing a detailed, text-based version with accompanying charts for later reference. This fluid transition between modalities preserves user context, respects situational constraints, and delivers the right interaction style at each moment.

Designing for modality fluidity requires careful state management and context persistence so that information, task progress, and user preferences are never lost in transition. Agents must also adapt their communication style to suit each modality—for example, delivering concise spoken summaries while providing more detailed textual outputs for review.

This is an exciting time for the field of human-computer interaction. Recent advances in foundation models, multimodal architectures, and agent orchestration are unlocking entirely new ways of interacting with intelligent systems. For the first time, it is technically feasible to build agents that engage users across text, voice, images, and video in a single, unified workflow.

However, while the technology frontier is expanding rapidly, it is critical to remember that core UX and product principles remain unchanged. Building successful agent experiences isn't about showcasing the latest modality integrations or generative UI capabilities for their own sake. It is about understanding users deeply, meeting them where they are, and creating intuitive, trustworthy, and delightful experiences that solve real problems in their lives.

The best products are not those that merely demonstrate technological sophistication, but those that use technology to amplify human capability in elegant and unobtrusive ways. As we continue to push the boundaries of modality design, let us stay grounded in the timeless goal of great product design: creating tools that people love to use, that make their lives easier, and that empower them to achieve what matters most.

The Autonomy Slider

A critical yet often overlooked dimension in UX design is the level of autonomy granted to agents. As Andrej Karpathy described, effective agentic systems should allow users to smoothly adjust an agent's autonomy—from fully manual control to partial automation to fully autonomous operation. This concept, often called an autonomy slider, empowers users to choose how much control they wish to retain versus delegate at any given time. Figure 3-4 illustrates a simple example of an autonomy slider interface, enabling users to set the agent to "Manual," "Ask," or "Agent" mode depending on their task, trust, and context.

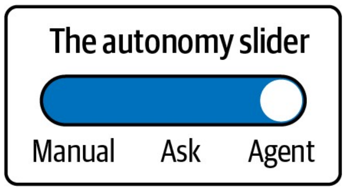


Figure 3-4. The autonomy slider enables users to adjust an agent's level of independence, ranging from fully manual control, to assisted "Ask" mode, to fully autonomous agent execution. This flexibility builds user trust by aligning system behavior with user preferences, task complexity, and context.

Different users, tasks, and contexts demand different degrees of agent autonomy. In some situations, users prefer full manual control to ensure precision, while in others, they may want to offload routine or complex tasks entirely to the agent. Critically, these preferences are not static; they evolve with user trust, task familiarity, stakes, and workload. For example:

Manual
: The developer writes all code themselves without agent assistance. The IDE acts purely as an editor with syntax highlighting and linting but no AI-driven suggestions.

Ask (assisted)
: The agent proactively suggests code completions, refactors, or documentation snippets, but the developer reviews and accepts each suggestion before it is applied. This mode speeds up development while keeping the human fully in control.

Agent
: The agent autonomously performs certain tasks, such as applying standard refactors, fixing linter errors, or generating boilerplate code files based on project conventions without requiring individual approvals. The developer is notified of changes but does not need to approve each action.

These three modes demonstrate how an autonomy slider empowers developers to balance control and efficiency within a single interface. The same principle applies beyond software development. For example, in a customer support platform:

Manual
: Human agents handle all incoming customer queries themselves. The AI is inactive or used only for backend analytics, not frontline interactions.

Ask (assisted)
: The agent drafts suggested replies to customer messages, surfacing recommended responses, policy references, or troubleshooting steps. The human agent

reviews, edits if necessary, and approves the reply before sending. This accelerates response time while maintaining human judgment.

Agent

The agent autonomously handles routine queries—such as password resets, order tracking, or FAQs—without human intervention, escalating only complex or sensitive issues to human agents. Users are notified of agent actions but do not need to approve each message for standard interactions.

These three modes coexist within the same customer support system, empowering teams to adjust autonomy based on query complexity, customer profile, and organizational trust in AI. This same autonomy slider pattern can extend to any field where workflows benefit from fluidly shifting between manual execution, AI assistance, and full agentic automation. This spectrum of autonomy must be consciously designed into agent experiences. Without it, agents risk feeling either underpowered (if they require too much manual input) or overbearing (if they act without user consent in sensitive contexts). To integrate an autonomy slider effectively, consider the following design principles:

Expose degrees of autonomy clearly

Users should understand the available levels of agent independence, from manual to assisted to autonomous. Label these modes in intuitive language, such as "Manual," "Assist," and "Auto," and explain their implications.

Enable seamless transitions

Users must be able to shift between autonomy levels effortlessly as their confidence, context, or workload changes. For instance, a toggle or slider in the interface should offer a quick transition from review mode to auto-approve mode.

Provide predictable and transparent behavior at each level

Each autonomy level should have well-defined behaviors. In partial automation, for example, the agent may draft an output but require explicit user approval before execution. In full autonomy, it should still provide status updates and options to intervene.

Communicate the risks and benefits of each level

Users should be aware of what they gain or risk by increasing agent autonomy. For critical tasks, it may be advisable to require an explicit user confirmation before enabling full autonomy.

Adapt autonomy based on user trust and competence

Intelligent systems can gradually suggest higher autonomy levels as users gain trust and as the agent demonstrates reliability. For example, after 10 successful uses in manual mode, the system might suggest trying assist mode to save time.

Importantly, the autonomy slider is not merely a feature—it is a trust-building mechanism. By giving users control over how much autonomy an agent exercises, systems communicate respect for user expertise and agency. It avoids the common pitfall of "one-size-fits-all" autonomy that either overwhelms or underutilizes user potential. Always ask: how easily can my users move between manual, assisted, and fully autonomous modes? The answer to this question will shape whether your agent is adopted as a reliable partner or sidelined as an untrusted tool.

Synchronous Versus 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 enable 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 and context 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 and maintains context without unnecessary friction.

Design Principles for Asynchronous 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 both asynchronous and synchronous 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. We'll cover this in more detail in Chapter 6 on memory.

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

Ensuring context retention and continuity across user interactions is an important aspect of designing effective agent systems. Whether an agent is guiding a user through a multistep 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.

While context retention is a technical capability, it is fundamentally a UX consideration because it determines whether users experience the agent as a cohesive, attentive collaborator or as a disconnected tool that forces them to repeat themselves. From the user's perspective, memory creates a sense of continuity, personalization, and intelligence. If an agent remembers previous interactions, user preferences, or in-progress tasks, it can seamlessly continue conversations and workflows, reducing cognitive load and frustration.

Implementation approaches directly shape UX. A purely client-side context (e.g., stored in browser memory) may feel fast within a session but loses continuity across devices or logins, undermining seamless UX. A purely server-side context (e.g., stored in a database tied to user ID) enables long-term memory and cross-device experiences but can introduce latency or privacy considerations. A hybrid approach —maintaining short-term context on the client side for responsiveness and persisting long-term context on the server side for continuity—often achieves the best UX balance. Choosing the right strategy depends on the user journey, privacy requirements, and level of personalization intended. Ultimately, context is UX: it is how an agent remembers, adapts, and responds in ways that make it feel human-centered and supportive rather than stateless or mechanical.

Effective context retention requires agents to manage both short-term and long-term memory effectively. Short-term memory enables 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, enabling 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 the next 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 multiturn conversations, task handoffs, and workflows with intermediate states, where losing context can result in frustration, inefficiency, and abandonment of tasks.

Effective state management depends on how the system identifies and tracks users or sessions. For logged-in users, state can be tied directly to their user accounts, enabling memory persistence across devices and sessions. For anonymous interactions, maintaining context typically requires a session identifier—such as a cookie or token—to track the conversation between the client and server.

As agent systems scale to thousands or millions of users, session state should not reside only in memory. Persisting state in a database or distributed cache ensures continuity across server restarts, enables load balancing, and supports multidevice experiences. The choice between user-based memory (persistent, personalized) and session-based memory (ephemeral, session-scoped) depends on your application's privacy requirements, user expectations, and operational architecture. Regardless of implementation, robust identification and storage strategies are fundamental to delivering seamless, context-aware agent experiences at scale.

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 enables agents to resume tasks across multiple sessions so that users can 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 enables 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 multistep 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 critical aspects of designing effective agent experiences is ensuring users understand what the agent can do and how to interact with it effectively. While backend agent design determines what functions an agent supports, the user experience determines whether those capabilities are discoverable, intuitive, and usable in practice. In traditional applications, discoverability is straightforward: menus, buttons, and interface elements visually communicate available actions. In agentic systems, especially those using text or voice interfaces, the absence of visible affordances often leaves users guessing what the agent can and cannot do.

Effective agent UX addresses this challenge by proactively communicating capabilities through the interface itself. For example, many chat-based agents include suggested action buttons below the input field, highlighting common or contextually relevant actions such as "Track order," "Generate summary," or "Create meeting note." These buttons serve as visual affordances, guiding users toward supported workflows without requiring them to remember specific commands or guess what is possible. Similarly, onboarding tutorials or first-use walkthroughs can introduce users to an agent's core functions, helping them build confidence early on.

Another useful pattern is the inclusion of expandable menus or capability cards that list available functions in a structured way. In a graphical agent interface, for instance, a sidebar might contain sections for data retrieval, analysis, summarization, and workflow automation. This mirrors the menu structures that users expect in traditional apps while communicating the breadth of agent capabilities upfront. Dynamic suggestions, where the system recommends actions based on user input, also help bridge the gap between open-ended natural language and structured tool invocation. If a user begins typing "book…," the agent might suggest "Book meeting with [name]," "Book conference room," or "Book travel," anticipating intent and making actions easier to execute.

In systems relying primarily on open-ended text input, agents themselves must communicate their capabilities clearly in conversation. This can include proactive introductions when a session begins, such as: "Hi, I can help you generate content, analyze data, or summarize documents. What would you like to do today?" When users request actions beyond current capabilities, the agent should not simply reject the request but provide alternatives: "I can't process payments directly, but I can update your billing preferences or connect you with an agent who can assist." Such responses reduce user frustration while reinforcing the agent's utility.

While it is important to surface capabilities, it is equally critical not to overwhelm users with too many options at once. Effective designs prioritize progressive disclosure, showing core capabilities initially and revealing advanced features as users become more comfortable. Contextual relevance also plays a key role. Displaying the most likely actions based on current user inputs, historical behavior, or workflow stage ensures the agent feels supportive rather than cluttered. Visual grouping and clear hierarchy within menus or suggested actions help users navigate available options efficiently.

These principles apply across modalities. In text-based chat interfaces, quick-reply buttons and example prompts improve clarity. In graphical dashboards, capability menus and tooltips communicate functions without crowding the interface. Voice agents must balance brevity with clarity, listing only a few high-priority options at a time to avoid cognitive overload. Generative UI systems can combine natural language and dynamically generated visual outputs to make available capabilities immediately visible and actionable.

Ultimately, communicating agent capabilities is not merely about stating what the agent can do; it is about designing an experience that empowers users to harness those capabilities confidently and efficiently. When users understand an agent's scope and limitations, they are far more likely to engage productively, trust its outputs, and integrate it into their workflows. Thoughtful UX design turns invisible functions into visible affordances, transforming agents from opaque black boxes into transparent, collaborative digital partners.

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.

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 nonnegotiable, 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 with "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 agentic 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 multistep 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 in Interaction Design

Trust is gained in drops and lost in buckets. This certainly applies to agentic systems as well. 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 is 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 remainder of this section, we'll explore two key components of trust-building: ensuring predictability and reliability in agent behavior.

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 up front.

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.

Conclusion

Designing exceptional user experiences for agent systems goes far beyond technical functionality—it requires an 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, tradeoffs, 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.

Exceptional agents seamlessly retain context and adapt to users, 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. Some common patterns to keep in mind:

Communicate capabilities clearly
: Show users what the agent can do through onboarding, suggestions, or buttons.

Combine modalities thoughtfully
: Align text, GUI, voice, or video with the task and user context.

Retain context thoughtfully
: Maintain relevant conversation state without overwhelming memory or violating privacy.

Handle errors gracefully
: Provide clear, polite fallbacks when the agent can't fulfill a request.

Build trust
: Be transparent about limitations, confidence, and reasoning.

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. Trust is earned not just through success but also through how agents handle ambiguity, failure, and recovery.

As the agent landscape continues to shift and expand, designers and developers must remain agile—continually reevaluating interaction paradigms, adapting to new multimodal capabilities, and experimenting with novel UX patterns. The design patterns described here provide a robust starting point, but the future of agentic UX will be shaped by rapid innovation in modalities, context management, and human-agent collaboration. 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 UX at every stage of development, we can ensure that agents become not just tools, but indispensable partners in our increasingly intelligent digital ecosystems. In Chapter 4, we'll cover tool use, which is how we move from ordinary chatbots to systems that can do real work for users.

CHAPTER 4

Tool Use

While foundation models are great at chatting for hours, tools are the building blocks that empower AI agents to retrieve additional information and context, perform tasks, and interact with the environment in meaningful ways. In the context of AI, a tool can be defined as a specific capability or a set of actions that an agent can perform to achieve a desired outcome. These tools range from simple, single-step tasks to complex, multistep 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, tools will be how those changes are executed.

The significance of tools in AI agents parallels the importance of competencies in human professionals. Just as a doctor needs a diverse set of tools to diagnose and treat patients, an AI agent requires a repertoire of tools to handle various tasks effectively. This chapter aims to provide a comprehensive understanding of tools 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 tools. These tools 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 tool 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 tools are critical to the overall functionality and effectiveness of the AI agent. We'll start with some fundamentals of LangChain, and then cover the different types of tools that can be provided to an autonomous agent, which we will cover in sequence: local tools, API-based tools, and MCP tools.

LangChain Fundamentals

Before diving deeper into tool selection and orchestration, it is helpful to understand some core LangChain concepts. At the heart of LangChain are foundation models and chat models, which process prompts and generate responses. For example, ChatOpenAI is a wrapper class that provides a simple interface to interact with OpenAI's chat-based models like GPT-5. You initialize it with parameters such as the model name to specify which model to use:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-4o")
```

LangChain structures interactions as messages to maintain conversational context. The two main message types are HumanMessage, which represents user inputs, and AIMessage, which represents the model's responses:

```
from langchain_core.messages import
HumanMessage messages = [HumanMessage("What is the weather today?")]
```

Tools, meanwhile, are external functions that your model can call to extend its capabilities beyond text generation—for instance, calling APIs, retrieving database entries, or performing calculations. You define a tool in LangChain using the @tool decorator, which registers the function and automatically generates the schema describing its inputs and outputs:

```
from langchain_core.tools import tool

@tool
def add_numbers(x: int, y: int) -> int:
    """Adds two numbers and returns the sum."""
    return x + y
```

Once you have defined your tools, you bind them to the model using .bind_tools(), which enables the model to select and invoke these tools in response to user inputs. To interact with the model, you use the .invoke() method, providing it with a list of messages representing the current conversation. If the model decides to call a tool, it will output a tool call, which you then execute by invoking the corresponding function and appending its result back into the conversation before generating the final response:

```
llm_with_tools = llm.bind_tools([add_numbers])
ai_msg = llm_with_tools.invoke(messages)
for tool_call in ai_msg.tool_calls:
    tool_response = add_numbers.invoke(tool_call)
```

These building blocks—chat models, messages, tools, and tool invocation—form the foundation of LangChain-based systems. Understanding how they fit together will help you follow the examples in this chapter and build your own agents that can seamlessly integrate language understanding with real-world actions.

Local Tools

These tools are designed to run locally. They are often based on predefined rules and logic, tailored to specific tasks. These local tools can be easily built and modified, and are co-deployed with the agent. They can especially augment weaknesses in language models that traditional programming techniques perform better at, such as arithmetic, time-zone conversions, calendar operations, or interactions with maps. These local tools offer precision, predictability, and simplicity. As the logic is explicitly defined, local tools tend to be predictable and reliable.

The metadata—the tool's name, description, and schema—is just as critical as its logic. The model uses that metadata to decide which tool to invoke. Therefore, the following is important:

- Choose precise, narrowly scoped names. If your name is too general, the LLM may call it when it's not needed.
- Write clear, distinctive descriptions. Overly broad or overlapping descriptions across multiple tools guarantee confusion and poor performance.
- Define strict input/output schemas. Explicit schemas help the foundation model understand exactly when and how to use the tool, reducing misfires.

Despite these benefits, local tools have some important drawbacks:

Scalability
: Designing, building, and deploying local tools can be cumbersome, time-consuming, and challenging, and local tools are harder to share across use cases. While tools can be exposed as libraries and shared across multiple agent use cases, this can be challenging in practice and at scale.

Duplication
: Every team or agent deployment that wants to use local tools will need to deploy the same library along with their agent service, and pushing changes to these tools will require coordinating deployments to each agent service that uses these tools. In practice, many teams simply reimplement the same tools independently to avoid the coordination overhead.

Maintenance
: As the environment or requirements change, handcrafted tools may need frequent updates and adjustments. This ongoing maintenance can be resource-intensive and typically requires a redeployment of your agent service.

Despite these drawbacks, manually crafted tools 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 on dates and times, and operations over maps and graphs, for example, are all areas where handcrafted tools can substantially improve the efficacy of agentic systems.

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

```python
from langchain_core.runnables import ConfigurableField
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 bind the tool with the foundation model in LangChain:

```python
tools = [multiply, exponentiate, add]

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

This "binding" operation registers the tool. Under the hood, LangChain will now check if the foundation model response includes any requests to call a tool. Now that we've bound the tool, we can ask the foundation model questions, and if the tool is helpful for answering the question, the foundation model will choose the tools, select the parameters for those tools, and invoke those functions:

```python
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,
       "exponentiate": exponentiate}[tool_call["name"].lower()]
   tool_msg = selected_tool.invoke(tool_call)
```

```
    print(f'{tool_msg.name} {tool_call['args']} {tool_msg.content}')
    messages.append(tool_msg)
    final_response = llm_with_tools.invoke(messages)
    print(final_response.content)
```

With those added print statements for visibility, we can see that the foundation model 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
```

The model will then include this result from the tool call in the generated final response, producing a result such as:

```
393 times 12.25 is 4814.25, and 11 + 49 is 60.
```

While the effect of this is simple, the implications are profound. The foundation model is now able to execute the computer programs that we bind with it. This is a simple example, but we can bind arbitrarily useful and consequential programs to the foundation model, and we now rely on the foundation model to choose which programs to execute with which parameters. Doing so responsibly, and only binding tools that the foundation model will execute in ways that produce more good than harm, is among the paramount responsibilities of developers building agents and agentic systems.

API-Based Tools

API-based tools 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 tools 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 tools 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 tools have multiple benefits.

By leveraging external services, these tools 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 tools. 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 tools, let's begin with enabling your agent to browse the open web for additional information. In this code snippet, we register a tool to retrieve information from Wikipedia, a step toward a full web browsing 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=1, doc_content_chars_max=300)
tool = WikipediaQueryRun(api_wrapper=api_wrapper)

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

messages = [HumanMessage("What was the most impressive thing" +
                         "about 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 foundation model 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:

```
{'query': 'Buzz Aldrin'}
Page: Buzz Aldrin
Summary: Buzz Aldrin (born Edwin Eugene Aldrin Jr. January 20, 1930) is an
American former astronaut, engineer and fighter pilot. He made three spacewalks
as pilot of the 1966 Gemini 12 mission, and was the Lunar Module Eagle pilot on
```

```
the 1969 Apollo 11 mission.

One of the most impressive things about Buzz Aldrin is that he was the Lunar
Module Eagle pilot on the 1969 Apollo 11 mission, making him one of the first
two humans to land on the Moon. This historic event marked a significant
achievement in space exploration and human history. Additionally, Aldrin made
three spacewalks as pilot of the 1966 Gemini 12 mission, showcasing his tools
and contributions to advancing space travel.
```

Let's now look at a second example, for an agent that is designed to fetch and display stock market data. This process involves defining the API interaction, handling the response, and integrating the tool into the agent's workflow. By following this approach, agents can integrate external data sources seamlessly, 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 tool for our agent, and we can then invoke it just like the previous tools:

```python
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 ticker for the company."""
   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 for {ticker}")


# 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])

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)
```

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

When designing API tools for agents, focus on reliability, security, and graceful failure. External services can go down, so agents need fallbacks or clear error messages. Secure all communications with HTTPS and strong authentication, especially for sensitive data.

Watch out for API rate limits to avoid disruptions, and ensure compliance with data privacy laws—anonymize or obfuscate user data when needed. Handle errors robustly so the agent can recover from network issues or invalid responses without breaking the user experience. When possible, consider alternatives and multiple providers for greater reliability if any given provider is degraded.

APIs empower agents with real-time data, heavy computation, and external actions they couldn't perform alone, making them far more capable and effective.

Plug-In Tools

These tools 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 tools enable rapid deployment and scaling of the agent's functionalities. These tools are predesigned 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 tools has become a standard offering from leading platforms such as OpenAI, Anthropic's Claude, Google's Gemini, and Microsoft's Phi as well as a growing open source community. Plug-in tools provide powerful tools to expand the capabilities of AI agents without extensive custom development.

OpenAI's plug-ins ecosystem offers powerful extensions—everything from real-time web search to specialized code generators—but they're only available inside the ChatGPT product, not the public API. You cannot invoke Expedia, Zapier, or any first-party ChatGPT plug-in through the standard OpenAI Completions or Chat endpoints. To replicate similar behavior in your own applications, you must build custom function-calling layers (for example, via LangChain) that approximate plug-in functionality.