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 th
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	W —
GPT-5 mini (2025-08-07)	0.819	0.835	0.756	0.927	0.8
o4-mini (2025-04-16)	0.812	0.82	0.735	0.929	0.8
o3 (2025-04-16)	0.811	0.859	0.753	0.869	0.8
GPT-5 (2025-08-07)	0.807	0.863	0.791	0.875	0.8
Qwen3 235B A22B Instruct 2507 FP8	0.798	0.844	0.726	0.835	0.8
Grok 4 (0709)	0.785	0.851	0.726	0.949	0.7
Claude 4 Opus (20250514, extended thinking)	0.78	0.875	0.709	0.849	0.8
gpt-oss-120b	0.77	0.795	0.684	0.836	0.8
Kimi K2 Instruct	0.768	0.819	0.652	0.85	0.8
Claude 4 Sonnet (20250514, extended thinking)	0.766	0.843	0.706	0.84	0.8

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.