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.

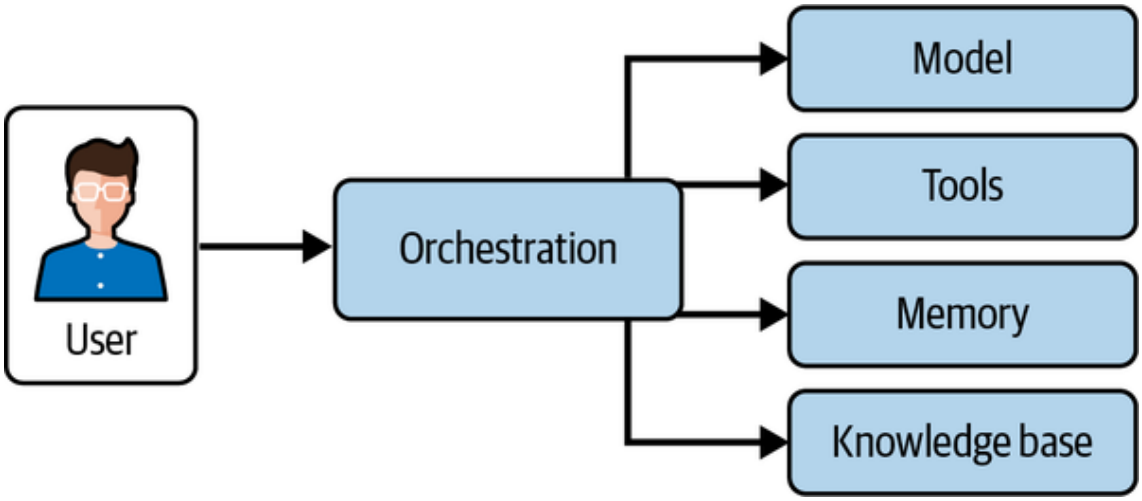

Figure 2-1. Core components of an agent system.

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.

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
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

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

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.

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
Grok 3	xAI	79.9	15	25
Nova Pro	Amazon	82.0	4	5.33
Mistral Large 2	Mistral	80.0	10	10

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

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.