A practical guide to the architectures of agentic applications

It's tempting to assume that the most important decision in building an AI product is choosing between GPT-4, Claude, or an open-source model, but this is a misconception. High-profile failures like IBM Watson for Oncology show that even the most advanced models can fall short when the architecture, integration, and overall system design aren't built for real-world complexity.

A powerful AI model with poor architecture will burn through the budget, waste GPU time, create technical debt, and perform worse than a less capable model built on a strong architectural foundation.

This article discusses the role of architecture in building effective agentic applications, describing the two core architectural models – single-agent and multi-agent – and how to think about workflow design, autonomy, and coordination. You'll also find common design patterns, real-world examples, and a decision framework to help you choose the right approach for your use case.

First, understand what your system needs to do

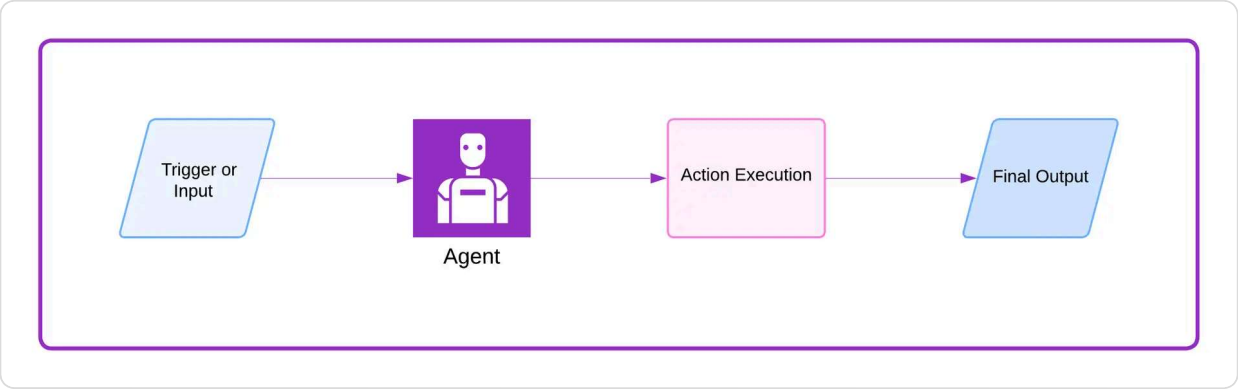
Before choosing an architecture, you should first define the end-to-end workflow your system needs to support. What must the system do from the first input to the final output, regardless of how it's implemented?

Is the workflow short and linear, or long with multiple steps? Can some steps run in parallel, or are there strict dependencies between stages? Does the flow involve multiple roles, services, or types of data? Does the process require coordination, retries, or adaptability?

Your system's workflow should shape its architecture and help you decide whether a single-agent setup is enough or a distributed, multi-agent design is needed to support its purpose.

Single-agent architectures

Single-agent architectures rely on one agent to handle the entire workflow from start to finish, making them well suited to simple, sequential workflows that require little coordination. The agent is responsible for reasoning (evaluating input and deciding what to do), planning (breaking goals into steps), executing (calling functions or APIs), and interacting with tools (using capabilities beyond the model, like databases or search).

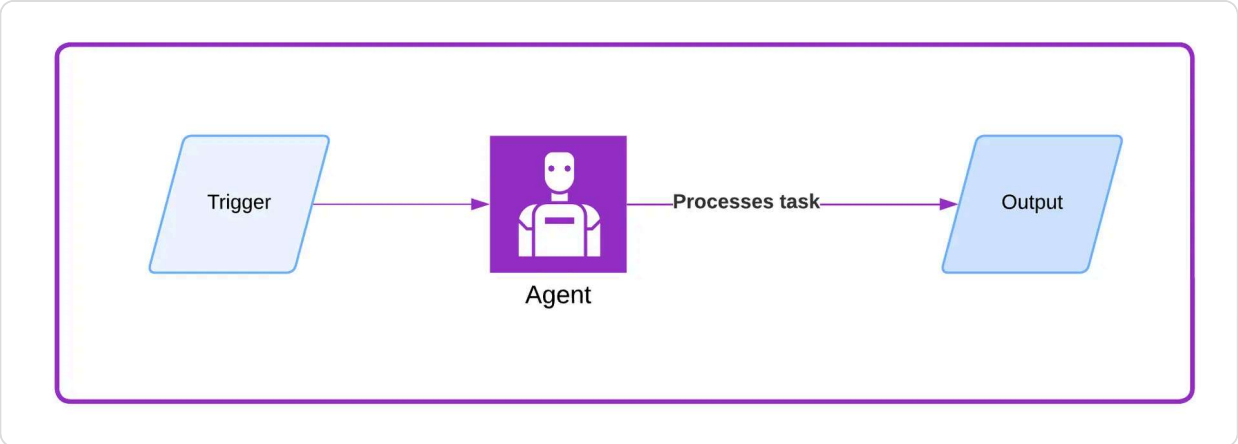


With thoughtful design, single-agent architectures can manage more complex workflows under the right conditions. For example, transparent, open-source SWE-agent (inspired by Devin) autonomously uses tools to fix issues in GitHub repositories, detect security vulnerabilities, and carry out other scripted workflows through a single control loop, without delegation or parallelism.

Let's take a look at some common patterns in single-agent architecture and how even basic workflows can benefit from making the right structural choices.

The single-agent pattern

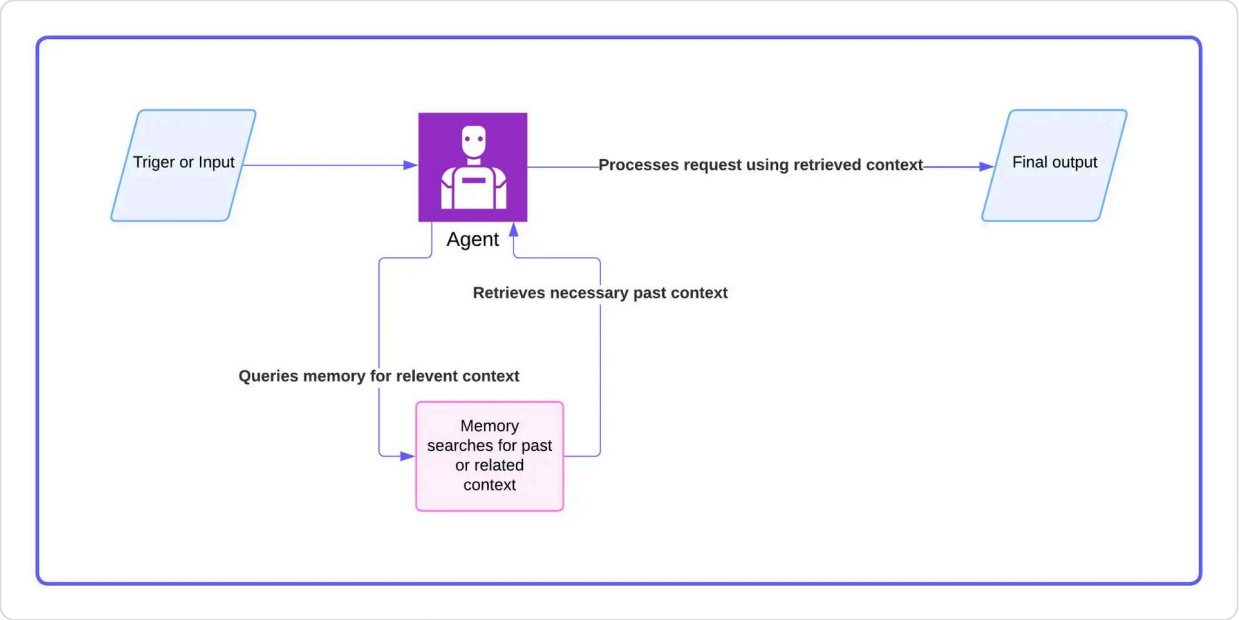
In its simplest form, a single-agent system reacts to a trigger, processes a task, and returns an output. No memory, planning, or external interaction is involved – only input, reasoning, and response. This pattern is useful for simple automation or prototypes and helps validate workflows and ideas when building agentic applications.



Open-source automation agent `bumpgen` is an example of a single-agent architecture that shows how even basic automation tasks benefit from thoughtful architecture. The agent watches your project for new package releases, fetches the latest versions, and creates automated pull requests to update them. The workflow is straightforward: detect, fetch, update. No coordination or parallelism is needed, so one agent can handle the full flow independently.

The memory-augmented agent pattern

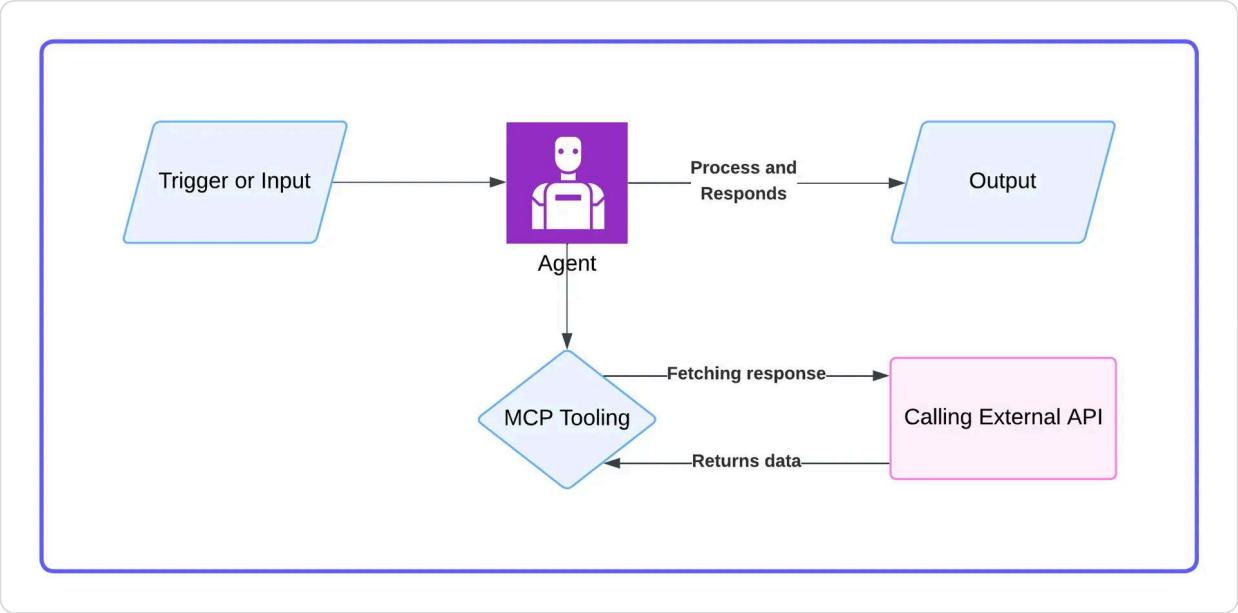
This pattern is useful when your system needs to remember past context (like previous user interactions, historical data, or external states) to make better decisions. For example, imagine you're building an automatic reminder system that sends personalized nudges to users. A cron-based trigger runs daily, the agent queries past messages or actions from a vector memory store, and uses that context to generate tailored reminders. This setup allows the agent to respond with awareness of past events.



The tool-using agent pattern

Let's say you're building a customer support agent that handles invoice requests. When a user opens a support ticket, the agent fetches billing data, formats it, and returns a summary – steps it can't complete alone. It calls the billing API, processes the response, and completes the task automatically.

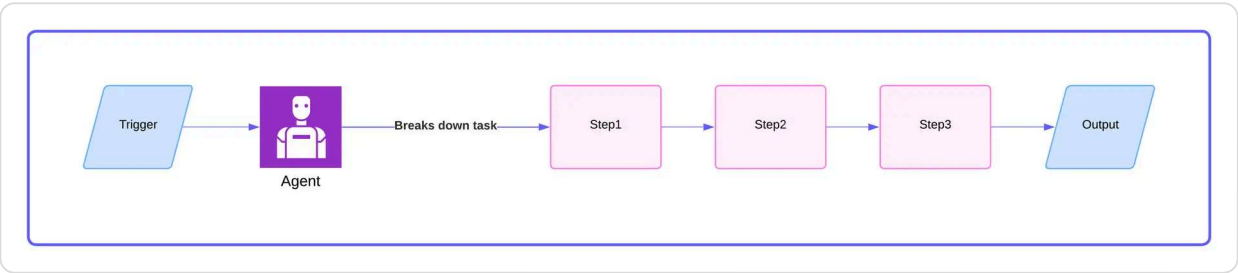
You don't want to hardcode every API interaction to make this process work reliably inside the agent. Instead, you can introduce, for example, an MCP (Model Context Protocol) layer – MCP is a protocol that standardizes access to external tools and services. The MCP layer represents the tooling interface where all external interactions, mostly APIs, are abstracted and maintained. This way, the logic stays inside the agent, and the heavy lifting is delegated to tools.



The planning-agent pattern

A planning agent generates a multi-step plan based on the initial input, walks through each action sequentially, and adapts as necessary by understanding dependencies between tasks and tracking execution.

One use case for the planning agent pattern is an AI onboarding assistant for a SaaS product: When a user signs up, the system must schedule a welcome email, set up a product tour, check in after three days, and escalate to human support if there's no engagement after a week. These steps don't just happen—they need to be planned and executed in the right order.

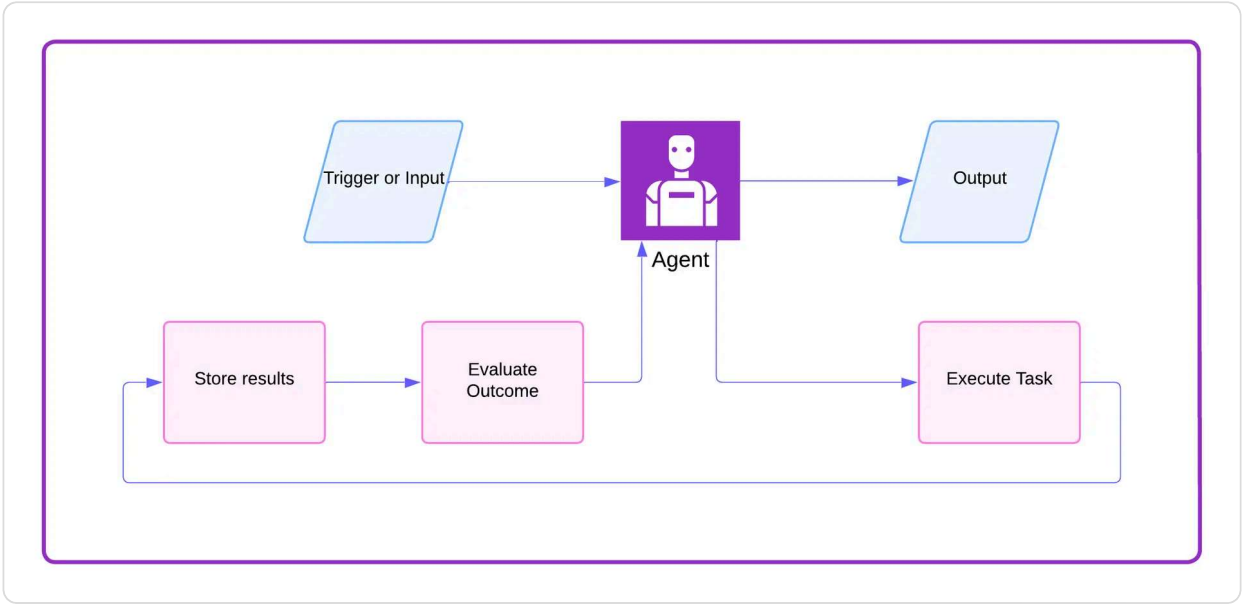


The planning-agent pattern is useful for tasks that can't be completed in one step and require a sequence of coordinated actions.

The reflection-agent pattern

A reflection agent is useful for tasks that require improvement over time, in addition to execution. After completing an action, the reflection agent stores the results, compares them to goals or metrics, and updates its strategy. Over time, this feedback loop helps an agent become more effective, even without human intervention.

Suppose you build a trading assistant that makes daily trades based on market signals. At the end of each day, the agent evaluates which trades performed well, which didn't, and how to adjust the strategy going forward.

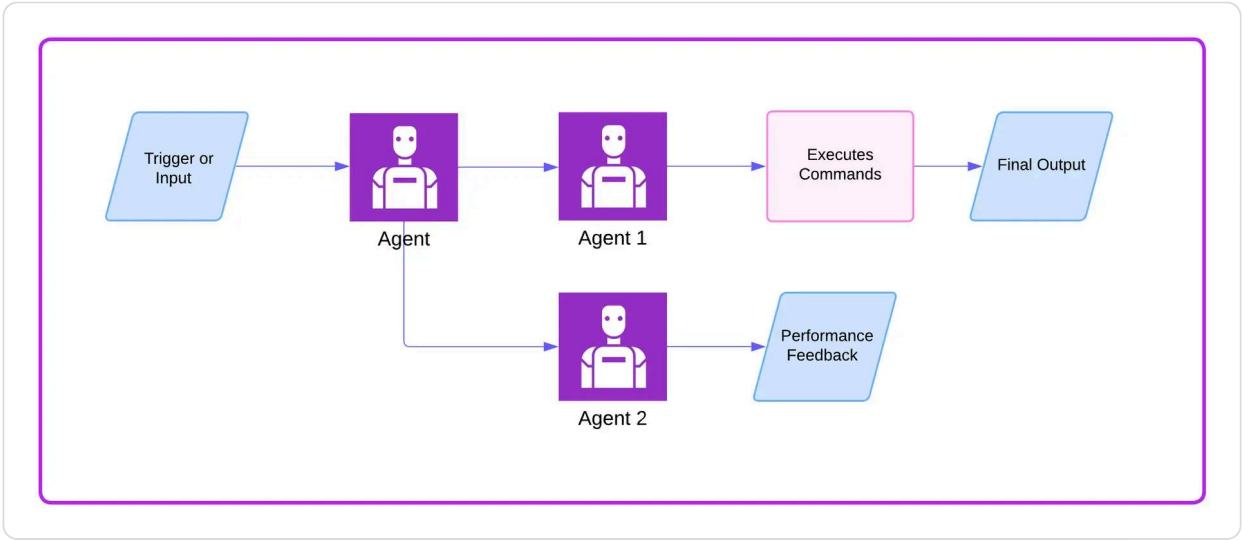


This pattern is useful when your system needs to learn from past outcomes to improve future performance.

Multi-agent architectures

Real-world workflows aren't simple. Once you've built an MVP or validated your flow with a single-agent setup, you'll likely need to scale, adding more reasoning, parallelism, precision, or specialization. That's when multi-agent architectures come in.

In a multi-agent architecture, multiple agents collaborate to complete a complex workflow. Each agent owns a specific responsibility – planning, retrieval, analysis, or execution – and communicates with others to move the process forward.



The open-source TaskWeaver project from Microsoft follows the multi-agent pattern: Goals are broken down into subtasks – such as retrieving documents, summarizing them, and drafting outputs – and delegated across agents. A central orchestrator manages collaboration and how results are passed between agents. This setup is common: Most effective multi-agent systems include a coordinating agent that supervises task delegation. Whether it's called a lead agent, supervisor, or manager, this agent ensures the workflow stays organized.

In a multi-agent setup, each agent is typically designed as a single-agent system with its own memory, tools, and decision logic. For example, in an AI trading system, you may have distinct agents responsible for:

- Fetching live market data.
- Performing technical and sentiment analysis.
- Reflecting on past performance to adjust strategy.
- Placing the trades via a broker API.

In multi-agent architectures, the focus shifts from what each agent does to how they collaborate.

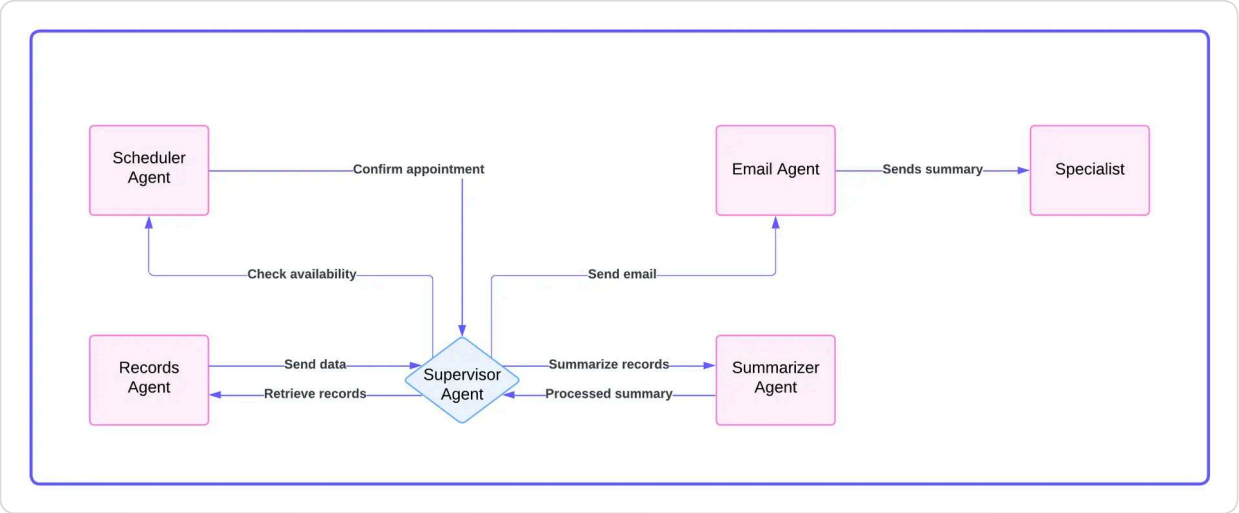
The supervisor pattern

The supervisor pattern is one of the more commonly used multi-agent architectures, and the core idea is straightforward: A single agent takes the lead. The supervisor agent receives a trigger, breaks the task into sub-tasks, and delegates each to a specialized agent, then ensures that agents run in the right order, with the right context, and that the output flows back properly.

A practical application of this architecture might be a specialist appointment system for hospitals. When a patient initiates a request, the supervisor agent takes control of the entire workflow:

- First, it checks availability through a scheduler agent.
- Next, it retrieves the patient's medical records via a records agent.
- Then, it sends the records to a summarizer agent to generate a concise overview.
- Finally, it forwards the final summary to an email agent responsible for notifying the specialist.

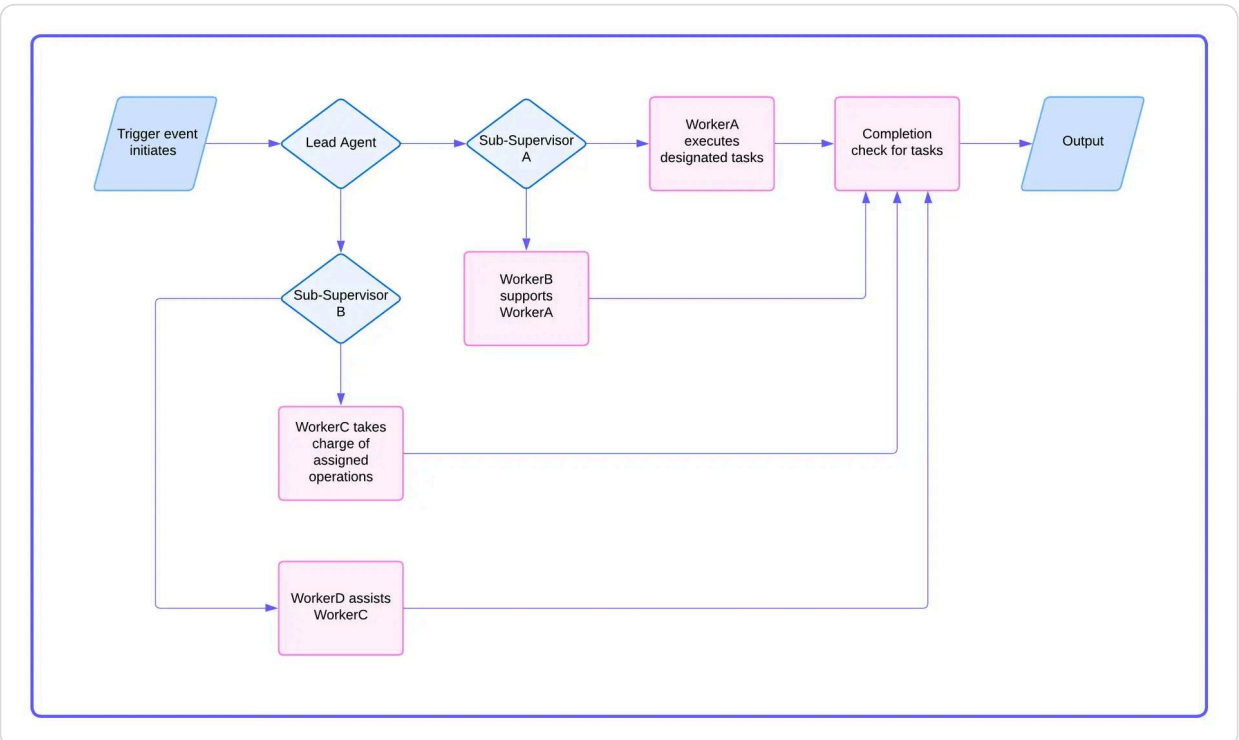
The supervisor pattern in a hospital appointment system might look like this:



The hierarchical pattern

An extension of the supervisor pattern, the hierarchical pattern is used when tasks are too complex or broad to be managed by a single supervisor. It introduces layers of coordination: A top-level agent handles the high-level goal and delegates parts of it to mid-level agents, which further break the work down and assign tasks to lower-level agents. This approach is useful in systems where responsibilities must be split across specialized teams or domains.

For example, in an enterprise document processing system, a top-level agent may oversee the entire pipeline, delegating summarization to one mid-level agent and data extraction to another, each of which manages its own group of workers.

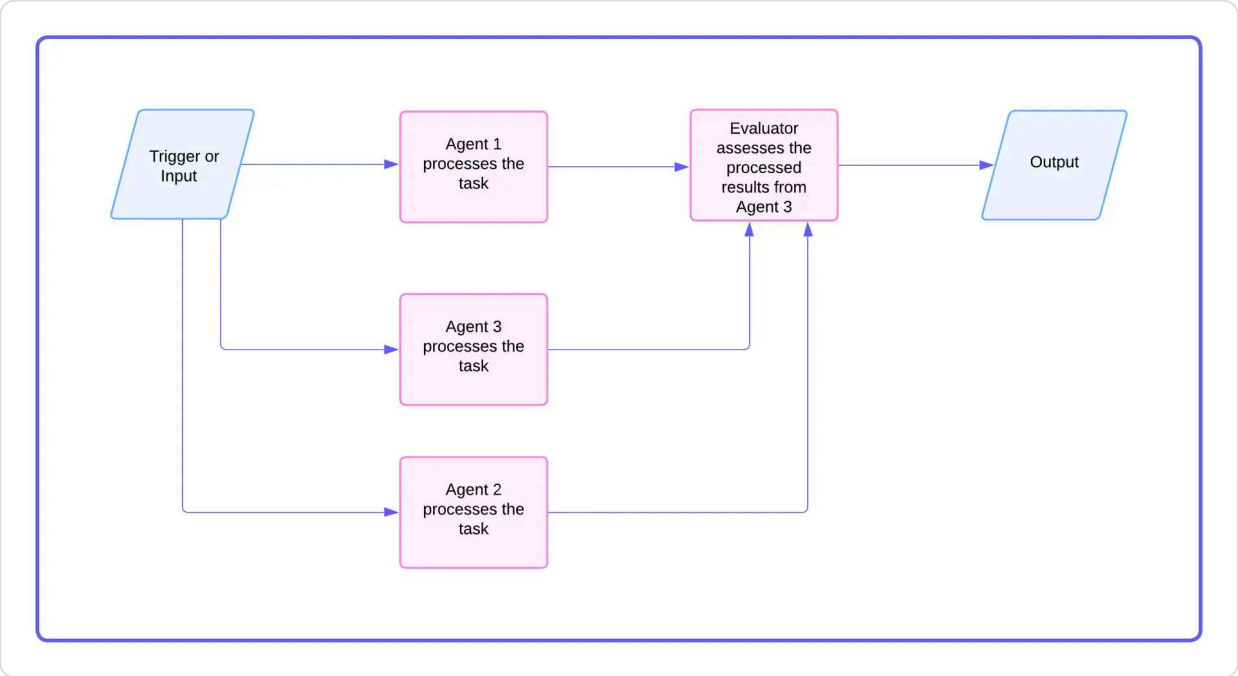


The competitive pattern

The competitive pattern involves multiple agents independently working on the same problem, each proposing its own solution. A separate evaluator agent reviews all submissions and selects the most suitable one based on predefined criteria such as speed, accuracy, creativity, and cost-efficiency.

This approach is useful when diversity of thought or redundancy can lead to better outcomes. It also adds robustness: If one agent fails or performs poorly, others may still succeed.

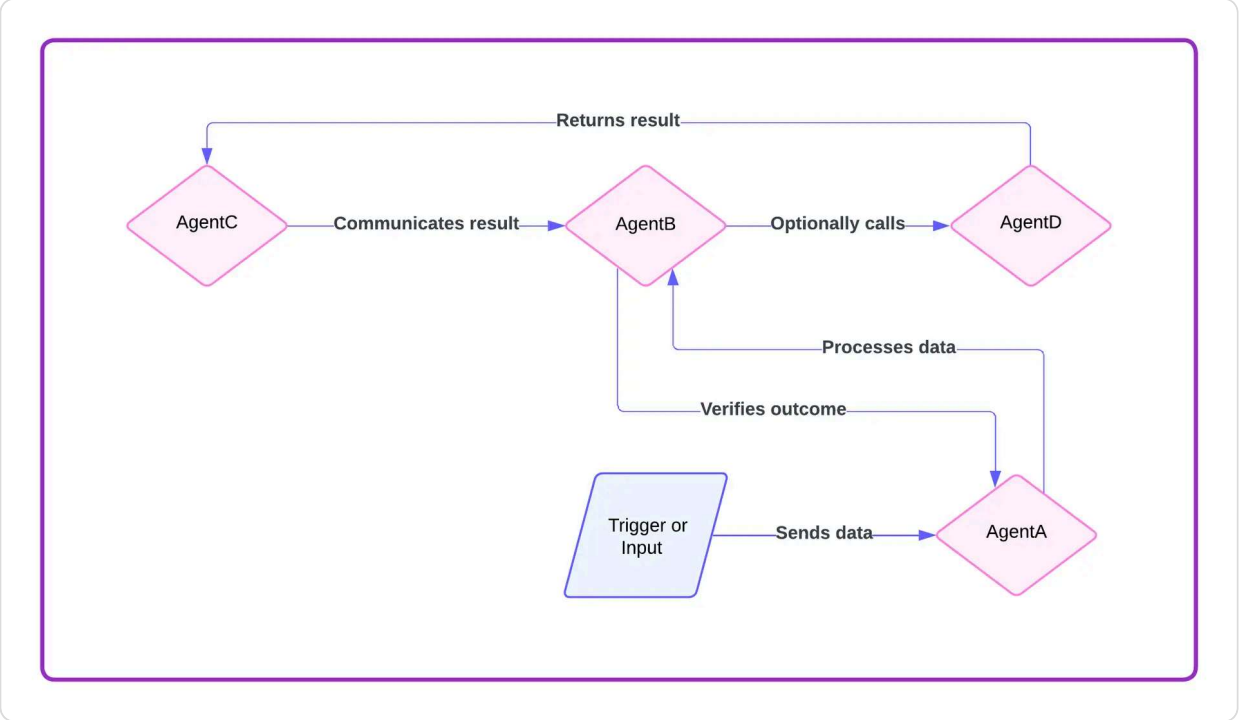
A typical use case for the competitive pattern is generating marketing copy: Several agents generate different headlines or content snippets, and an evaluator chooses the one that best fits a set of brand guidelines through A/B test criteria.



The network pattern

The network pattern has no lead agent. Each agent has its own tools and communicates directly with others to coordinate tasks. Frameworks like OpenAI Agents and Crew AI are designed around this model.

While the network pattern offers flexibility, it often proves impractical in real-world applications. Without a clear flow, agent-to-agent communication is unstructured, making the system hard to debug, unreliable, and costly to run. Each step may trigger an additional LLM call, increasing latency. For these reasons, the network pattern is generally not suited to production use, unless you specifically need a decentralized setup, such as in research or simulation environments.



Final thoughts

Once you understand the architectural options for building agentic systems, the next step is choosing the one that fits your use case.

We've covered a range of architectural patterns, from simple single-agent setups to more complex multi-agent designs. Our final recommendation is this: If your application follows a multi-agent architecture, consider including a **reflective agent.** While multi-agent systems benefit from improved precision, reasoning, parallelism, and task specialization, adding a reflective agent introduces a feedback loop that enables the system to learn and improve over time. This is one of the most effective ways to take advantage of the adaptive potential of AI.

If you're building a simple MVP or prototype, start with a **tool-using agent**. It's practical, fast to implement, and already supports memory and tooling. Add an MCP layer for external API access, and you can move quickly.

If your application calls for a multi-agent setup, the supervisor pattern is usually the best place to start. Other multi-agent patterns (like network, hierarchical, and competitive) can be useful, but only in specific contexts:

- Choose the **competitive pattern** when you want multiple agents to propose different solutions and compare the outcomes.
- Choose the **hierarchical pattern** when tasks are complex and need to be divided across worker groups under sub-supervisors.

Here's a quick reference to help you choose the right architecture for your use case.

Criteria	Recommended architecture
MVP, prototype, or linear automation	**Single-agent:** Tool-using pattern
Simple task needing context or history	**Single-agent:** Memory-augmented pattern
Workflow requiring planning or sequencing	**Single-agent:** Planning-agent pattern
System needing self-correction or improvement	**Single-agent:** Reflection-agent pattern or **Multi-agent:** Reflective agent
Complex workflow requiring task delegation	**Multi-agent:** Supervisor pattern
Workflow requiring many workers per task	**Multi-agent:** Hierarchical pattern
Need for diverse solutions and output comparison	**Multi-agent:** Competitive pattern
Decentralized system for research or benchmarking	**Multi-agent:** Network pattern