How to Choose the Right Architecture to Build AI Agents

5 June 2025 - ID G00826183 - 12 min read

By Tigran Egiazarov, Gary Olliffe, and 4 more

AI agents' high business potential has sparked interest in the technologies and architecture choices for their development. Software engineering leaders should use this research to grasp key components and patterns, aiding their teams in making the right architecture decisions for AI agents.

Overview

Key Findings

- The development of large language models (LLMs) has reached a more advanced stage, and when combined with technologies that improve grounding, like retrieval-augmented generation (RAG) or model fine-tuning, these models can significantly boost the capabilities of AI agents. Without the right architecture decisions and understanding of key components of AI agents, organizations will fail to deliver a functional AI agent that adds value to the business.
- Software engineering teams who lack a thorough understanding of AI agent architecture patterns risk not only implementing unnecessarily complex AI agents but also creating flawed architectures. This can lead to issues with AI agent performance, inflexibility, coordination, security, and technical debt, potentially paralyzing the future delivery of AI agents.

Recommendations

- Streamline AI agent development by establishing a well-defined architecture built around fundamental components necessary for creating sophisticated AI agents.
- Prevent delivery failures of AI agents by adopting core architecture patterns that minimize initial costs and ensure a consistent architecture that supports the delivery of business value.

Strategic Planning Assumption

By 2028, 80% of organizations will report that AI agents consume the majority of their APIs, rather than developers.

Introduction

Interest in AI agents has skyrocketed over the last 12 months, and many organizations are evaluating and building AI agents. There is a steady stream of new technologies being launched to support AI Agent development.

AI agents are autonomous or semiautonomous software entities that use AI techniques to perceive, make decisions, take actions and achieve goals in their digital or physical environments.

The Gartner Software Engineering Survey for 2025 shows that “building AI capabilities into applications” is among both the top priorities and top challenges for software engineering leaders. [1] Given the pressure from stakeholders demanding solutions to business problems they imagine AI agents can address, many software engineering teams will inevitably start building AI agents.

However, without a good understanding of AI agent architecture, software engineering teams will make suboptimal choices shaped by stakeholder pressure and the rapid growth of the

technology. This will inevitably lead to failed deliveries of working AI agent solutions and technical debt.

This research outlines architecture strategies for AI agents. Software engineering leaders can effectively guide their teams in selecting the right fundamental components and architectural patterns for building AI agents, avoiding delivery failures and technical debt and navigating stakeholder pressure.

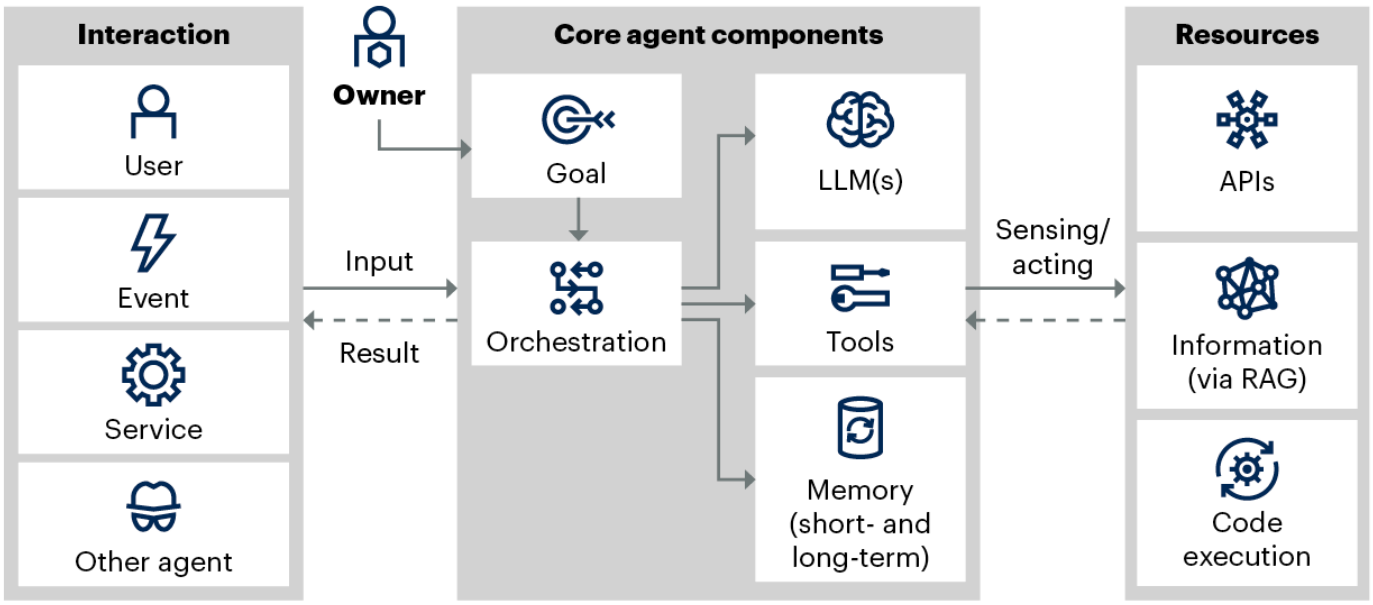
Analysis

Establish a Well-Defined AI Agent Architecture

The success of AI agent delivery relies on a well-defined architecture that ensures scalability, modularity and adaptability. AI agents interact with their environment and users to achieve defined goals by “evaluating” the inputs, reasoning about tasks, planning actions, and executing tasks using internal and external tools. Figure 1 outlines the AI agent architecture components for LLM-based AI agent architecture.

Figure 1: Simplified Architecture of an LLM-Based AI Agent

Simplified Architecture of an LLM-Based AI Agent



Source: Gartner
823011_C

Gartner

Software engineering leaders should direct their teams to master the main architecture components of an AI agent.

Interactions

This component diverts inputs that trigger an agent's reasoning and lets the agent perform the action on behalf of the caller. These inputs include direct user prompts, events and inputs from other AI agents. For example, a user might ask a customer service AI agent to initiate a refund, or an event from a monitoring system might notify an agent of a server failure. The input can be in any format, like a Kafka event, a REST API request, or a request received via Agent2Agent (A2A) protocol.

Core Agent Components

The minimum functional part of an AI agent comprises the following components.

Goal

This is the agent context in which agent should operate. The goal should include the role of the agent and its scope. In real-world examples, this could involve defining a sales support agent's objective to qualify prospects based on some criteria and schedule sales meetings.

Orchestration

This component acts as the central coordinator, managing AI agent workflows and determining the sequence of operations needed to achieve a goal. It ensures efficient communication between the LLM, tools and memory, allowing the agent to process complex tasks systematically. Technologies and frameworks like AutoGen by Microsoft, CrewAI and LangGraph by LangChain introduce a structured way to not only define agents but also enable teams to organize agents into multiagent systems by assigning roles and managing dependencies for distributed tasks completion.

AI Model (e.g., LLM)

The model serves as the reasoning engine, generating responses, interpreting queries and understanding context. For example, LLMs such as OpenAI GPT-4, Anthropic's Claude or open models like Meta Llama 3 act as the agent's inference engines. They use pretrained knowledge and real-time inputs to adapt to different situations, providing the semantic richness that elevates the AI agents beyond static bots. It is important to note that while LLM-based AI agents perform reasoning through LLMs, not all AI agents are LLM-based. The LLM(s) handles reasoning and response generation, but other types of AI agents can function via reinforcement learning or knowledge graph techniques. Reinforcement-learning-based agents use reinforcement learning techniques to optimize and learn strategies through interactions with users, environment and other agents.

Tools

Tools extend the agent's capabilities to perform tasks beyond pure text generation, making it more versatile and useful in various scenarios, such as calling APIs, supporting Model Context Protocol (MCP), querying databases and even executing generated code. Example implementations include calling a CRM system's API to update customer data, querying a Postgres database for real-time inventory levels or executing Python code in a sandbox environment such as serverless runtimes.

Memory (Short- and Long-Term)

Memory provides continuity in interactions by storing previous conversations, facts and contextual details. Short-term memory helps maintain relevance within a single session, while long-term memory allows the agent to learn and evolve over time. Vector databases are commonly used to implement long-term memory.

Resources

This is an integral set of mechanisms which an AI agent can use to interact with the environment, discover and retrieve additional data and external knowledge, and execute tasks. Such tools extend the capabilities of AI agents via the ability of calling APIs, leveraging RAG pipelines for sourcing unstructured information to reinforce the initial input, or connecting to MCP services, which serve up internal tools and data sources. For more details on MCP, see **Innovation Insight: Model Context Protocol**.

To summarize, the components outlined in Figure 1 form the blueprint for development of scalable and extensible AI agents. AI agent development frameworks like Autogen, CrewAI or LangGraph, as well as AI agent platforms like Salesforce Agentforce, Microsoft Copilot Studio or AWS Bedrock, can accelerate the development of AI agents. Success, however, still depends on rigorous architecture design: modularization, clear orchestration, memory governance and secure resource access. For more details on AI agent technologies and frameworks, see **How to Choose the Right Technology to Build LLM-based AI Agents.**

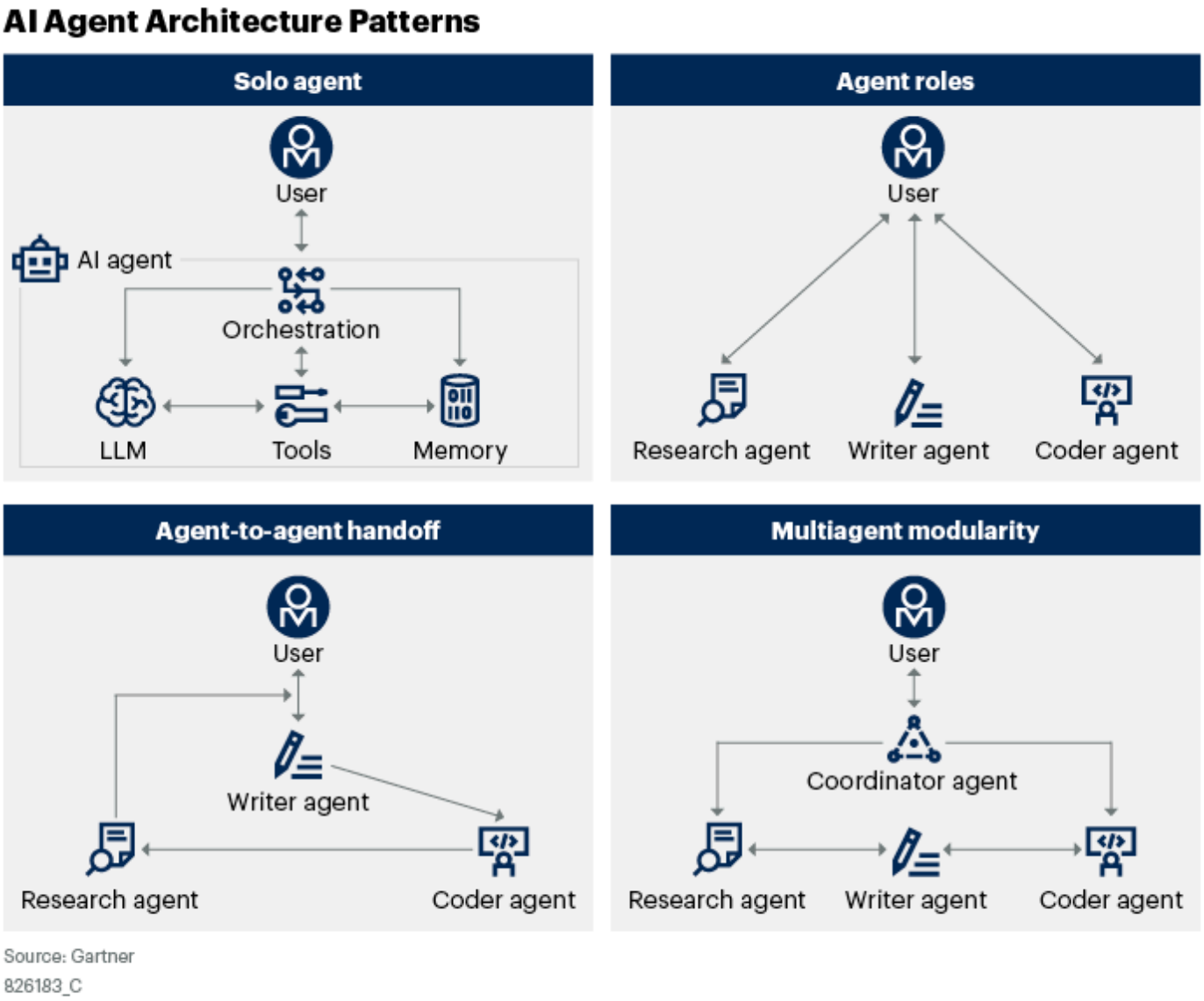
Adopt AI Agent Architecture Patterns

Choosing the right AI agent architecture patterns is critical to reach the targeted efficiency, scalability and adaptability. Your selection depends on the complexity of the tasks, the degree of automation required, and the needed integration into existing software systems.

Each of the below AI agent architecture patterns — solo agent, agent roles, agent-to-agent handoff, and multiagent modularity — offers distinct benefits and trade-offs (see Figure 2). In practice, they are not mutually exclusive (for example, a multiagent modularity may incorporate agent-to-agent handoffs as part of its design).

Software engineering leaders should direct their teams to choose the right patterns that fit their product complexity and goals. Start simple if unsure (it's often wise to prototype with a solo agent or minimal number of agents, then expand).

Figure 2: AI Agents Architecture Patterns



Solo Agent

Apply this pattern when the task is simple or self-contained and suitable for initial deployments. If a single agent can effectively handle the functionality, such as answering FAQs or generating reports, and the scope does not require diverse specialties, then the solo agent approach is appropriate. For instance, a single chatbot can address common customer inquiries or produce daily summaries, thereby avoiding the complexity and overhead associated with managing multiple specialized agents.

The benefits of a solo agent are architecture simplicity and maintainability, with no interagent communication complexity or orchestration needed. All logic resides in one agent's memory, avoiding the overhead of context sharing.

On the other hand, a solo agent has limited ability to perform complex or multifaceted tasks that exceed the agent's goal, potentially yielding mediocre results on tasks requiring diverse expertise.

Agent Roles

Apply this pattern when the task is complex and requires decomposition into subdomains, each benefiting from the AI agent specialization. For example, the agent roles pattern can be used to build multiple agents in the e-commerce shopping experience. This could be a product recommendation agent that suggests items (trained on product data), a customer service agent that answers order questions, and an inventory agent that checks stock levels.

Be aware that modularity comes with a cost, as ensuring AI agents understand the overall task context and pass along necessary information is crucial. This pattern is usually combined with a multiagent modularity or agent-to-agent handoff pattern to manage the coordination between the AI agents. For smaller or simpler tasks, this added complexity might not be worth the benefit — there's a risk of over-engineering a solution that a solo agent could handle.

Agent-to-Agent Handoff

Apply this pattern in multistep workflows or "escalation" scenarios where no single agent can handle everything. In such scenarios, tasks have to be handed over to a more specialized agent by a less specialized agent. For instance, a mobile banking AI agent within a mobile banking application handles routine queries (balance checks, FAQs). However, for more specialized, complex queries, such as payment transaction disputes, the mobile banking AI agent hands off the issue to a specialized dispute resolution AI agent, or even directly to human customer support representatives.

A major challenge is preserving context across handoffs. The receiving agent must understand what the previous agent accomplished and what requirements remain. If information isn't transferred completely, the second agent might ask the user to repeat info or, worse, make incorrect assumptions. In some complex scenarios, an agent-to-agent handoff requires a central coordinator entity, which can be implemented via Multiagent modularity.

Multiagent Modularity

As the complexity of agents' flow of activity to complete a task or process grows, it becomes important to coordinate agents and plan complex execution steps through a centralized coordination layer.

Apply this pattern for complex, large-scale systems where many different AI agent specializations and capabilities are needed. Coordination of such a system requires one or multiple coordination agents which ensure the execution of the tasks toward the end goal. For example, a user's questions may contain both a balance check and a dispute request. If the balance is below the expected value and the payment transaction is still not canceled, both the mobile banking AI agent and the dispute resolution AI agent will contribute responses and execute actions through a coordinating AI agent.

Designing a modular AI agent system with a coordinating layer requires the usage of communication protocols, such as Google's **Agent2Agent Protocol (A2A)**, as well as a complex context management system. A centralized coordinator agent manages workflows, sequencing and dependencies between specialized AI agents. It often involves state management to track ongoing processes across multiple agents.

Action Items

- **Start simple.** Pilot with a solo agent to test ideas and build internal capabilities.
- **Define boundaries.** As requirements grow, expand to multiagent systems to address more complex business tasks. This is ideal for distributed agents where different agents specialize in distinct tasks. AI agents with different roles and levels of granularity collaborate to complete tasks.
- **Invest in AI Agent orchestration.** Enhance flexibility and problem-solving by allowing multiple agents to interact dynamically. This requires efficient coordination mechanisms for task delegation, state management and execution flow. As systems scale, you will need a robust framework for planning, coordination and monitoring of AI agents.
- **Enable collaboration.** Use these patterns to align responsibilities across engineering teams.

Be aware that building complex, distributed AI agent systems requires significant investment in API-driven orchestration, as nondeterministic workflows can impact performance. While monolithic agents manage state within their own orchestration, distributed agents require

external orchestration, well-maintained state management and efficient context handoff between the agents. Additionally, distributed AI-agent architectures add deployment complexity, which demands strong DevOps practices for scalability, monitoring and automation.

To tackle the coordination, complexity and communication challenges in AI agent design, please see **Emerging Patterns for Building LLM-Based AI Agents**.

⊕ Evidence