Multi-Agent Reference Architecture

This repository presents a conceptual guide, complemented by practical resources, for architecting robust multi-agent systems. The focus is not on building an individual agent, but on the unique challenges and effectiveness of orchestrating, governing, and scaling systems where multiple specialized agents interact to solve complex problems. You will find actionable guidance for *designing for change*, balancing long-term extensibility with pragmatic, shipping-first engineering.

The recommendations are grounded in production-scale, real-world solutions built in collaboration with Microsoft customers. As such, the approaches offered in this reference are both opinionated (benefiting from field experience) and agnostic (applicable across enterprises, technology stacks, and frameworks).

This guide is intended for software architects, software engineers and data scientists familiar with agentic services design and development. It is aimed at those with experience in building and deploying agents, whether they aim to extend existing systems to multi-agent architectures or build them from the ground up.

> **Note:**
> Generative AI is advancing rapidly, with new models, patterns, protocols and paradigms constantly emerging. While the current design is intentionally agnostic and broad, we expect to refine and improve it as the ecosystem matures.

If you want to jump straight to the architecture reference, check out the Reference Architecture chapter. Otherwise, if you'd like to explore the concepts and recommendations in more detail, just keep reading the next chapters.

Table of Content

- Observability
- Evaluation
- Security
- Governance
- Reference Architecture

Introduction

Last updated: 2025-05-16

Generative AI is shifting rapidly from research to production, with enterprises seeking robust, maintainable and scalable solutions to solve complex problems. In this landscape, the design of multi-agent systems, where numerous specialized AI agents cooperate to solve problems, has become critically important: enables modularity, domain expertise, and agility, offering adaptability as business needs and AI capabilities evolve.

These systems reflect the natural structure of organizations: different roles, responsibilities, and domains mapped to individual or composite agents, each optimized for specific knowledge or workflows.

Design Principles

The following design principles are especially crucial for multi-agent systems, as identified through real-world implementations in large enterprises. These principles address challenges unique to environments where multiple agents interact, collaborate, and exchange information. Treat them as guiding foundations rather than rigid requirements:

> This repository is continuously maintained by contributors to provide the community with the most up-to-date guidance, grounded in real-world enterprise experience and aligned with the latest developments in multi-agent systems. We welcome your feedback, suggestions, and references to additional resources that could enrich this documentation. As we evolve this reference architecture, community input plays a vital role in shaping its relevance and impact.

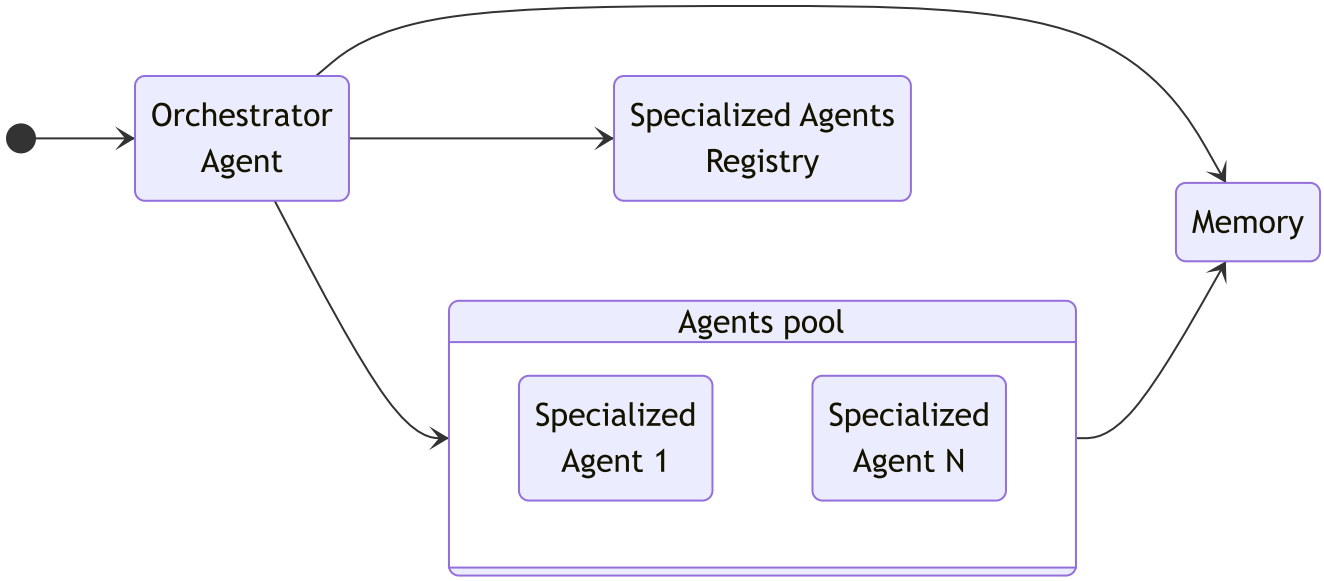
Name	Description
Separation of Concerns	In multi-agent systems, it's essential that each agent has a distinct and well-defined responsibility. This clarity enables focused development, scalable deployment, reduced cross-cutting changes, and makes it possible for agents to encapsulate deep domain expertise.
Secure by Design	With multiple agents communicating and acting within the same ecosystem, robust security is paramount. This includes strict authentication, authorization, and policy enforcement for every

Name	Description
	agent interface. It's also necessary to carefully manage sensitive data flows between agents, minimize data retention, and clearly define data handling practices to prevent unintended leakage or escalation of privilege.
Observability & Traceability	Multi-agent workflows generate complex, interdependent behaviors. Agents should be instrumented so their actions, data exchanges, and decisions can be traced end-to-end across the system using common identifiers and correlated metrics. This level of observability enables efficient troubleshooting, auditing, and deep understanding of system operations—far beyond what is required in single-agent scenarios.
Agent Registration & Lifecycle Governance	Agents should be explicitly registered, versioned, and validated before being included in production environment. Registration should capture agent capabilities, security posture, and lifecycle state to prevent duplication, control upgrades, and reduce risks from rogue or malfunctioning agents.
Failure Isolation & Graceful Degradation	Failures in one agent should not cascade to others. Consider fallback mechanisms, retries, or degraded modes of operation to ensure the workflow can continue, even in the presence of partial failures.
Context Management	Establish clear rules and policies regarding what context or conversational state is shared among agents and for how long. Carefully control context propagation to avoid privacy issues, data leakage, or logic confusion.

Building Blocks

Last updated: 2025-05-28

Core components



- `Orchestrator agent` acts as the central coordinator within the multi-agent workflow. It receives incoming requests—whether from users or external services—and formulates a high-level plan composed of individual tasks. These tasks are then delegated to appropriate specialized agents, each responsible for a specific function. Once the specialized agents complete their tasks and return their results, the orchestrator aggregates and synthesizes their outputs into a coherent final response.

> Although direct communication between specialized agents might seem convenient at first, it's recommended to route all communication through the orchestrator to maintain clarity and control. Avoid direct agent-to-agent messaging unless necessary; if agents are tightly coupled, group them into a single composite agent exposed to the orchestrator as one unit.

- `Specialized agents` are domain-focused experts within the multi-agent system. Each one is responsible for a distinct area of expertise—for example, one agent might be responsible for finding flights, another for booking hotels, another for planning activities, and another for estimating the total cost. These agents can range in complexity: some

may be simple wrappers around a language model, while others may coordinate their own sub-agents to complete more detailed tasks. For instance, a "Hotel Booking Agent" might internally manage sub-agents for comparing prices, checking availability, and reading reviews—working together to deliver the best lodging option.

> A specialized agent is not the same as a task or a tool. While an agent may use multiple tools or perform several steps to achieve a complex goal, it's a common mistake to treat tiny, granular tasks—like "call an API" or "parse a response"—as standalone agents. This creates unnecessary fragmentation and complexity. Instead, agents should be structured around meaningful, cohesive capabilities—not isolated tool calls.

- `Agents registry` serves as a centralized data service and source of truth for managing all specialized agents available in the system for the orchestrator. It functions like a dynamic directory—cataloging each agent's identity, capabilities, operational status, version, and metadata tags. This registry enables efficient discovery and auditability.

Additional reference of orchestration patterns: Semantic Kernel: Multi-agent Orchestration

---

Multi-agent Design Options

Last updated: 2025-05-15

This chapter explores architectural strategies for building multi-agent systems. Here we break down two primary design approaches:

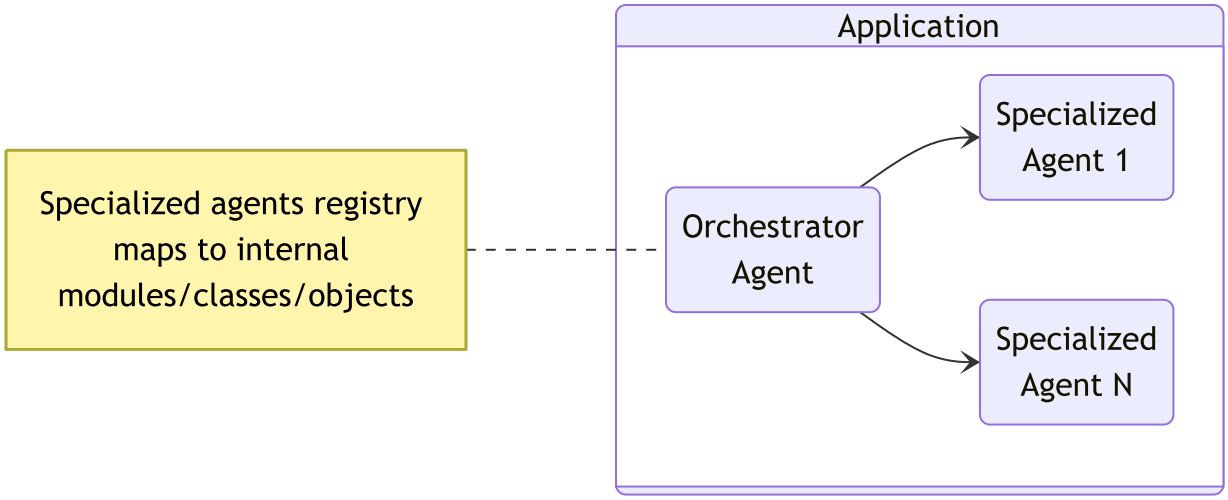
- **Modular Monolith**: A standalone application where orchestrator and specialized agents are organized as well-defined modules. This approach emphasizes simplicity, shared memory, and low-latency communication.
- **Microservices**: A distributed system where each agent (or group of agents) is encapsulated as a service. This model enables independent deployment, granular scalability and flexibility to use different tools, frameworks, or programming languages for each service.

Each approach has trade-offs in terms of performance, scalability, maintainability, team coordination, and operational complexity. This documentation explores those trade-offs and offers guidance to help you choose and implement the right architecture for your multi-agent system—based not only on technical goals, but also on your organization's structure and team dynamics.

Agents as a Modular Monolith

Last updated: 2025-05-15

In the modular monolith design, all components—the orchestrator and each specialized agent—reside within a single codebase and are deployed as one application. While they are logically decoupled as independent modules or libraries, physically these agents share a process space, a datastore, and often common infrastructure like caching and logging.



Key Characteristics

- **Simplicity of Operation**: With a single codebase, deployment artifact, and process to manage, this architecture radically lowers operational complexity compared to distributed systems.
- **Shared Infrastructure**: Agents can efficiently share memory, storage, logging, tracing, and context—enabling extremely low-latency communication patterns and reduced duplication of concerns.
- **Unified Governance**: Policy enforcement (such as agent registration and AI safety controls) and versioning are inherently centralized, simplifying traceability and compliance.
- **Ease of Observability**: Cross-module instrumentation and debugging are straightforward; correlation of traces and logs is seamless as everything runs together.

Trade-Offs

Advantages

- **Rapid Iteration**: Teams can develop, test, and deploy updates quickly, as changes to any agent or the orchestrator are integrated together and can be validated as a cohesive whole.
- **Lower Overhead**: No network-induced failure modes or communication latency between agents; debugging tools and traceability are easier to implement and use.
- **Centralized Security**: Security controls such as authentication, authorization, and firewalling are easier to enforce in a centrally managed process.

Disadvantages

- **Scalability Constraints**: The entire application must scale together. If one agent or workflow becomes a performance bottleneck, you cannot scale or optimize it independently. Memory or compute spikes from one agent can degrade service for all.
- **Deployment Coupling**: Any change—no matter how minor—requires a full deployment cycle for the whole system. This can slow down delivery if multiple teams work in parallel or have divergent release cadences.
- **Limited Technology Flexibility**: All agents must use the same language, framework, and underlying platforms; integrating agents built in other stacks (or by external teams) is challenging.
- **Additional engineering discipline required**: Without strict code discipline, teams may inadvertently introduce dependencies between modules, increasing coupling and reducing long-term maintainability.

Onion (Layered) Architecture Within a Modular Monolith

The **Onion Architecture** (also known as Layered Architecture) is a powerful structural pattern that can be applied *within the modular monolith*. While the monolith organizes code into independent business or domain modules (such as specialized agents or orchestrators), onion architecture further enforces **separation of technical concerns within each module** by defining clear, nested layers:

- **Orchestration Layer** – Handles workflow and high-level coordination.
- **Specialized Agent Layer** – Implements the core agent-specific business logic.

- **AI Layer** – Interfaces with language models and related AI components, providing a unified API for all AI tasks.
- **Knowledge Layer** – Interfaces with AI models, vector stores, files, web data, data analytics platforms, and more.
- **Storage Layer** – Manages persistence, data retrieval, and caching.
- **Integration Layer** – Connects the system to external APIs and services.

Each layer has **bounded responsibilities** and interacts through well-defined APIs, with strict rules: outer layers depend only on the inner ones, never the other way around.

Modular Monolith as a Migration Path

One of the strongest arguments for starting with a modular monolith is its role as a stepping stone to microservices. By building clear, well-factored module boundaries, explicit interfaces, and decoupled business logic, you make future extraction of agents (or even the orchestrator) into standalone services far simpler.

Recommended practices for a future-proof monolith:

- **Encapsulate agent logic** in independent modules with well-defined interfaces.
- **Avoid direct cross-module calls** beyond published interfaces, even inside a single codebase.
- **Enforce dependency direction** (e.g., orchestrator knows about agents, not vice versa).
- **Isolate state and configuration** per agent, simulating what a future service boundary would require.
- **Log and trace** with per-agent identifiers to facilitate distributed tracing down the line.

This discipline makes your code stronger, easier to evolve, and prepares you for a future where modules might become services.

Teams that start with a modular monolith can later extract complex agents into separate microservices with minimal effort. This approach lets you adopt microservices only when your product's scale or your team's needs truly justify it.

When Is a Modular Monolith Best?

- Early to mid-stage multi-agent systems, or those unlikely to face massive horizontal scaling pressures.
- Organizations with a single team, or closely collaborating teams, who can work efficiently inside a single codebase.

- Use cases where low-latency coordination and shared state are essential, but team autonomy and independent deployments are not the top concerns.

References

- Monolith First by Martin Fowler
- Microservices for Greenfield?
- The Onion Architecture

Agents as Microservices

Last updated: 2025-05-16

In the microservices design, each component—such as the orchestrator and every specialized agent—is encapsulated as an independent service running within its own process, potentially on separate machines or cloud services. These agents interact exclusively via well-defined APIs or messaging protocols. This distributed approach is a natural fit for complex multi-agent systems where the demands for scalability, maintainability, flexible technology stacks, and deployment autonomy outweigh the operational overhead of distributed systems.

Key Characteristics

- **Service Independence**: Each agent is developed, deployed, and scaled independently. Teams may choose distinct languages, frameworks, or even cloud platforms for each service.
- **Technology Flexibility**: Microservices enable teams to choose the most appropriate technologies for each service, rather than being constrained to a single tech stack. This allows agents developed by different teams—or even entirely separate organizations—to interoperate seamlessly. The architecture enables rapid innovation and the integration of specialized solutions tailored to the specific needs of each service.
- **Distributed Infrastructure**: Each service manages its own infrastructure resources (memory, storage, replicas), yet they collaborate through network communication (e.g. REST, gRPC, message queues).
- **Decentralized Governance**: Policy enforcement, agent governance, and versioning are implemented via inter-service contracts and registration protocols rather than assumed by process boundaries alone.
- **Observability across services**: Logs, traces, and metrics must be correlated across process and service boundaries for effective troubleshooting and monitoring.

Trade-Offs

Advantages

- **Scalability & Resilience**: Each agent or workflow component can be scaled independently, reducing the impact of failures and enabling cost-efficient scaling tailored

to each workload.
- **Deployment Autonomy**: Teams can deploy updates to each agent without redeploying the entire system, enabling continuous delivery and faster iteration cycles.
- **Bounded Contexts**: Agents encapsulate their own logic, state, and dependencies, reducing unintentional coupling and easing long-term codebase evolution.
- **Technology Diversity**: Teams can explore and adopt a variety of tools, language and libraries for each agent—freeing teams from monolithic technology decisions.

Disadvantages

- **Operational Overhead**: There is significant complexity in managing, monitoring, and securing distributed services (service discovery, API gateways, network policies, etc.).
- **Increased Latency**: Inter-agent communication occurs over the network, introducing latency and potential failure modes compared to in-process calls from monolithic services.
- **Consistency and State Sharing**: Sharing memory or context across agents is nontrivial and may require specialized infrastructure (e.g., distributed caches, messaging system, etc).
- **Complex Governance & Security**: Security policies must be enforced at network and application levels; clear agent registration and health-check protocols must be established to avoid ambiguity and reduce attack surface.

Layered Patterns Within Microservices

While orchestrator and specialized agents are physically separated into their own services, the **Onion (Layered) Architecture** can still be implemented *within each service*. This practice enforces internal discipline, clear separation of responsibilities, and maintainability:

- **API Layer** — Exposes network endpoints (REST, gRPC, etc.) for agent operations.
- **Service Layer** — Implements the agent's core business logic (task delegation, reasoning, workflow, etc.).
- **AI Layer** — Connects to language models, RAG pipelines, or other AI infrastructure, abstracting external dependencies.
- **Knowledge Layer** — Handles integration with vector databases, contextual knowledge stores, file systems, or external data sources.
- **Persistence Layer** — Manages local or distributed storage, caching, and transaction boundaries as needed by the agent.
- **Integration Layer** — Connects to external APIs, services, or third-party platforms as needed for agent tasks.

Within each service, these layers interact via internal contracts—mirroring the clean boundaries needed for future maintainability.

When Is a Microservices Approach Best?

Many successful systems begin as modular monoliths and evolve by gradually extracting agents or the orchestrator into services as scale, team size, or business requirements change. This staged approach helps avoid costly over-engineering early on, while preserving the option to scale, diversify, and federate component ownership later.

Consider starting with microservices in the following situations:

- For mature multi-agent systems operating at significant scale, or with non-uniform workload patterns requiring independent scaling or cost optimization.
- In organizations with multiple teams, or those desiring to onboard/exchange third-party agents built in divergent technology stacks.
- For scenarios where team autonomy, deployment velocity, and continuous evolution of individual agents or workflows are top priorities.
- When strong, observable governance and authentication boundaries between agents are required (e.g., regulated or sensitive domains, or when hosting agents from external parties).

Recommended practices:

- Design agents with explicit APIs and versioning from the start.
- **Adopt service discovery and registration mechanisms** to identify which agents are available at runtime and to ensure orchestrator-to-agent compatibility.
- **Enforce strict network boundaries:** Control agents communication from the networing perspective (e.g. specialized agents only communicating with the orchestrator, not directly with each other). This provides clear workflow governance, traceability, and better security.
- **Implement robust observability:** Use distributed tracing standards (e.g., OpenTelemetry) and log correlation mechanisms to maintain visibility across the system.
- **Plan for graceful failures** and fallback strategies, as network partitions, agent unavailability, and other distributed failure modes are inevitable.

References

- Microservices Guide

- Microservices Patterns

Discuss this page

Agent Registry

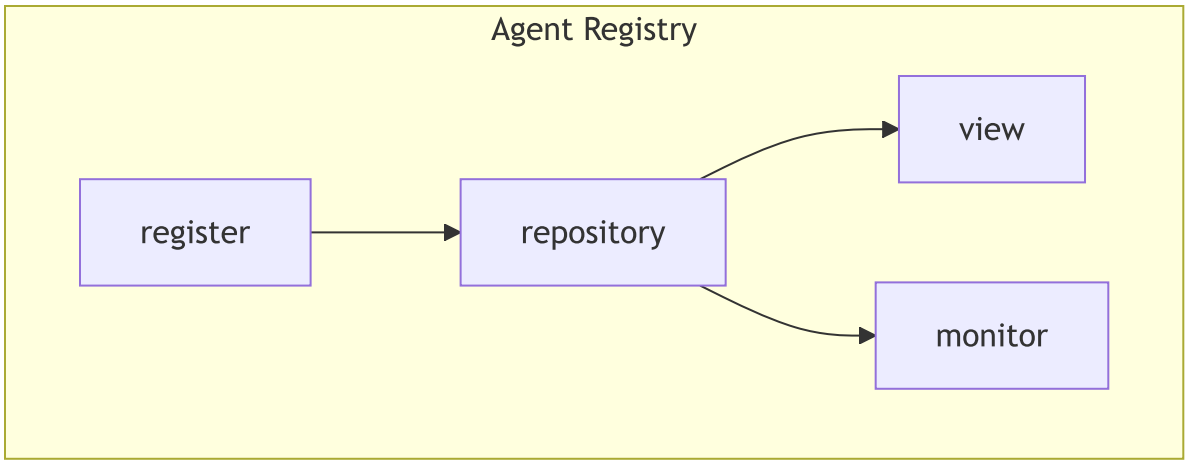
Last updated: 2025-05-15

Agent Registry is the component that contains the information regarding available agents in the multi-agent system.

Depending on the architecture of the multi-agent system (monolithic or distributed), Agent Registries can be centralized, or distributed per agent or agent group.

The primary goal of an agent registry is to provide an information repository for the agents in the system to know how to communicate with one another. In most cases though, in an Orchestrator-based multi-agent architecture, the registry enables the Orchestrator Agent to query which agents are best to carry the immediate tasks at hand.

Core Components



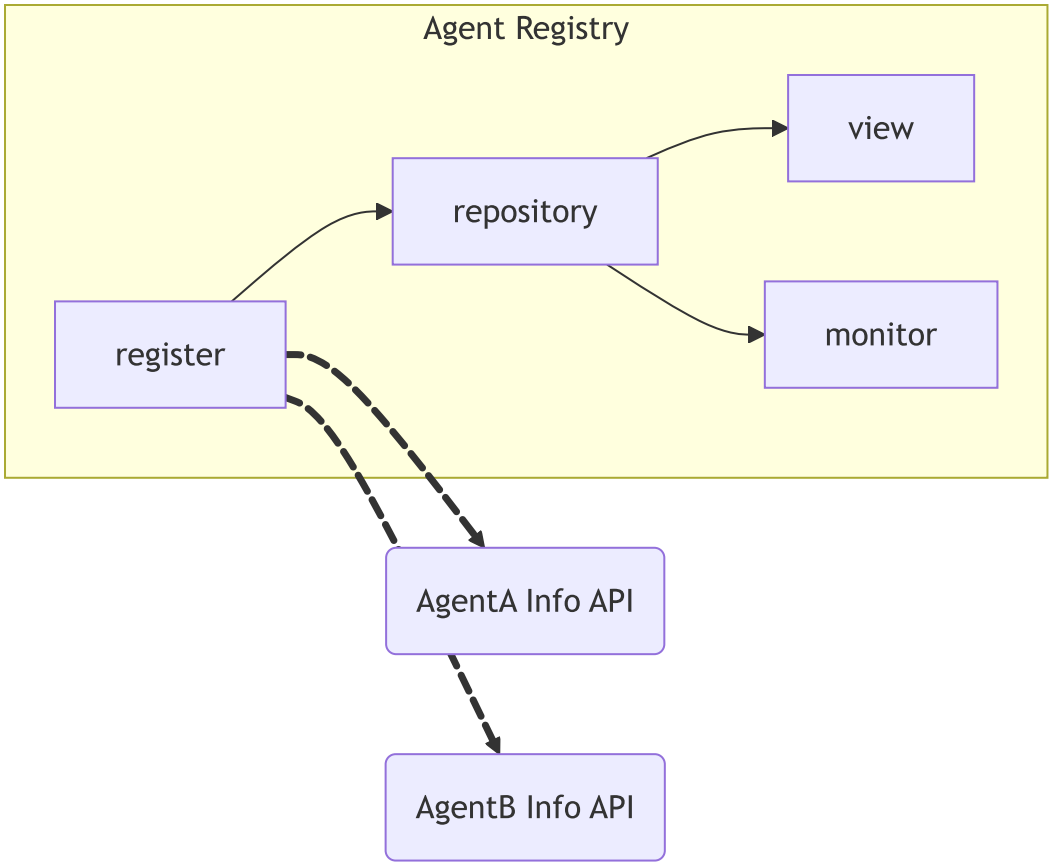
Register Component (Discovery)

A register mechanism is necessary to input Agent information into the repository. There are several ways to implement this Register mechanism

Registry-Initiated Agent Register

In a scenario where the Agent is available, and has a way to get the Agent Information, the register mechanism can make a request to the target agent URL to get the Agent Information.

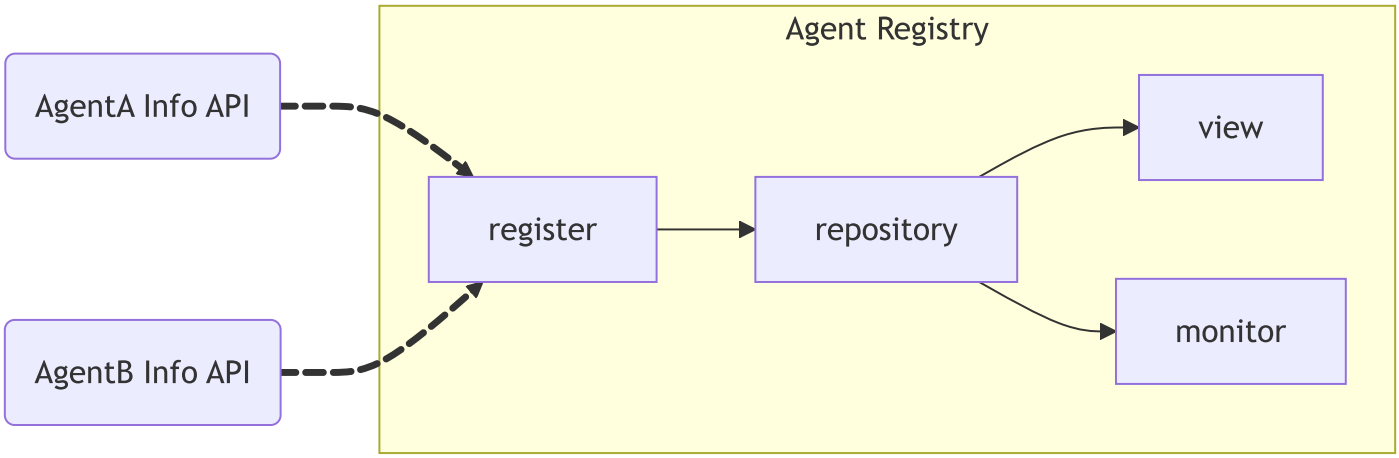
In order to implement this mechanism, the register mechanism needs to know where and how to request the different Agent Info endpoints



For a detailed sequence diagram of this process, see Dynamic Agent Registry (Service Mesh for Agents) Diagram

Agent-Initiated Self Register

Alternatively, the register mechanism can be an API endpoint, where the Agents can register "themselves" into the registry. In order to implement this mechanism, the register mechanism needs to provide an endpoint where the Agents can reach to provide their Agent Information.



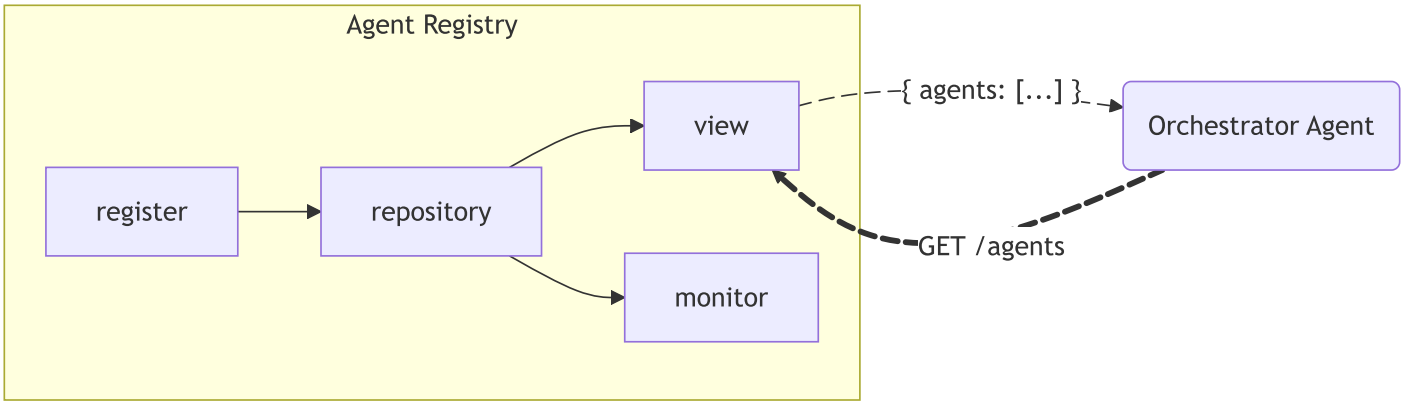
Repository Component (Storage)

We need a storage mechanism that houses all the available Agent Information.

The actual storage can be flexible, but it's important to consider that the Agent Information needs to be easily listed, and it could potentially be queriable.

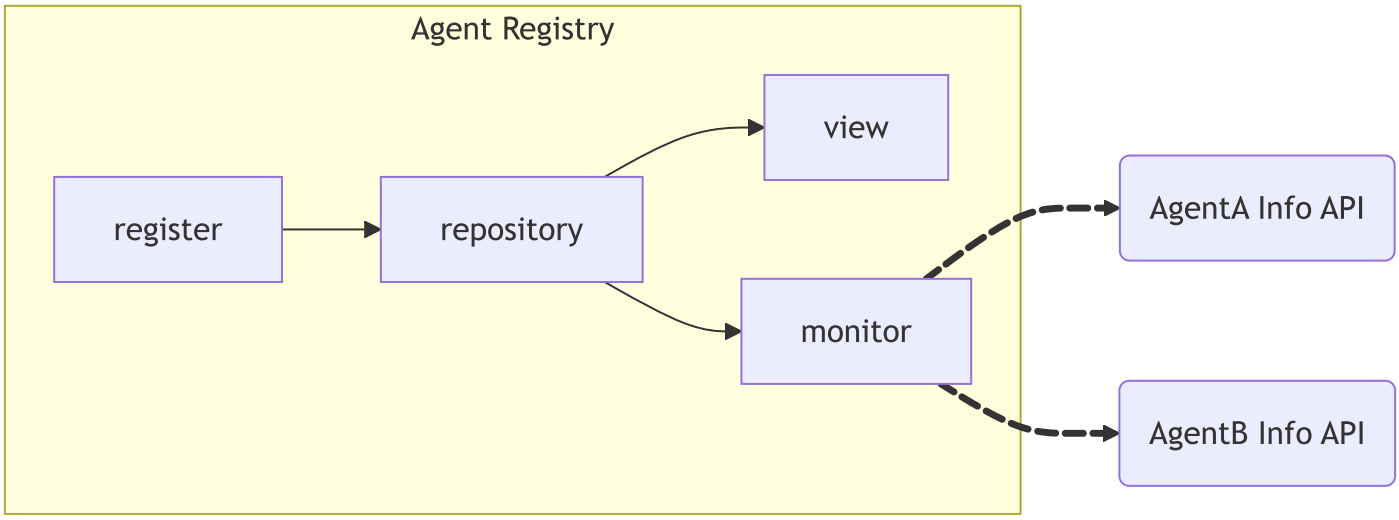
View Component

The View Component lists all the available agents or in more advanced cases, provides filtering capabilities



Monitor Component

In an enterprise scenario, it's important to ensure that the agents registered in the Agent Register are up and running, and that the Agent Information are up to date. So often it is necessary to have a monitoring mechanism to regularly call the registered agents for uptime and information



Data: Agent Information

The information stored in the Agent Repository (so far referred in this documentation as "Agent Information") is used for two purposes:

- For the maintainers/developers of the system (humans) to see the agents are available
- For the Agents to determine what's the best agent(s) to interact with to complete the task

For Human Consumption

In order to form a more robust Agent Catalog more information might be needed for categorization or classification (such as `tags` or `owners` in the metadata of the ACP Agent Detail Spec).

For Agent Consumption

At a very minimum, two pieces of information are needed to describe an agent:

- `name` : the name of the agent that should convey an idea of the purpose of the agent. (For example, "TravelBookingAgent" is better than "Bob")
- `description` : A detailed description of what the bot can do.

Additional information such as the expected input and output and communication mechanism are also helpful for the consuming agent to achieve effective communication

Furthermore, since the Agent Information is used by Agents to communicate with other Agents, additional information is needed to inform the consuming (i.e. Orchestorator) Agent **how** to communicate with the registered Agent. Such as `url` and `authentication` properties in the A2A Agent Card.

For more information on current standarizations of Agent Information from the community:

- A2A Agent Card Specification
- ACP Agent Detail Specification

Evaluation of Registering Agent

In Enterprise Multi-Agent Scenarios, there needs to be processes and policies in-place to ensure that by adding agents to the registry will not degrade the system's behavior.

During the registration process, it's recommended to contain a validation step to evaluate the validity of the agent being registered.

See the Dynamic Agent Registry (Service Mesh for Agents) Diagram for a detailed walkthrough.

Additional Examples and Resources

- Google's Public A2A Agent Registry
- ACP Documentation on Agent Registry

Agents Communication

Last updated: 2025-06-30

This chapter introduces two foundational paradigms for agent interaction: request-based and message-driven communication. These models shape how agents coordinate, scale, and recover in distributed systems.

- Request-Based Communication: Agents communicate by sending direct requests to one another—through synchronous or asynchronous interactions— offering predictability and simplicity, making it well-suited for tightly coupled or latency-sensitive scenarios.
- Message-Driven Communication: Agents communicate asynchronously via a broker or event bus, exchanging commands, events, or responses. This promotes loose coupling, scalability, and resilience—especially in distributed or dynamic environments.

Many systems adopt **hybrid models**, combining request-based communication with asynchronous messaging to balance control, flexibility, and fault tolerance (e.g. agents emitting events to be consumed by external systems).

This chapter outlines the strengths, trade-offs, and design considerations of each approach to help you align communication strategies with your agents' roles and system goals.

Request-Based Communication in Multi-Agent Systems

Last updated: 2025-06-04

In a request-based model, all interactions—between clients, orchestrators, and expert agents—are initiated by explicit requests. A central orchestrator receives the client's request, plans the workflow, and delegates tasks to specialized agents. Each agent processes its task independently and returns results to the orchestrator, which compiles and returns the final output.

Non-Streaming vs Streaming Interactions

Most client-to-agent communication falls into two categories:

- **Non-Streaming (Single-Response):** The client sends a request to the orchestrator that initiates the multi-agent workflow, and waits for a single, complete response—well suited for short-lived or atomic tasks where only the final result matters.
    - Advantages: Simplicity, straightforward error handling and easier observability.
    - Limitations: Not ideal for long-running or incremental workloads; client is blocked until task completes.
- **Streaming (Incremental Response):** Results or updates are sent incrementally during processing (e.g., via server-sent events, gRPC streaming, or websockets). This enables real-time progress feedback or delivery of large outputs as they are generated, common in LLM-driven use cases
    - Advantages: Immediate progress visibility, lower perceived latency for users, supports scalable delivery to multiple consumers.
    - Limitations: More complex protocol handling, error management, and observability; potential network compatibility issues.

Streaming recommendation

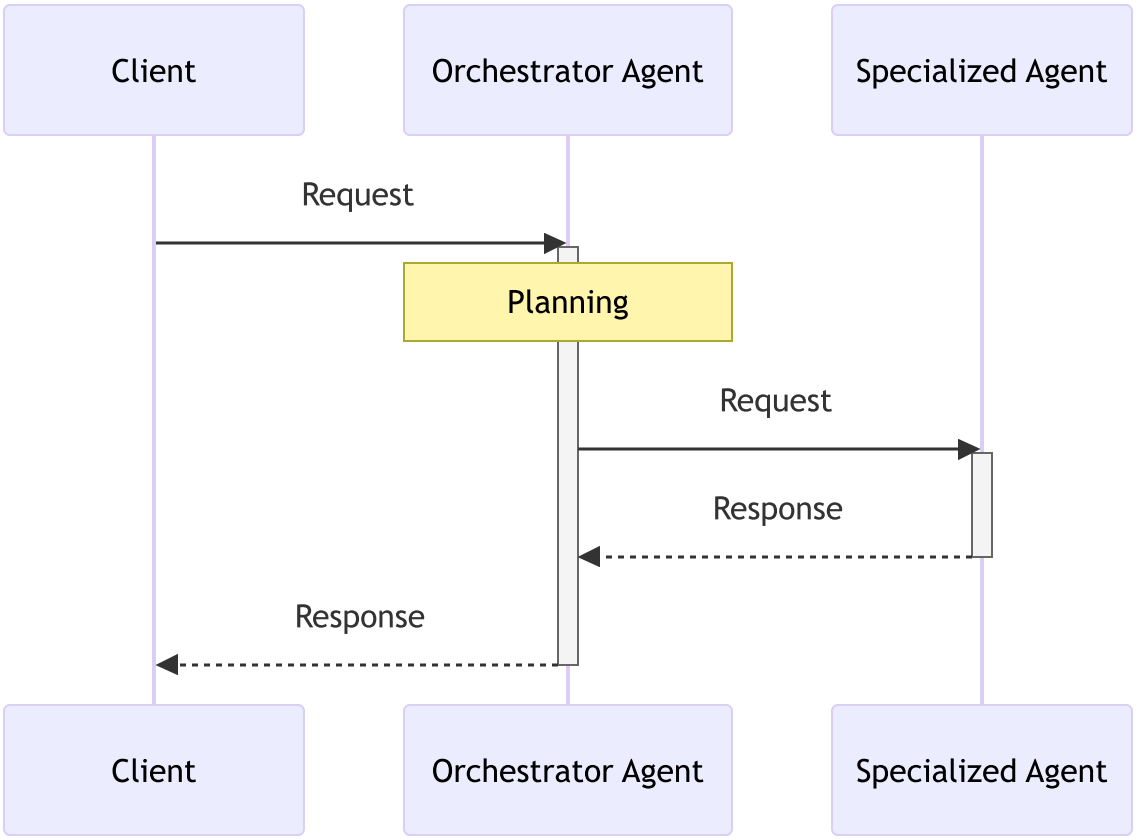
In practice, for multi-agent architectures, it's most effective to only stream responses from the orchestrator agent to the end client—not between the orchestrator and internal expert agents due the following reasons:

- **Workflow Control**: The orchestrator is responsible for the workflow's consistency and correctness. If expert agents stream content directly, the orchestrator cannot guarantee ordering, error handling, or workflow structure.
- **Error Handling**: Streaming across multiple hops makes error recovery more complex—if an expert agent fails or returns incomplete data, the orchestrator must reconcile fragmented streams, resulting in fragile and error-prone logic.
- **Increased Complexity**: Maintaining streaming protocols, session state, and message ordering across all agent-to-agent boundaries increases architectural complexity and operational risk.
- **Observability & Traceability**: When multiple agents stream data independently, it becomes challenging to trace events, maintain version control, and audit workflows in a predictable sequence.
- **Value vs. Effort**: Direct streaming between agents rarely delivers end-user value that justifies the development and maintenance overhead. The orchestrator is the only agent with enough context to assemble, filter, and sequence partial results before presenting them to users.

Communication Patterns

1. Synchronous Request-Reply

Non-streaming



Key characteristics

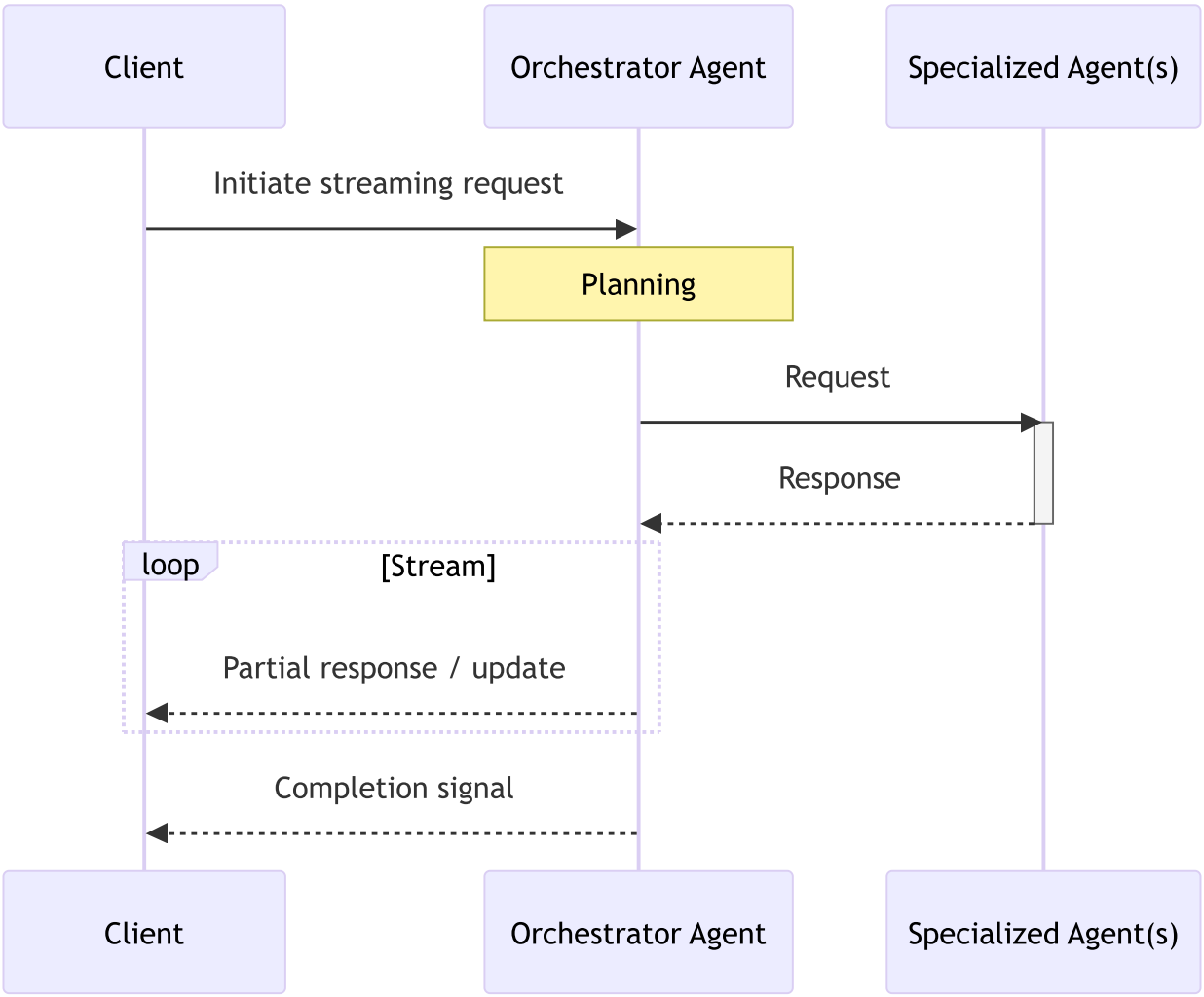
- **Direct Feedback**: Results or errors are delivered immediately in the same session.
- **Straightforward observability**: Tracing and debugging form linear, easy-to-follow flows.
- **Blocking operation**: The client waits for a response, pausing its workflow until the agent response completes.

Tradeoffs

- **Temporal coupling**: Both client and agents must be available and responsive at the same time.
- **Scalability**: Each request consumes resources until completion, constraining throughput under high concurrency.
- **Latency Sensitivity**: Downstream slowness or outages have immediate upstream impact.

- Availability Risk: Agent outages or delays directly affect the client.
- **Challenges with long-running tasks**: Increased risk of timing out and overloading system resources.

Server-Streaming



Common streaming connection patterns:

- **Server-Sent Events (SSE)**: The orchestrator agent streams incremental updates to the client over a single HTTP connection using the `text/event-stream` format—well-suited for browsers and lightweight consumers.
- **HTTP Chunked Responses**: The server delivers result fragments as they become available via HTTP chunked transfer encoding, keeping the connection open for the duration of the task.

Key Characteristics

- **Non-blocking workflow**: The client submits a request to initiate a potentially long-running process and receives prompt acknowledgment (containing a unique task ID). The workflow executes independently, freeing the client from waiting for completion.

  > While the overall pattern is asynchronous, the initial request is typically a synchronous HTTP exchange to obtain the reference/task ID.

- **Deferred response via polling**: The client periodically polls a status endpoint (using the task ID) to check for task progress or retrieval of the final result.
- **Temporal decoupling**: The client and orchestrator do not need to maintain a persistent connection or block on each other—the client can reconnect and resume polling as needed.

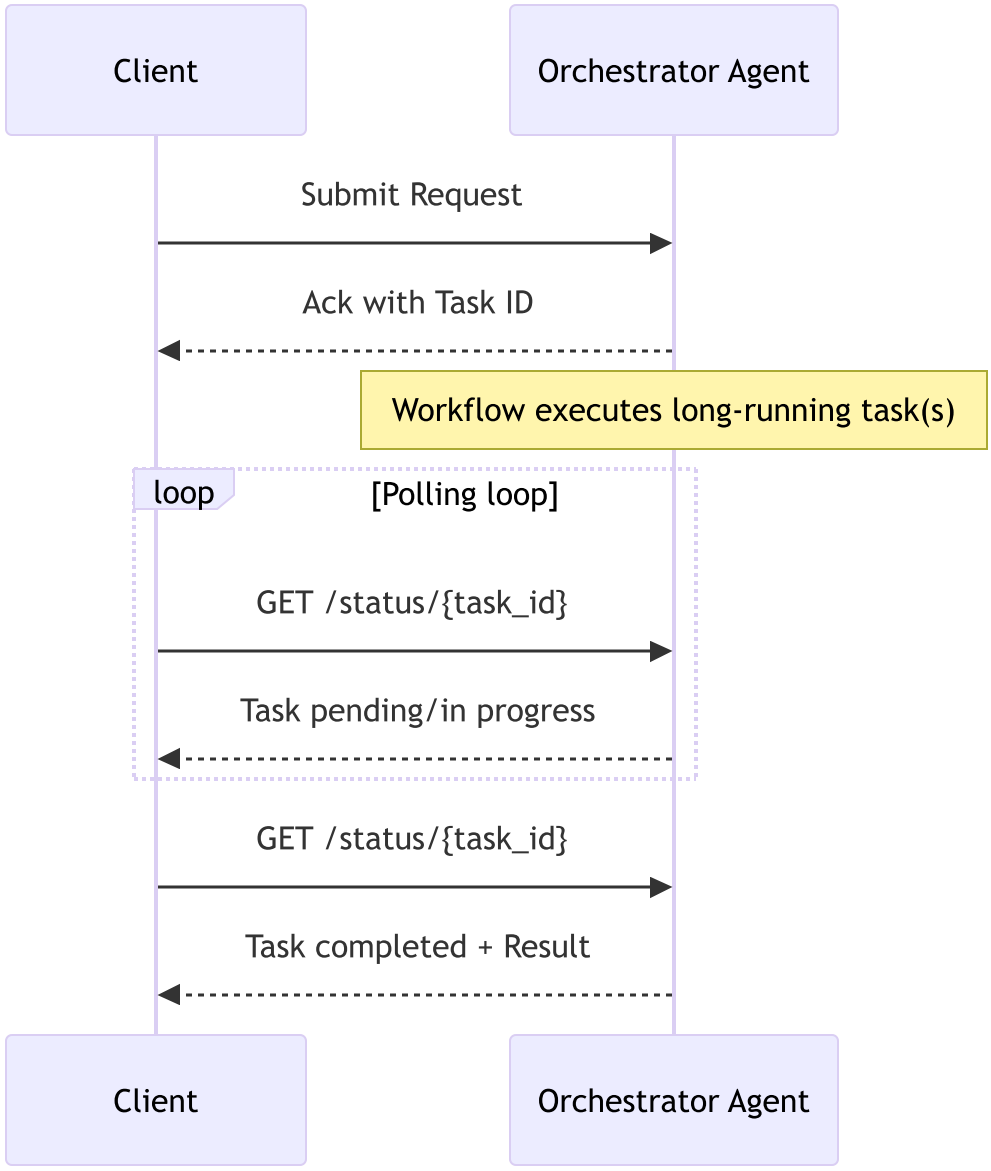
Tradeoffs

- **Client/Transport Support**: Not all clients (e.g., legacy proxies, web clients) handle open or long-lived connections or all protocols (gRPC, SSE, etc.).
- **Error Handling Complexity**: Client must handle mid-stream errors, partial or interrupted streams, and implement reconnection logic to continue receiving updates without data loss or duplication.
- **Potential Networking Restrictions**: Firewalls, proxies, or load balancers may block or prematurely terminate long-lived HTTP connections; mobile or low-bandwidth networks can disrupt streams; TLS termination proxies might interfere with HTTPS streaming.

---

2. Asynchronous Request-Reply

Non-streaming



Key Characteristics

- **Non-blocking Interaction**: The client sends a request to trigger a long-running process and receives an immediate acknowledgment, usually containing a reference ID. The actual result is delivered later, allowing the client to continue without waiting.

> While the overall pattern is asynchronous, the initial request often involves a brief synchronous exchange to initiate processing and retrieve a task reference.

- **Deferred Responses**: Results are delivered through asynchronous polling endpoint channel.
- **Temporal Decoupling**: The client and orchestrator do not need to block while waiting for each other. However, polling requires the client to remain available or reconnect periodically.

Tradeoffs

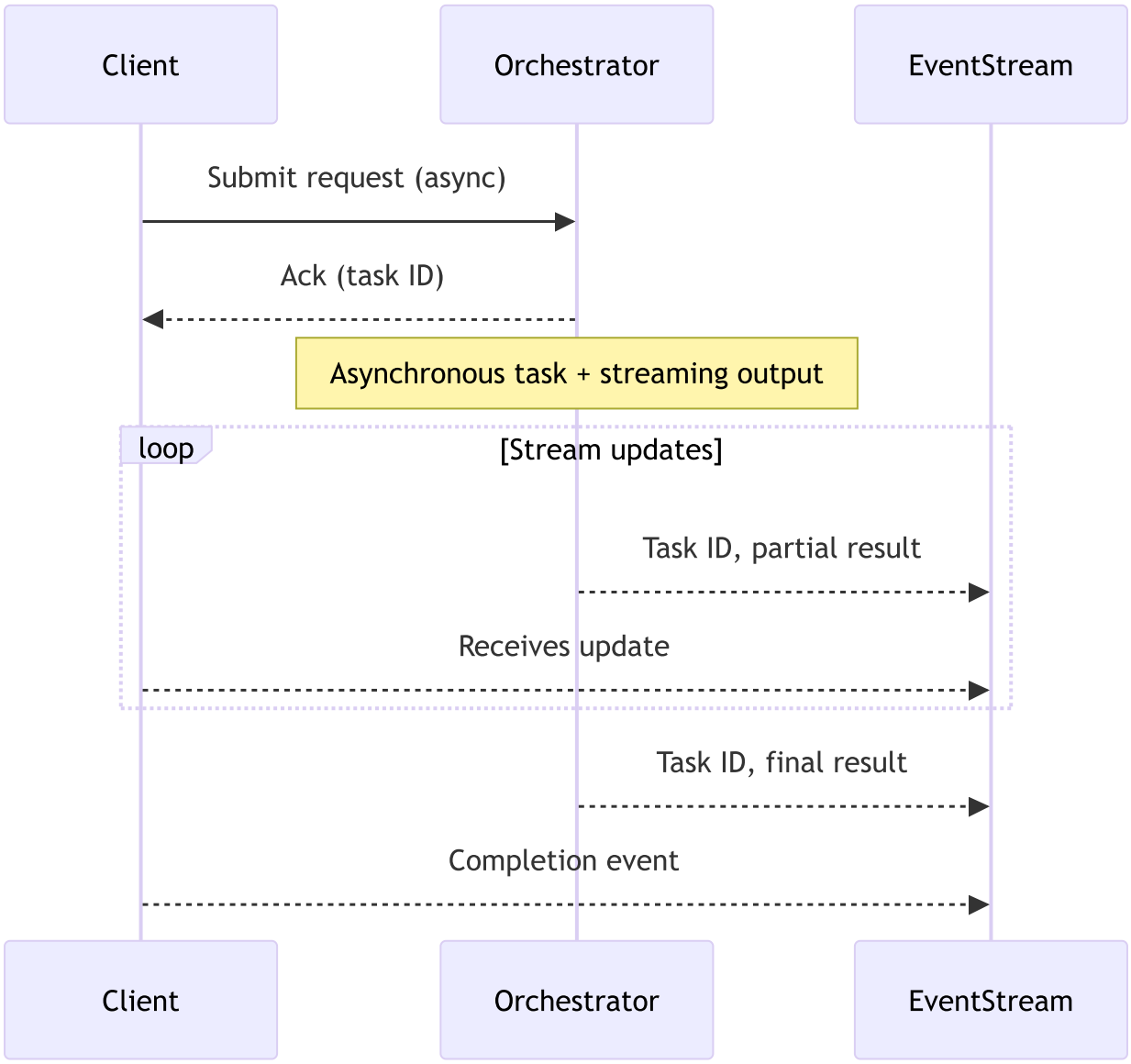
- **Increased Complexity**: Requires mechanisms for reliable delivery, idempotency (the ability of a given operation to always produce the same result), correlation IDs, and robust failure handling on both client and coordinator sides.
- **Management Overhead**: Polling introduces additional load on the orchestrator and may result in inefficient resource usage if polling intervals aren't well-tuned.
- **Eventual Consistency**: State about long-running tasks may briefly diverge between client expectations and orchestrator status, especially under failures or network partitions.
- **Observability Challenges**: To accurately view workflow progress, distributed tracing and proper correlation of request/response cycles (e.g., using trace or context IDs) is essential.
- **Duplicate/Out-of-Order Processing**: Clients may issue redundant requests or receive delayed/out-of-order responses, requiring deduplication and idempotency strategies.
- **Delayed Feedback**: Clients must rely on repeated polling for updates rather than receiving instantaneous notifications—potentially increasing perceived latency.

> Message-driven alternatives (such as webhooks, message queues and brokers) will be covered in Message-driven Communication.

Server-Streaming

Similar to non-streaming approach, the client sends a request to trigger a long-running process and receives an immediate acknowledgment, usually containing a reference ID. The client then opens a special streaming connection referencing the Task ID to receive updates as the task progresses.

Streamed messages can include a TaskID or similar identifier to help the client associate updates with the correct ongoing request.



Key Characteristics

- **Truly Decoupled Streaming**: Neither client nor agent is blocked; updates are pushed as they become available using HTTP-based streaming mechanisms such as Server-Sent Events (SSE) or chunked transfer encoding.
- **High Scalability**: Well-suited for distributed, high-throughput systems—whether using lightweight server-push methods like SSE or message brokers.

Tradeoffs

- **Higher Complexity**: Requires robust handling for message ordering, deduplication, replay, and idempotency across distributed systems.
- **Potentially "Lost" Events**: Ensuring reliable delivery (e.g., at-least-once, at-most-once, exactly-once) demands careful design of brokers and consumers, especially with transient connections like SSE.

- **Stream-to-Task Correlation**: All result messages must include correlation metadata such as `TaskID`, `RequestID`, or similar to allow downstream processing to track and associate updates correctly.

Summary Table

Aspect	Synchronous (Non-Streaming)	Synchronous Streaming	Asynchronous (Non-Streaming)	Asynch Strea
Responsiveness	Immediate, single response	Immediate start, incremental results	Deferred response	Deferred incremer results
Scalability	Low (client blocking)	Low–Medium [1]	High[2]	Very High
Complexity	Low	Medium (connection mgmt, reconnections)	High (queues, retries, correlation)	Very High (ordering deduplica idempote
Best Fit	Short tasks, APIs, low latency	Progress bars, live updates, partials	Batch jobs, background tasks, events	Long-run with feed event wo
Failure Handling	Simple	Medium (stream errors, reconnects)	Advanced (timeouts, retries, idempotency)	Complex replay, cc IDs, lost r

[1] Partial results are flushed and transferred as they are produced. This means the server can free memory sooner, and the client can process each chunk without needing to buffer everything until the end.

[2] Enables time decoupling between the client and orchestrator. Leverages background jobs, improving scalability, fault tolerance, and supporting offline processing.

Recommendations

- **Start Simple**: Start with the simplest communication pattern that effectively serves your use case—typically synchronous or asynchronous request-reply without streaming. These approaches are easier to implement, debug, and observe, making them ideal during early development or when validating your core business logic.
- **Adopt Streaming Incrementally**: Introduce streaming only when clearly justified—such as when delivering incremental results, providing real-time feedback, or handling large outputs that would otherwise block or degrade performance. Avoid starting with streaming unless it solves a proven bottleneck.
- **Use Standards-Based Protocols**: Favor open and emerging protocols to future-proof your architecture and benefit from built-in security, governance, and interoperability.

  > Examples include Agent-to-Agent Protocol (A2A), Agent Network Protocol (ANP), and Agent Communication Protocol (ACP), which are particularly suited for structured request-based communication between agents.

- **Prioritize observability for async and streaming**: Asynchronous and streaming systems introduce additional operational complexity. Invest early in tracing, correlation IDs, error handling, and monitoring tools to maintain visibility and quickly resolve issues.
- **Iterate based on real feedback**: Let actual production signals and user experience guide your evolution from simple to more complex communication patterns. This iterative, feedback-driven approach helps avoid premature optimization and aligns well with agile development principles.

References

- Asynchronous Request-Reply pattern
- A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)

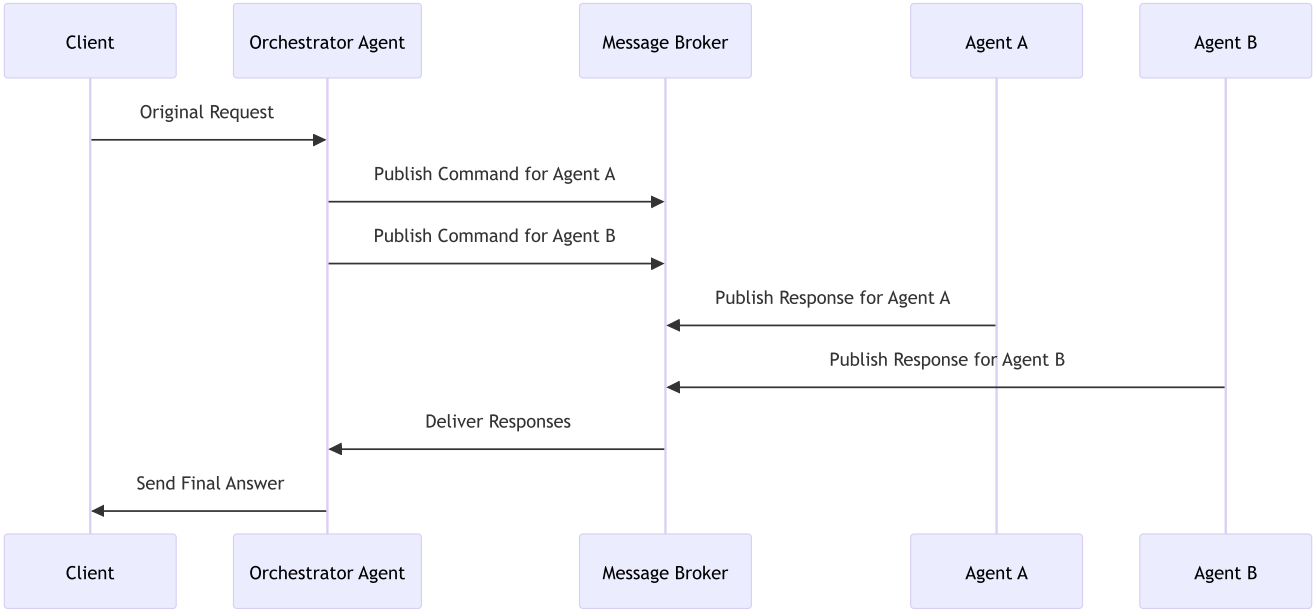
Message-Driven Communication in Multi-Agent Systems

Last updated: 2025-06-30

In a message-driven model, agents interact asynchronously by exchanging discrete messages—such as commands, events, or responses—via a broker or event bus. This architecture promotes loose coupling, scalability, and resilience, especially in distributed environments.

Orchestration

1. The client initiates the multi-agent workflow by sending a request—or publishing a message to a broker or event hub—to the orchestrator.
2. The orchestrator decomposes the request into subtasks and issues *command messages* to appropriate specialized agents.
3. Each specialized agent processes its command independently and publishes a response message, typically to a reply-to queue or topic indicated by the orchestrator. Responses include results or status, and often carry a correlation ID to link them to the original request. This asynchronous flow allows the orchestrator to aggregate and correlate responses from multiple agents efficiently.
4. The orchestrator collects specialized agent responses, synthesizes the final output, and sends it to the client.



Key Characteristics

- **Asynchronous & Decoupled**: Client and orchestrator operate independently in time; message delivery is decoupled from direct request-response lifecycles.
- **Scalability**: Message queues/brokers can act as a buffer to absorb sudden traffic spikes by temporarily holding messages until the consumer (orchestrator) is ready to process them, balance load, and support horizontal scaling across agents.
- **Resiliency**: Built-in retries, dead-letter queues, and durable message storage provide resilience against transient network issues and agent failures, ensuring reliable delivery and fault tolerance.

Tradeoffs

- **Broker management overhead**: While brokers help smooth workload spikes, scaling to very high message volumes demands careful planning, provisioning, and ongoing management to maintain system performance and reliability.

- **Fault Tolerance**: Partial results and error handling require protocol discipline. To handle this reliably, the system must:

    - Define clear message contracts that specify how success, failure, and retries are communicated.
    - Track correlation IDs and message state to know which tasks have completed and which have failed or are pending.
    - Implement idempotent handlers, so retries don't cause duplicate processing or side effects.
    - Use dead-letter queues or similar mechanisms to capture and inspect failed messages for debugging or reprocessing.
    - Apply consistent error-handling strategies, such as retries, compensation actions, or fallback logic, to avoid data inconsistency.

- **Latency**: While throughput can be high under ideal conditions, the asynchronous nature of message-driven systems makes it harder to guarantee low latency under certain conditions, such as:

    - Queue delays: Messages may sit in a queue waiting to be picked up if agents are under-provisioned, busy, or scaling up. Even if the message is sent instantly, it may not be processed right away.
    - Batch processing: Some systems intentionally process messages in batches to improve throughput or efficiency, but this adds wait time as messages are held until the batch fills or a timeout is reached.

    - Retries and backoff: Failed messages may be retried with exponential backoff or delay intervals, extending overall processing time for those messages.
    - Broker throughput limits: Brokers can become bottlenecks when handling large volumes, leading to increased end-to-end latency.
    - Async dependencies: When multiple services rely on one another through messaging, delays can cascade across the system, compounding latency even if individual services are fast.

- **Workflow Management Complexity**: Message-driven systems introduce architectural complexity in several key areas such as:

    - **Message correlation**: Given a workflow is broken into asynchronous steps, the orchestrator must track which responses belong to which specialized agent using correlation IDs. Managing these IDs consistently across services adds cognitive and implementation overhead.
    - **Idempotency**: To prevent unintended side effects from duplicate message delivery (which can happen in at-least-once delivery models), agents must ensure that processing the same message multiple times leads to the same result. This often requires state checks, deduplication logic, or idempotent storage operations.
    - **Reprocessing and retries**: Supporting reprocessing—whether for recovery, auditing, or data correction—requires careful handling to avoid data corruption or logic errors. Agents must distinguish between new and replayed messages and handle them appropriately.

- **Observability**: Achieving end-to-end visibility of logs, traces and metrics requires careful discipline across all agents:

    - **Correlation IDs** must be consistently generated and propagated with each message so that actions across distributed services can be tied back to the same workflow or request.
    - **Trace context propagation** is essential for distributed tracing tools (compliant to standardized formats such as OpenTelemetry) to stitch together spans from multiple services into a coherent timeline.
    - **Observability tooling** must be integrated into all agents and message handlers, including retries, dead-letter handling, and asynchronous workflows.

- **Messages ordering**: When the orchestrator dispatches tasks to multiple specialized agents, their responses may arrive in any order. Relying on arrival time to infer sequencing (e.g., "this agent's message came last") is unreliable and fragile. To ensure correctness when order matters, the system should rely on explicit sequencing strategies, such as:

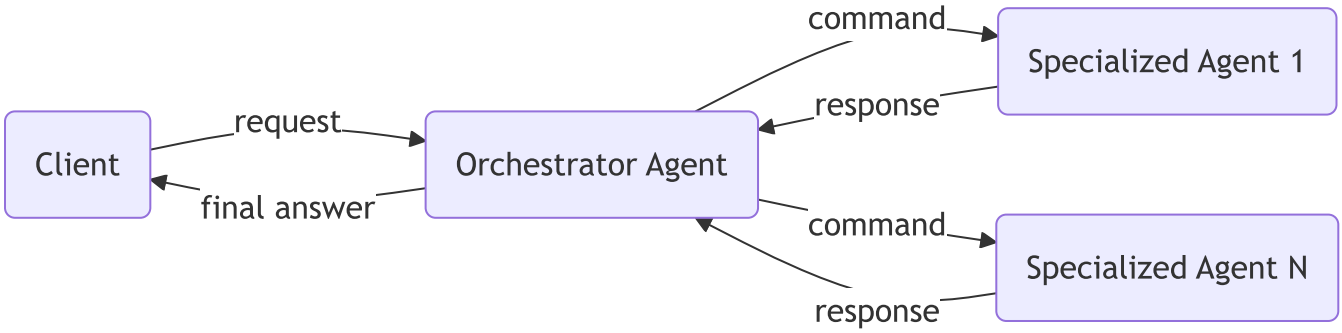
    - **Task dependency chains** (e.g. directed graph), where agents are triggered only after prior steps complete.

- **Sequence numbers or stage metadata** attached to messages for reordering incoming messages.
- **Per-agent or per-key partitioning** to preserve local order without blocking the entire workflow.

Communication Patterns

Parallel Fan-Out

The orchestrator fans out commands to multiple agents, then waits for all responses (within a timeout) before synthesizing the result.



Key Characteristics

- **High concurrency and parallelism**: Enables simultaneous execution of independent agent tasks, maximizing throughput for suitable workloads.
- **Aggregated response handling**: The orchestrator collects and synthesizes results from multiple agents, supporting flexible aggregation logic (e.g., voting, merging, or best-effort completion).
- **Loose coupling between agents**: Agents do not need to know about each other or coordinate directly, reducing interdependencies and simplifying agent logic.
- **Flexible timeout and fallback strategies**: The orchestrator can define custom policies for how long to wait and what to do if some agents do not respond in time.

Tradeoffs

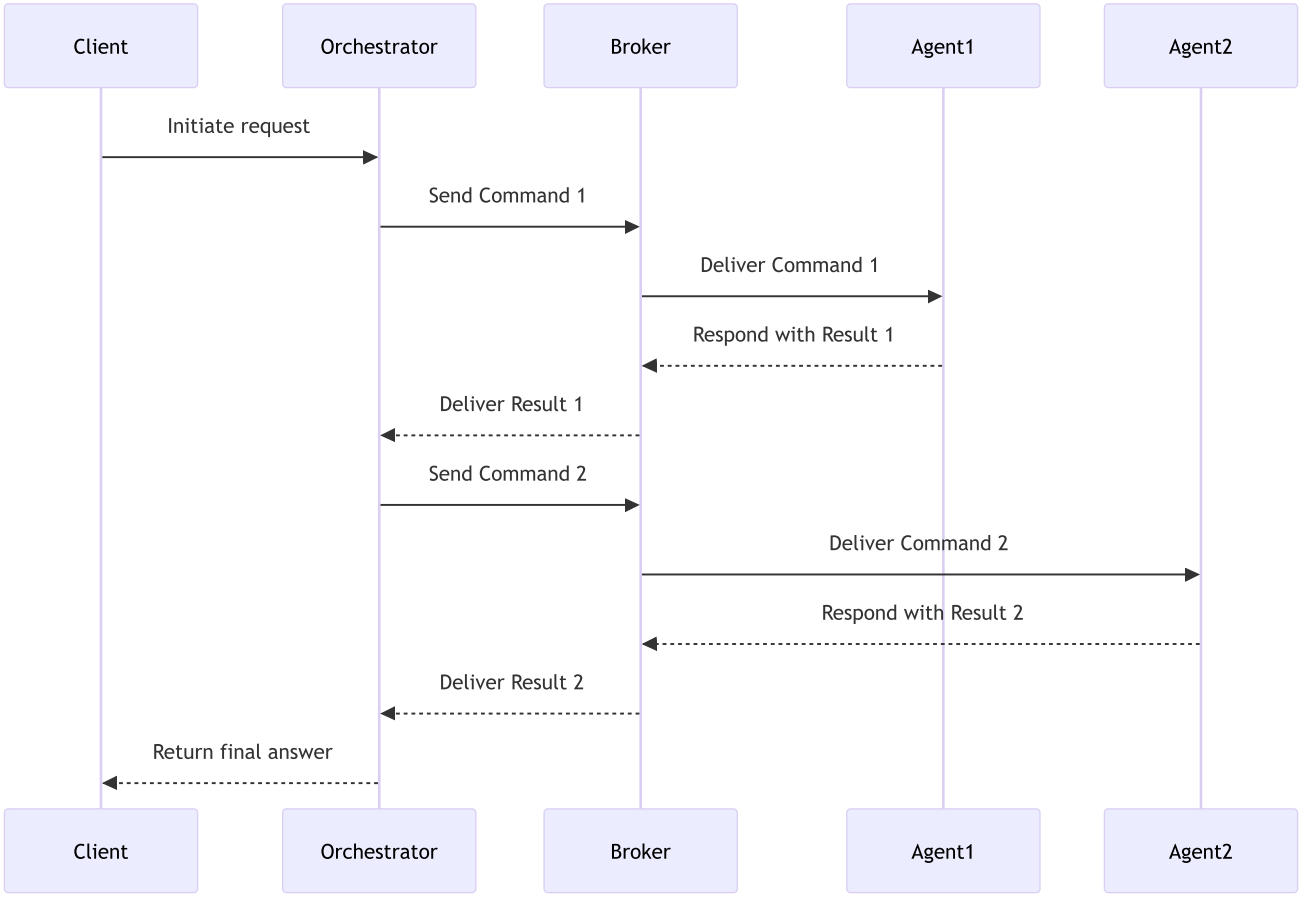
- **Risk of partial or inconsistent results**: If some agents fail or are slow, the orchestrator may need to proceed with incomplete data, which can affect the quality or reliability of the final output.

- **Increased orchestrator complexity**: Aggregating, correlating, and synthesizing multiple asynchronous responses requires careful design, especially as the number of agents grows.
- **Potential for resource contention and downstream bottlenecks**: Large-scale fan-out can stress shared infrastructure or external dependencies, requiring proactive scaling and monitoring.
- **No guarantee of response order**: The orchestrator must be robust to out-of-order arrivals and design aggregation logic accordingly.

Typical use cases: data aggregation, consensus, distributed analysis.

Chained Task Sequencing

The orchestrator performs a sequence of commands, where the output of each step is used as input for the next. This enables tasks that require intermediate context, validation, or enrichment before the next operation.



Key Characteristics

- **Stepwise processing with data dependencies**: Each agent's output can be validated, transformed, or enriched before passing to the next step, supporting complex workflows.
- **Deterministic execution order**: Tasks are performed in a strict sequence, ensuring that each step receives the required context from previous results.
- **Fine-grained error handling and validation**: The orchestrator can inspect and handle errors or validation failures at each stage before proceeding.
- **Supports conditional branching**: The orchestrator can make decisions at each step, enabling dynamic workflows based on intermediate results.

Tradeoffs

- **Increased end-to-end latency**: Sequential execution means each step must wait for the previous one to complete, which can increase total processing time compared to parallel patterns.
- **Potential bottlenecks at slowest step**: The slowest agent or operation determines the overall workflow speed, as each subsequent step must wait for the previous one to finish. This means that even if most agents are fast, a single slow or overloaded agent can delay the entire sequence. Bottlenecks can arise from resource constraints, external dependencies, or complex processing logic in any step. To mitigate this, monitor step-level performance, optimize the slowest stages, and consider parallelizing independent sub-steps where possible. Additionally, design for timeouts, retries, or fallback paths to prevent indefinite blocking if a step becomes unresponsive.
- **Orchestrator complexity**: Managing state, error handling, and conditional logic across multiple steps increases orchestrator implementation complexity.
- **Reduced fault isolation**: Failures in early steps can block the entire workflow, requiring robust error handling and compensation strategies.

Typical use cases: multi-stage data processing, validation pipelines, enrichment chains, workflows with strict task dependencies.

Recommendations

Message Design

- **Use clear message contracts**: Define explicit schemas for commands, events, and responses to ensure consistency across agents.
- **Include correlation IDs**: Always attach unique identifiers to track message flows across the distributed system.
- **Design for idempotency**: Ensure that processing the same message multiple times produces the same result.
- **Implement structured error reporting**: Use consistent error formats that include error codes, descriptions, and recovery suggestions.

Broker and Infrastructure

- **Choose the right broker**: Select message brokers based on your throughput, latency, and durability requirements (e.g. Choose between Azure messaging services).
- **Plan for scaling**: Design your broker topology to handle peak loads and implement auto-scaling where possible.
- **Monitor broker health**: Track queue depths, processing rates, and error rates to identify bottlenecks early.
- **Implement dead letter queues**: Configure mechanisms to capture and analyze failed messages for debugging and recovery.

Agent Design

- **Keep agents focused**: Design each agent around a single, well-defined responsibility to maintain loose coupling.
- **Implement circuit breakers**: Protect against cascading failures by implementing circuit breaker patterns in agent communication.
- **Use asynchronous processing**: Design agents to handle messages asynchronously to improve system responsiveness.
- **Plan for graceful degradation**: Implement fallback mechanisms when dependent agents are unavailable.

Orchestration Best Practices

- **Minimize orchestrator complexity**: Keep orchestration logic simple and push complex business logic into specialized agents.
- **Implement timeout strategies**: Define appropriate timeout values for different types of operations and implement fallback behaviors.
- **Use event sourcing when appropriate**: For complex workflows, consider event sourcing to maintain an audit trail and enable replay capabilities.
- **Design for observability**: Implement comprehensive logging, metrics, and tracing to monitor system health and troubleshoot issues.

References

- Choose between Azure messaging services
- A Distributed State of Mind: Event-Driven Multi-Agent Systems

---

Memory

Last updated: 2025-05-18

Memory is a foundational aspect of multi-agent systems, shaping how agents understand context, make decisions, and collaborate effectively. This chapter introduces two core memory types in multi-agent systems:

- **Short-term Memory (STM):** Enables agents to maintain recent context within an active session (also known as conversation history), supporting coherent interaction and task coordination across agents.
- **Long-term Memory (LTM):** Provides persistence of information across sessions, allowing agents to recall knowledge, preferences, and outcomes over time to provide personalized experiences.

Designing STM and LTM in multi-agent systems brings unique challenges around synchronization, ownership, privacy, and data consistency. The following sections outline key patterns and trade-offs for integrating effective memory into your architecture.

Short-term Memory

Last updated: 2025-05-26

When building the STM (Short-Term Memory) layer for a multi-agent system, the storage engine choice is critical. STM typically serves to persist conversation history, contextual state, and intermediary data that agents use to manage context and continuity during workflows.

This topic covers:

Key Requirements

- **Fast Read/Write Performance:** Conversations and agent states change quickly, requiring low-latency operations.
- **Flexible Schema:** Messages and state objects may evolve over time, especially if you enrich them with metadata, add new message types, or change the workflow.
- **Scalability:** As concurrent conversations and active sessions grow, the storage must scale horizontally.
- **Simple Data Access Patterns:** Retrieval by conversation/session ID, time range, or participant/user.

Given those requirements, document-oriented NoSQL databases are preferable. They typically offer:

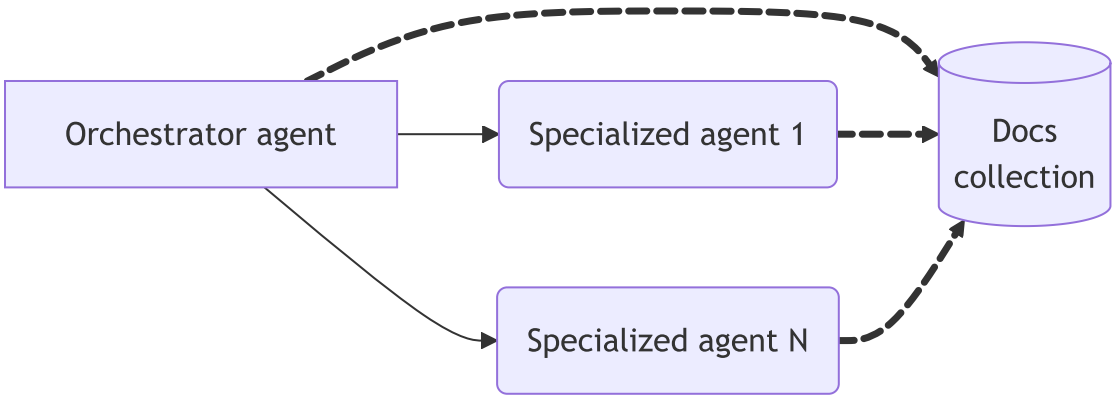
- **Flexible Schema:** Easily accommodates changing or varied agent message formats without migrations.
- **Nested/Hierarchical Data:** Supports complex, nested data structures ideal for conversation history.
- **Horizontal Scalability:** Designed for high throughput, automatically scales as data and sessions increase, and supports partitioning for session cleanup.
- **Efficient Retrieval:** Optimized for access patterns (by keys, IDs, timestamps) common in chat and STM use cases.

As a reference, see How Microsoft Copilot scales to millions of users with Azure Cosmos DB

Design approaches

1. Shared Memory

All participating agents read and write session context to a centralized documents collection, which acts as the single source of truth.



Key Characteristics

- **Simplicity**: Easy to implement and operate.
- **Unified Traceability**: A complete interaction content in one place, ideal for debugging and auditing.
- **Consistent Context**: All agents can access the latest, synchronized session data.

Data Modeling

Consider designing the documents to capture:

- **Session ID:** Unique ID that groups all messages within the same conversation.
- **Message ID:** Unique ID for each individual message.
- **User ID**: Tracks the end-user (if applicable).
- **Source:** Identifies which agent generated the message.

- **Message:** Includes messages from different authors (such as users, assistants, tools, or system/developer) and the actual message payload. If authored by the orchestrator, include planning steps and which agents were invoked.
- **Session Metadata:** Additional data relevant for the session (e.g. communication channel, tags).
- **Timestamp:** When the message was written.

Leverage chat history message objects from agentic frameworks rather than normalizing data when possible. Frameworks such as Semantic Kernel and LangChain already provide rich, extensible and tested chat history objects. Storing the full object as message helps on troubleshooting scenarios (e.g. more detailed information available and the ability to reconstruct the exact conversation history).

Trade-offs

- **Scale Limitations**: One store can become a bottleneck with high throughput.
- **Security and Privacy limitations:** Harder to restrict message visibility between agents.
- **Potential Data Pollution**: An agent may write irrelevant data for the other agents, lowering context quality.

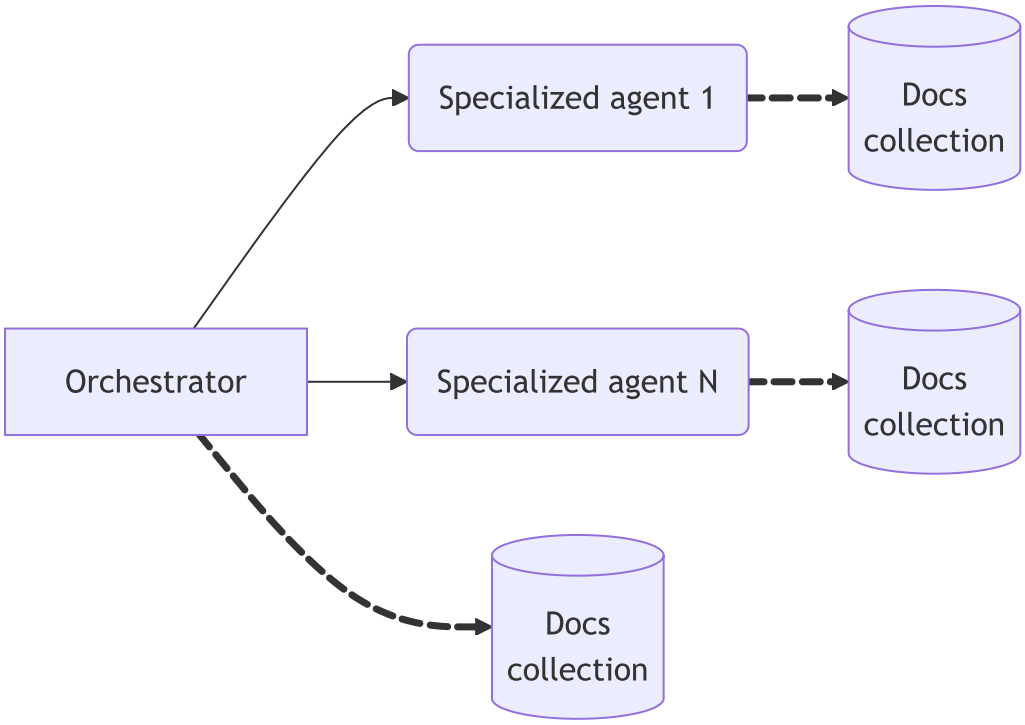
To prevent data pollution, one effective strategy is to isolate each agent's documents within the same collection by incorporating agent-specific data into the message ID (e.g., `<session_id>-<agent_id>`). This structure can then be used to reliably query memory information for individual agents

---

---

2. Distributed Memory

Each participating agent maintains its own documents collection, storing only the context and messages relevant to its domain. Correlation between messages produced by all agents is achieved through a session ID.



Key Characteristics

- **Scalability**: Agents can scale storage independently.
- **Isolation and Privacy:** Avoids cross-agent context leakage; enables agent-specific retention policies.
- **Flexibility:** Agents can optimize memory data modeling for their use case.

Data Modeling

The same recommendations from Shared memory data modeling design applies to distributed memory except the `source` data that is not needed, given the documents for each agent are logically and physically isolated.

Trade-offs

- **Auditing complexity:** Reconstructing the full session context requires aggregating queries from multiple collections.
- **Synchronization:** Requires tight coordination on session/message IDs, or risk losing context.
- **Management overhead:** Extra complexity for maintaining multiple collections.

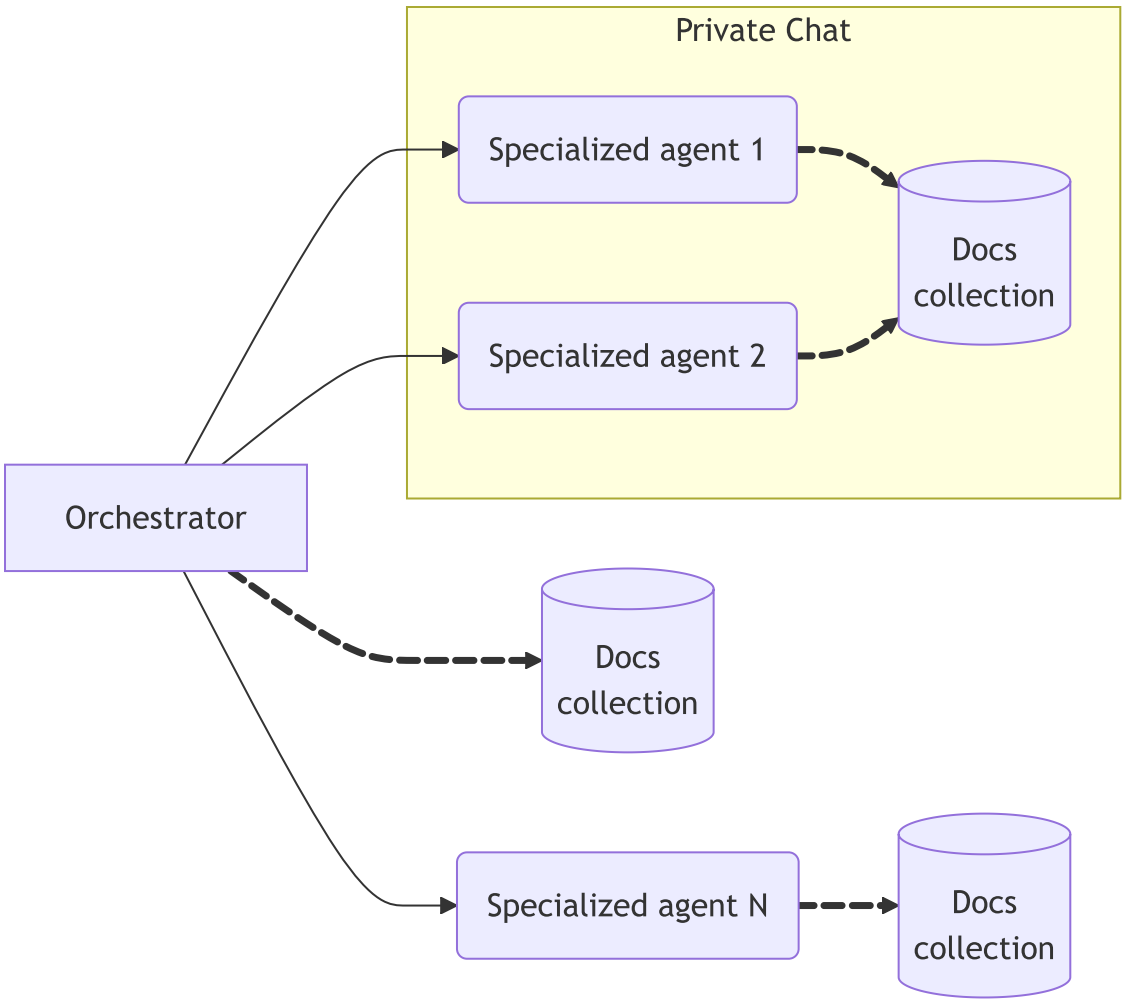
When it's a best fit

- Large-scale, cross-team/department applications.
- When privacy, isolation, or different data retention policies per agent are critical.
- Where agents are implemented in different deployment boundaries (e.g. multi-cloud).

---

3. Hybrid Memory

In this approach, memory is shared only within explicit subgroups of agents. For example, a "private chat" is scoped to a working group of specialized agents collaborating on a task, invisible to others.



Key Characteristics

- **Customizable context sharing**: Only selected agents have access to the memory group.

- **Fine-grained access control:** Supports privacy requirements; only a subset of agents see the context.
- **Natural sharding:** Splits load by team/task force.

Data modeling

The same recommendations from Shared memory data modeling design applies to distributed memory, except the `source` data is not needed for the isolated collections.

Trade-offs

- **Management overhead:** Extra complexity for group membership, permissions, and context boundaries.
- **Fragmentation:** Auditing and reconstructing sessions across overlapping groups is more complex.
- **Consistency challenges:** Context splits and reconciliation logic may be required.

When it's a best fit

- Multi-domain collaboration (e.g., escalation, hand-off, cross-expert panels).
- Compliance-focused, privacy-sensitive workflows.

Recommended Practices

- **Always include traceability fields:** `session_id` , originating agent, user, or system, and timestamps.
- **Store the agent/tool/origin for each message.** Critical for audit and debugging, especially when using shared or group memory.
- **Session Metadata Matters:** Record context like session status, involved users, communication channel, and group membership for each session.
- **Implement unique identifiers for correlation:** When using distributed or selective memory, always correlate on `session_id` .
- **Record orchestrator reasoning:** Orchestrator STM should capture its internal planning, delegation choices, invoked agents/tools, and their responses, not just its final responses.
- **Security and Privacy:** Apply access control on memory access based on agent roles, group memberships, and message sensitivity.

Data Retention

While STM is suitable for hot, fast-access storage, it is recommended to archive conversation data after a defined period—such as when a session expires or after a set number of days. This practice serves several key purposes:

- **Cost Optimization:** Cold storage (e.g. blob storage or data lake) is significantly cheaper than high-performance databases, reducing costs for long-term historical data retention.
- **Analytics & Insights:** Archived conversations can drive analytics, back-office dashboards, and system improvement efforts. The archived data is a rich resource for both business and technical analysis.

Recommendations

1. **Retention Policy:** Define policies for how long data remains in hot storage before being archived or deleted.
2. **ETL/Archiving Job:** Design a job that periodically move expired session documents from STM to a cold storage.
3. **Indexing for Analytics:** Optionally, batch-load archived data into an analytics warehouse for reporting and custom queries.

Observability

Last updated: 2025-10-21

Building observable multi-agent systems requires expanding the traditional pillars of logs, metrics, and traces to address the unique challenges of AI. This involves capturing specialized signals such as agent actions, tool usage, model invocations, and response patterns, to effectively debug, monitor, and optimize agent performance across key areas:

- **Agent Communication**: Tracking inter-agent message flows, coordination patterns, and communication bottlenecks
- **Performance Monitoring**: Measuring response times, resource utilization, and throughput across distributed agents
- **Error Handling**: Detecting failures, cascading errors, and recovery mechanisms in agent workflows
- **Security & Compliance**: Monitoring for unauthorized access, data leaks, and regulatory compliance across agent interactions

Evaluation-Driven Observability

Observability gives us metrics, but **evaluation** is the process of analyzing that data (and performing tests) to determine how well an AI agent is performing and how it can be improved. In other words, once we have traces and metrics, how we can use them to judge the agent and make decisions?

For AI evaluation strategies, see the **Evaluation** section.

For reference:

- Monitoring Generative AI applications
- Observability in Semantic Kernel

Evaluation

Last updated: 2025-10-21

While **observability** provides the instrumentation to collect metrics and traces, evaluation analyzes that data to determine how well agents are performing against defined success criteria and business requirements.

Multi-agent evaluation challenges

Multi-agent systems introduce evaluation complexities like:

- **Path optimization**: Agents may reach correct solutions through inefficient routes, making it difficult to assess optimal performance
- **Error propagation**: Upstream failures can cascade through agent interactions, obscuring the root cause of downstream issues
- **Emergent behavior**: Collective agent interactions produce behaviors that cannot be predicted from individual agent analysis
- **Non-deterministic outputs**: The same input may produce different valid responses, complicating traditional success metrics

Evaluation types and methodologies

Code-based evaluation

This approach uses deterministic and programmatic criteria (like unit testing) to validate system behavior, for example:

Evaluation Type	Purpose	Example
API Response Validation	Ensures endpoint responses match expected results.	A `POST /agent/start` call should return HTTP 200 and include `task_id`.
Output Format Verification	Validates structure and data types of responses.	An agent's reply must include `action`, `result`, and `status` fields.

Evaluation Type	Purpose	Example
Performance Benchmarking	Measures response times and resource usage.	The agent must respond within 500ms under 100 requests per second.
Security Compliance Checks	Verifies adherence to security standards.	Ensure tokens aren't logged and sensitive headers are encrypted.

LLM-as-a-judge evaluation

Unlike code-based evaluation, which uses deterministic rules, this method leverages Large Language Models (LLMs) to assess the semantic, stylistic, and contextual quality of responses—dimensions that are difficult or impossible to capture with hard-coded assertions.

Evaluation Aspect	Purpose	Example
Relevance Scoring	Measures how well the response answers the input.	An agent asked to summarize an article should produce a coherent summary.
Tone and Style Evaluation	Assesses whether the response aligns with expected tone or voice.	A customer support agent should respond politely and professionally.
Bias Detection	Identifies implicit or explicit bias in output.	Flag responses that reinforce gender stereotypes or political bias.
Hallucination Detection	Flags statements that are factually incorrect.	Detect when an agent invents citations or misrepresents known facts.

Evaluation phases

Offline evaluation

Pre-deployment testing using curated datasets and controlled environments:

- Benchmark dataset comparison with expected input/output pairs
- Regression testing against historical performance baselines

- Edge case validation using synthetic scenarios
- Load testing for scalability assessment

Maintain comprehensive test datasets that evolve with your system. Regular updates with new edge cases and failure examples ensure evaluation relevance.

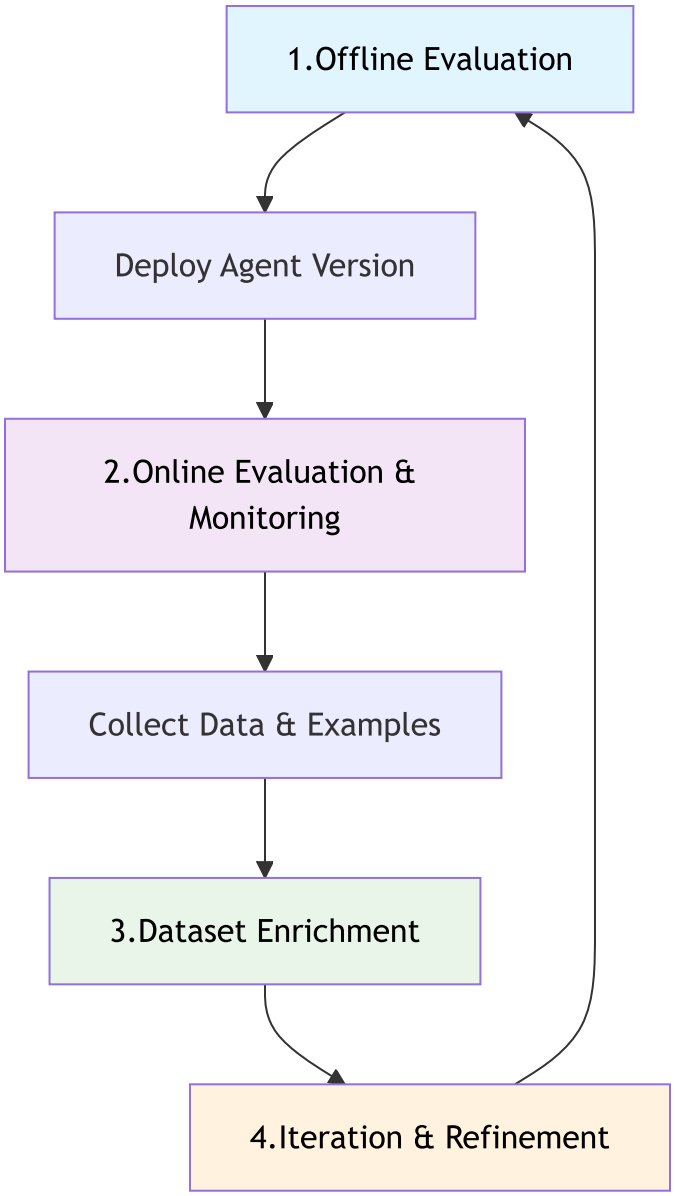
Online evaluation

Production monitoring with real user interactions:

- Real-time performance tracking
- User satisfaction measurement
- Anomaly detection and alerting
- Continuous feedback collection

Evaluation feedback loop

Adopt an iterative loop that blends offline analysis, live monitoring, and data-driven refinement:



Critical evaluation areas

High-risk operations

Systems that modify databases, trigger external actions, or handle sensitive data require enhanced evaluation rigor:

- **Accuracy validation**: Verify correctness of all data modifications
- **Authorization checks**: Ensure proper access control enforcement
- **Audit trails**: Maintain comprehensive logs of all system changes
- **Rollback testing**: Validate recovery mechanisms for failed operations

Security and compliance

- **Access control**: Validate authentication and authorization between agents
- **Data protection**: Test privacy preservation and data handling protocols
- **Adversarial resilience**: Assess system behavior under attack scenarios
- **Regulatory compliance**: Verify adherence to industry standards and regulations

---

> Don't select metrics simply because they're available in your toolbox. Choose evaluation strategies that align with your specific use case and business requirements.

Evaluation strategies by system component

Orchestrator agent evaluation

- **Intent resolution**: Validate correct routing and task decomposition decisions
- **Plan optimization**: Assess efficiency of generated execution plans
- **Response synthesis**: Evaluate quality of final output aggregation
- **Error handling**: Test recovery mechanisms when specialized agents fail

Specialized agents evaluation

- **Task completion**: Measure accuracy and completeness of domain-specific outputs
- **Tool call**: Validate correct function/tool calling and API interactions
- **Response quality**: Assess output relevance, accuracy, and formatting
- **Boundary handling**: Test behavior at capability limits and edge cases

Registry evaluation

- **Discovery accuracy**: Verify correct agent selection for given capabilities
- **Metadata integrity**: Validate agent descriptions and capability mappings
- **Performance tracking**: Monitor agent availability and response times
- **Version management**: Test compatibility across different agent versions

For reference:

- Agent Evaluation in 2025: Complete Guide

- AI Agent Observability and Evaluation
- LLM Evaluation
- Evaluating Multi-Agent Systems

Discuss this page

Tool Call Evaluation

Last updated: 2025-10-21

This document outlines the importance of evaluating tool usage. It defines the characteristics of successful tool invocations, and strategies to assess their correctness and alignment with intended behavior.

Overview

In agent-based architectures, **tool calls** represent the bridge between reasoning and action.

Evaluate this steps is essential for understanding how well the agent translates intent into action, whether it invokes the appropriate tool, and if the execution contributes meaningfully to solving the user's request.

Core Aspects to Evaluate

Tool call evaluation should consider multiple dimensions beyond traditional accuracy metrics:

1. Correctness of Invocation

- Did the agent choose the expected tool/function/module?
- Were the arguments valid, complete, and well-formed?
- Was the call executed without runtime failure?

2. Intent Alignment

- Does the selected tool match the user's request or goal?
- Could another tool have fulfilled the intent more effectively?
- Does the call reflect appropriate use of system capabilities?

3. Context Awareness

- Did the agent account for the conversation history or prior tool outputs?
- Were dependencies between multi-step calls respected?

4. Robustness and Fallibility

- How does the agent behave when a tool is unavailable or fails?
- Does it recover gracefully, retry, or escalate?

5. Reasoning Traceability

- Is the rationale behind the tool choice observable from logs or metadata?
- Can the evaluation system distinguish between strategic use vs random invocation?

Example Scenarios

Tool call evaluation can be applied in various operational contexts:

- **Simple invocation**: Agent correctly calls a REST API with the expected endpoint and parameters.
- **Multi-turn planning**: Agent chooses a sequence of tools that contribute cumulatively to task resolution.
- **Fallback detection**: Agent attempts a backup tool when the preferred one is unavailable.
- **Hallucinated call prevention**: Agent avoids fabricating tools that were not part of the registered capabilities.
- **Policy enforcement**: Certain tools may be restricted by role or context, evaluators can verify compliance.

Methods and Tools

Approach	Use Case
Static rules	Validate schema, tool name, required args
Golden dataset	Compare to labeled ground truth
Behavior heuristics	Infer consistency, fallback, retries

Approach	Use Case
Semantic evaluation	LLM-based matching of tool intent vs input
Human review	For gray areas or disputed calls

Frameworks like **Azure AI Evaluation**, and custom telemetry services can provide instrumentation, log export, and evaluation pipelines that support tool call introspection at scale.

Best Practices

- **Log consistently**: Include tool name, arguments, results, timestamps, and agent metadata. *(Always ensure sensitive or personally identifiable information (PII) is protected or redacted according to data governance policies)*.
- **Decouple evaluation logic**: Allow swapping evaluation strategies without modifying agent logic.
- **Define tool policies**: Make tool usage contracts explicit to simplify validation.
- **Correlate with task completion**: Understand how tool behavior affects broader goals.

For reference:

- ToolCallAccuracyEvaluator (Microsoft / .NET)
- Agent evaluators

Security

Last updated: 2025-05-12

This document outlines the security considerations and best practices for implementing multi-agent systems in enterprise environments. The architecture prioritizes responsible AI usage, data protection, access control, and observability at every layer.

Core Security Principles

Identity Enforcement

- Agents and orchestrators authenticate via Azure AD or equivalent identity providers.
- Role-based access control (RBAC) governs agent execution and orchestration permissions.
- Ensure mutual authentication between agents using enterprise identity systems (Entra ID, SPIFFE, etc.).
- Use X.509 certificates or JWTs signed by an internal CA to verify agent provenance.
- Support rotating credentials for long-lived agents.

Scoped Agent Capabilities

- Agent registry enforces capability declarations and metadata tagging.
- Only approved capabilities can be executed in runtime.

Data Governance & Storage Policies

- All storage layers (state, conversation history, registry) support encryption-at-rest.
- Retention policies are configurable per agent or orchestrator.

Secure Communication

- All communication between agents, orchestrator, and external tools is encrypted via HTTPS or mutual TLS.
- MCP protocol supports signed payloads and call tracing.

Policy-Controlled Tool Invocation

- MCP server validates tool access policies before invocation.
- Integration layer enforces audit logging for all tool calls.

Memory Management and Redaction

- Short-term memory is scoped to current conversation thread.
- Long-term memory undergoes PII redaction and consent-based storage.

LLM Usage & Guardrails

- Prompt templates include safety instructions.
- Responses are filtered by safety classifiers and optionally passed through moderation APIs.

Auditability and Observability

- Every orchestration and agent call is logged with metadata: timestamp, caller identity, input hash, output hash.
- Logs are shipped to centralized observability platform (e.g., Azure Monitor or OpenTelemetry).

Adversarial Testing & Red Teaming

- Simulate attacks (prompt injection, message corruption, impersonation).
- Use chaos security engineering to validate robustness of inter-agent communication.

Agent Versioning and Rollback Strategy

- Version every agent's logic, prompt configuration, and communication contract (e.g., using SemVer).
- Support graceful coexistence of multiple versions (e.g., v1 agents in production, v2 agents in testing).
- Use feature flags or routing rules to selectively enable agent versions per customer or region.

- Always enable immediate rollback to a previous version if anomalies are detected (e.g., via GitOps or CI/CD workflows).

Business Continuity and User Safety

- Establish SLOs for agent performance (latency, availability, decision quality).
- Define user impact metrics before deployment (e.g., error rate, abandonment rate, CSAT).
- Automate runtime health checks and user experience monitoring tied to release gates.
- Include manual override mechanisms (e.g., pause/disable an agent class) without affecting the overall system.

Enterprise Data Protection in RAG Scenario

Document Indexing Layer

When indexing documents, ensure to include sensitivity and/or permissions (owner, edittable, etc) information in included as a retrivable field.

Retrival Layer

When retriving/vector search for documents, filter on relevant sensitivity or permissions based on agent or user role or identity

Agent (using RAG) Layer

Before generating response from the resulting document, ensure the retrieved documents are viewable by the user as another layer of protection

Indentity Layer

Ensure that the user identity has roles or permissions associated and the identity is secured by following the "Identity Enforcement" section of this guide

Compliance Considerations

- Support for GDPR/CCPA consent flows.
- Optional on-premise deployment for sensitive workloads.
- Integration with enterprise data loss prevention (DLP) systems.

Future Additions

- Integration with confidential computing enclaves (TEE).
- Fine-grained token scopes for agents.
- Agent-level anomaly detection models.

For implementation details and threat modeling scenarios, refer to the Threat Model Worksheet.

Governance

Last updated: 2025-05-13

One of the key aspects of adopting any new technology in an enterprise environment is governance. This becomes even more critical with the adoption of Generative AI (GenAI). While GenAI represents a powerful and transformative technology, its value does not emerge in isolation. On its own, it is just a sophisticated engine — without access to high-quality, governed data, its potential remains untapped.

This becomes especially evident when comparing deterministic systems to AI-driven solutions. Traditional (deterministic) applications are rule-based, predictable, and typically operate within predefined inputs and outputs. Their governance often revolves around access control, compliance, and auditability of code and workflows. However, multi-agent systems powered by GenAI introduce additional governance concerns:

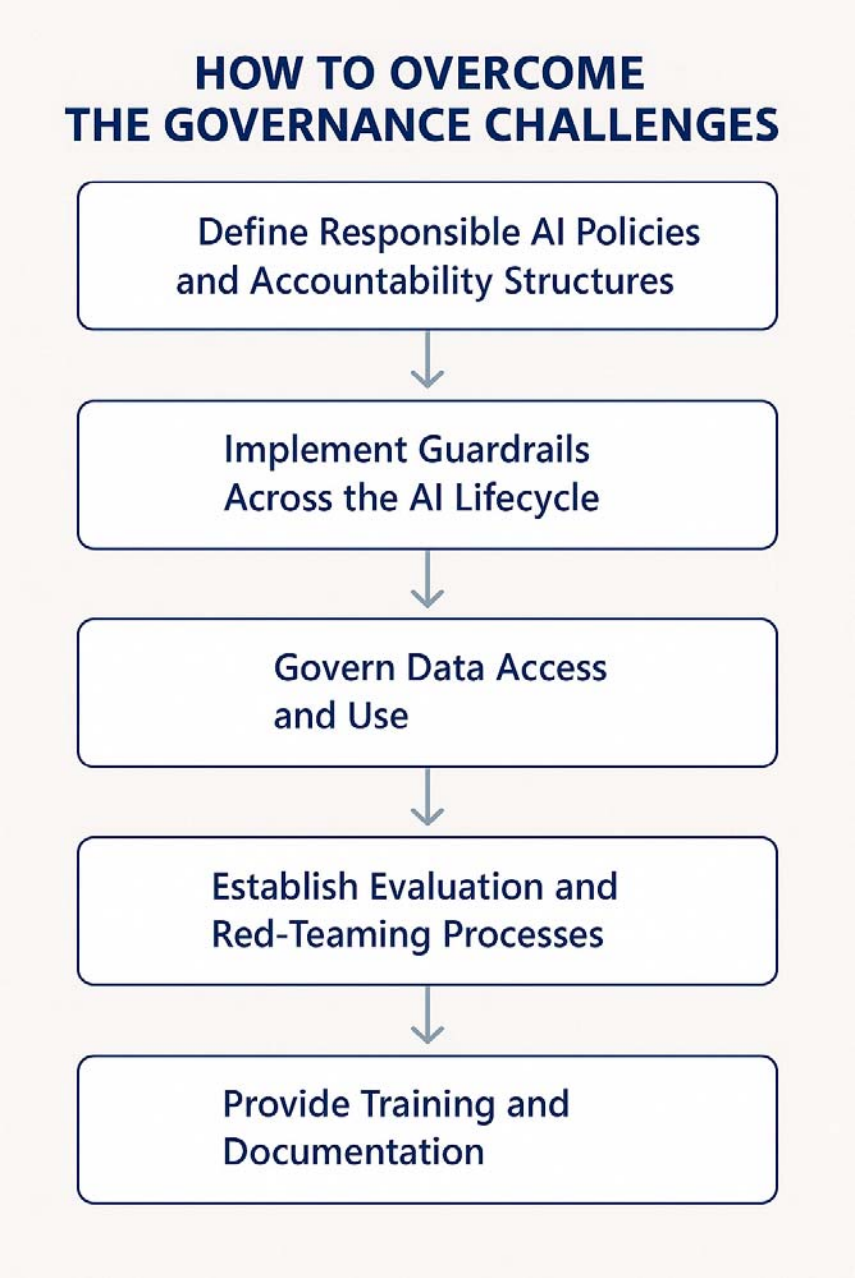
- Data provenance and usage policies: AI agents must access data responsibly and in alignment with enterprise policies.
- Output accountability: As agents act autonomously or semi-autonomously, ensuring traceability and responsibility for generated outputs is essential.
- Operational boundaries: It's critical to define the scope of what agents are allowed to do, particularly in dynamic orchestration settings.
- Model oversight: The lifecycle of the models used (training, fine-tuning, evaluation) must be managed under clear governance to avoid misuse or drift.

Without strong governance in place, organizations risk uncontrolled automation, data exposure, hallucinated outputs, and non-compliance with regulations. Establishing a governance framework early ensures responsible innovation, maintains trust, and paves the way for enterprise-scale adoption of GenAI and multi-agent architectures.

How to overcome the governance challenges

To govern GenAI-powered multi-agent systems effectively, enterprises must establish a clear and actionable Responsible AI (RAI) framework. A governance plan should not only align with ethical and regulatory standards but also enable scalable innovation without compromising security, compliance, or trust.

One strong reference is the Microsoft Responsible AI Standard (v2), which outlines foundational practices to ensure that AI systems are built responsibly. Building on these principles, organizations can create a governance roadmap as follows:



Define Responsible AI Policies and Accountability Structures

- Adopt RAI principles such as fairness, reliability, privacy, inclusiveness, transparency, and accountability.
- Establish a cross-functional governance committee with representation from legal, compliance, IT, security, and AI teams.

- Define clear roles and responsibilities for model development, approval, deployment, and monitoring.

Implement Guardrails Across the AI Lifecycle

- Use tools like Azure AI Content Safety and Prompt Shields to detect and filter unsafe or policy-violating outputs.
- Monitor for model drift, prompt injection, or malicious orchestration in multi-agent systems using telemetry and evaluation pipelines.
- Ensure auditability of agent actions via structured logging, versioning, and traceable decision trees, check Observability section.

Govern Data Access and Use

- Classify data according to sensitivity and restrict agent access based on data governance policies.
- Apply role-based access control (RBAC) and monitor for unauthorized data usage.
- Use data loss prevention (DLP) techniques and encryption to reduce exposure risk.

Establish Evaluation and Red-Teaming Processes

- Continuously evaluate system behavior through human-in-the-loop feedback loops.
- Conduct adversarial testing (red-teaming) to identify failure modes in agent collaboration or unintended capabilities.
- Track hallucination rates and bias impacts using Azure AI Evaluation tools or similar frameworks.

Provide Training and Documentation

- Train teams on your RAI framework and decision-making criteria.
- Document system design choices, model limitations, data sources, and intended uses.
- Ensure users and stakeholders understand system capabilities and boundaries, particularly in customer-facing scenarios.

Discuss this page

Experimentation to Productization

Last updated: 2025-05-29

This document outlines the process of transitioning an agent from experimentation to production. It defines the key steps required to ensure that an agent is reliable, performant, secure, and observable in real-world deployments.

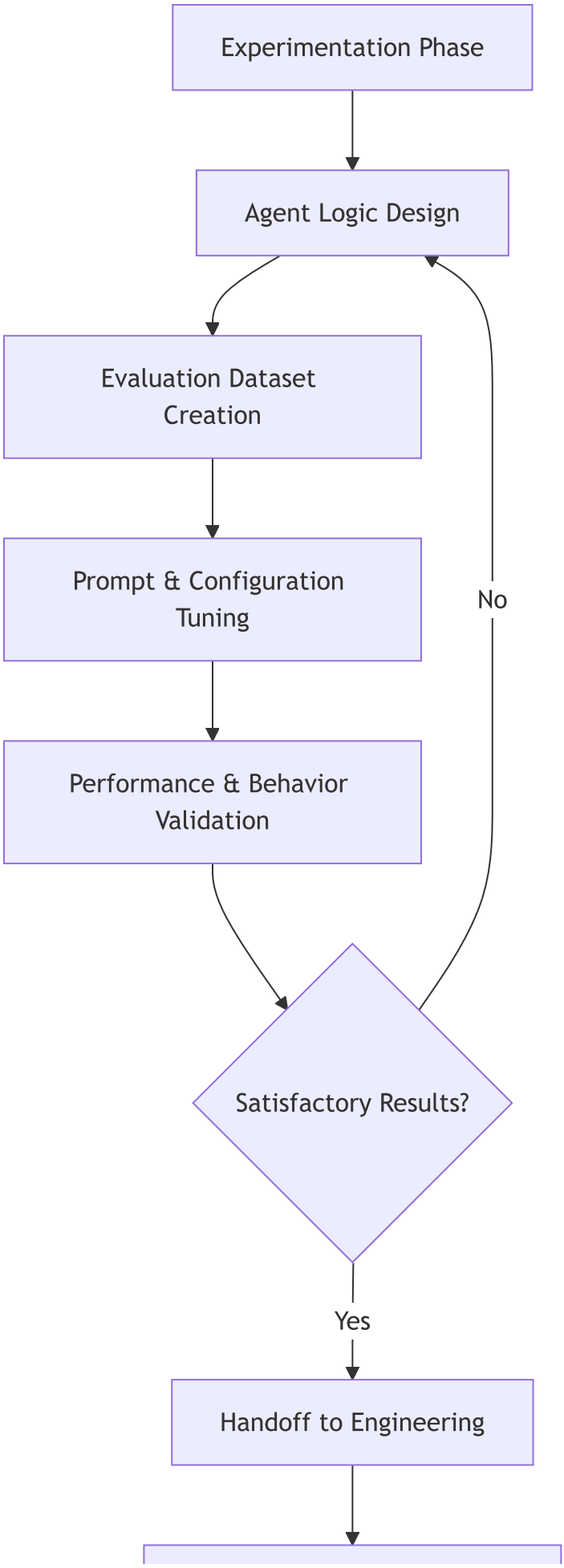
Overview

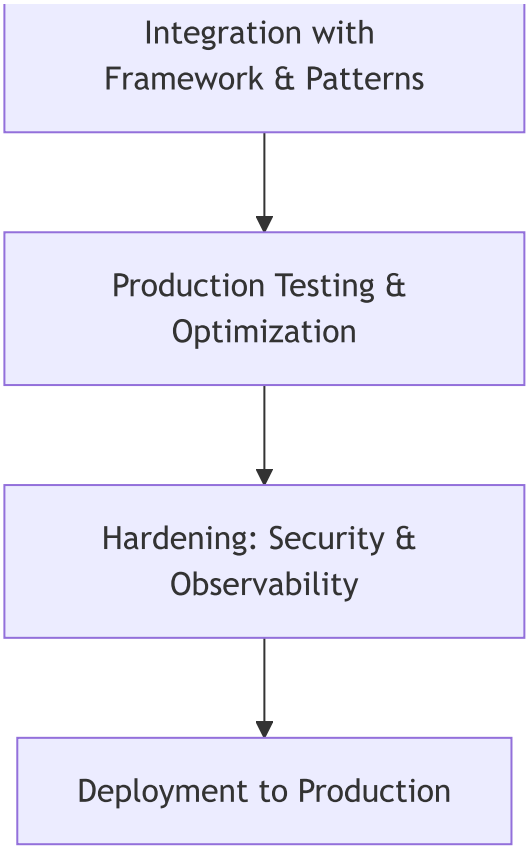
Agents are typically developed and tested in controlled environments during the experimentation phase. At this stage, the focus is on solving a particular problem, designing the agent's logic, and iterating over data preparation, prompt strategies, and configuration parameters.

Experimentation often begins with a Data Science team, using tools such as Jupyter Notebooks or SaaS platforms like Azure AI Foundry Playground. The goal is to identify the most effective orchestration or configuration strategy aligned with business objectives. Evaluation data and test suites are also created to assess performance.

Once results are satisfactory, the Software Engineering team takes over to productize the agent using enterprise-level tools and patterns. This includes refactoring the agent if necessary, ensuring compatibility with frameworks, and aligning with organizational standards.

Flow Diagram


Experimentation Phase
Agent Logic Design
Evaluation Dataset Creation
Prompt & Configuration Tuning
Performance & Behavior Validation
Satisfactory Results?
No
Yes
Handoff to Engineering



Key Stages

1. **Testing**
    - Conduct unit, integration, and end-to-end tests.
    - Prioritize real-world scenarios through automated pipelines.
    - Leverage test suites created during experimentation.
    - **Ensure deterministic tests for routing and decision-making logic** in multi-agent systems, enabling predictable outcomes and traceability during orchestration.
    - Use synthetic inputs and golden outputs to validate routing logic over time.
2. **Documentation**
    - Provide API references, usage guides, and examples.
    - Include onboarding instructions for integration teams.
    - Ensure reproducibility of the configuration and behavior.
3. **Performance Optimization**
    - Profile the agent to identify performance bottlenecks.
    - Optimize latency-critical paths and consider parallelism.
    - Validate load characteristics using stress tests.
4. **Security**

- Enforce secure data handling and access controls.
- Apply encryption in transit and at rest.
- Use RBAC and policy-based execution controls.

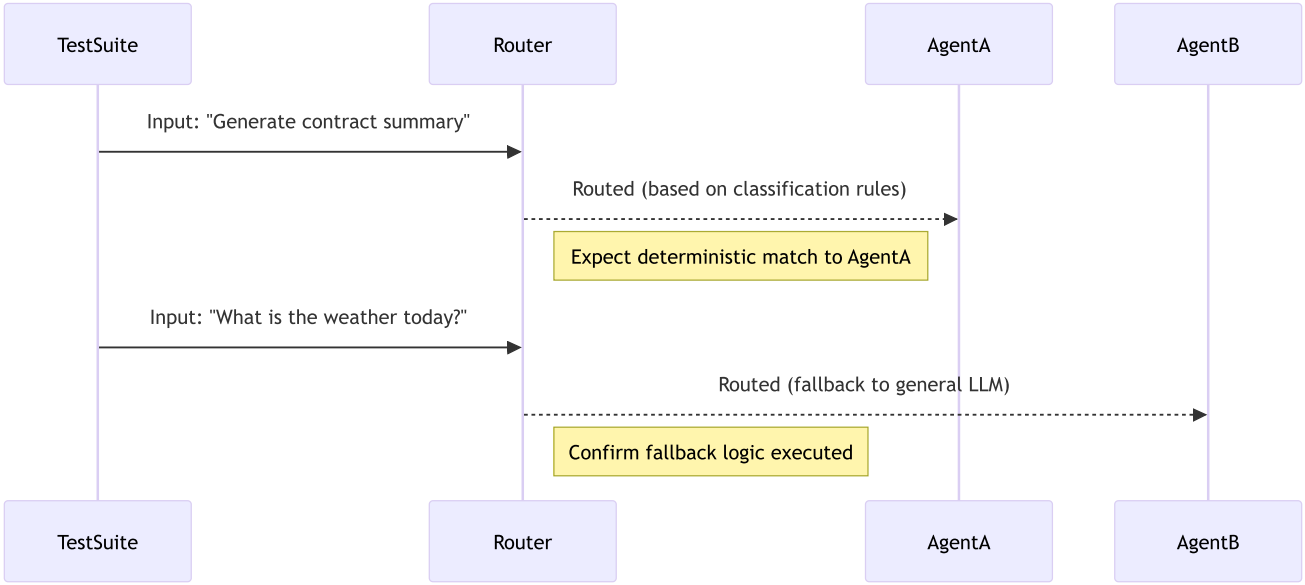
Refer to the Security Guide for more details.

5. **Monitoring and Observability**

   - Instrument agents with logging, metrics, and tracing.
   - Capture operational health and user interactions.
   - Integrate with enterprise observability platforms.

   Refer to the Observability on Agents guide for more details.

Deterministic Routing Validation Example



Integration with DevOps and DataOps

As with any production-grade system, agents should be subject to continuous validation and improvement. See the DevOps and DataOps on Agents Lifecycle guide for CI/CD integration strategies, data drift monitoring, and ongoing model evaluation pipelines.

This includes automated retraining triggers, performance benchmarking, and model governance integration.

Common Pitfalls to Avoid

- Lack of reproducibility from experimentation to production.
- Overfitting to sandbox or evaluation data.
- Insufficient governance checkpoints (e.g., security assessments, RAI reviews).
- Poor observability and incident response readiness.

Governance and Testing Recommendations

- Define clear **gates**: including Responsible AI reviews, security audits, and performance benchmarks.
- Apply advanced testing strategies: chaos testing, synthetic queries, and eval frameworks with regression baselines.

Conclusion

By following these structured steps, developers and teams can confidently transition agents from prototype to production in a secure, observable, and scalable way. This structured process reduces risk, improves maintainability, and enables faster time-to-market.

As organizational maturity increases, these transitions become more streamlined and automated—empowering continuous innovation and unlocking the full potential of agent-based systems in real-world deployments.

Context Engineering

Last updated: 2025-07-03

In Generative AI, `context` is the information a language model uses to guide its outputs, such as instructions, memory, and external data. Context engineering is the practice of designing, preparing, and managing this information to shape model behavior and results.

Context engineering is especially critical in multi-agent systems, where multiple agents collaborate to solve complex tasks. In these environments, every improvement in context quality and data volume is amplified: even small gains from an agent can have a significant impact as the number of agents increases. Well-optimized context not only enhances individual agent effectiveness but also drives overall system efficiency and scalability.

Crafting and delivering the most relevant context to each agent is essential for achieving high-quality, efficient, and cost-effective results. This chapter introduces strategies for optimizing context, focusing on two main objectives:

- **Minimizing irrelevant content:** Filter out outdated, redundant, or noisy information to reduce misleading results, token usage, costs, and latency.
- **Maximizing usefulness for each agent:** Curate and structure context to prioritize relevant, actionable, and timely information—such as recent or high-confidence data, summaries, or dynamically adapted content—empowering agents to make better decisions.

> In some cases, further optimization can be achieved by delegating tasks to simpler or more efficient components instead of language models.

Effective context optimization improves response quality, reduces computational overhead, and enables scalable, collaborative multi-agent systems.

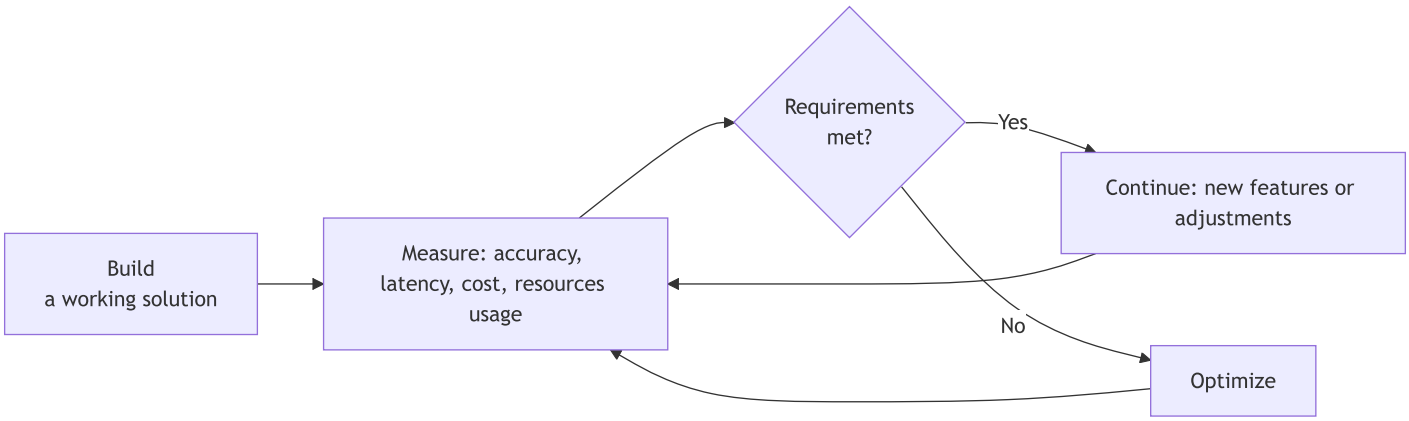
The Iterative Optimization Loop

Last updated: 2025-07-03

Premature optimization can lead to unnecessary complexity and wasted effort. Instead, it's recommended to follow an iterative approach to context optimization:

1. **Get it right:** Start with a simple, correct solution that meets the functional requirements.
2. **Test it right:** Measure performance and cost using real data and scenarios.
3. **Analyze:** Evaluate whether the results meet your requirements.
4. **Optimize:** If not, make targeted improvements, prioritizing the changes that are likely to have the greatest impact.
5. **Repeat:** Re-measure and continue the loop as needed.

This process ensures that optimizations are both necessary and effective.



Preventing optimization overload

Avoid attempting several optimizations at the same time. While it may seem efficient, making many changes at once can increase cognitive load, obscure the impact of individual improvements, and make troubleshooting more difficult.

Focusing on one or a few targeted optimizations at a time allows you to clearly measure their effects, reduces the risk of introducing new issues, and helps maintain a clear understanding of the system's behavior. This disciplined, incremental approach not only streamlines the optimization process but also makes it easier to roll back changes if needed and ensures that each adjustment is truly beneficial.

Discuss this page

Specialized Agents Context for Orchestration

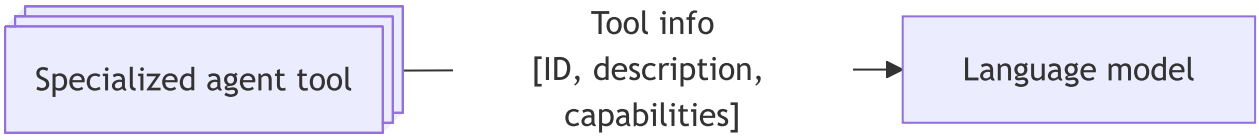
Last updated: 2025-07-07

Effective orchestration requires the language model to be aware of available specialized agents, their capabilities, and how to invoke them. Below are common patterns for exposing this information, along with their key characteristics and tradeoffs.

Context Injection Patterns

1. Static Tools

Pre-load the language model with a static number of tools representing the specialized agents, including their descriptions and capabilities.



Key characteristics

- **Fast inference**: All specialized agents' information is immediately available to the language model.
- **Predictable**: The set of agents and their capabilities are fixed at runtime.
- **Simplicity**: No need for dynamic lookups or external dependencies.

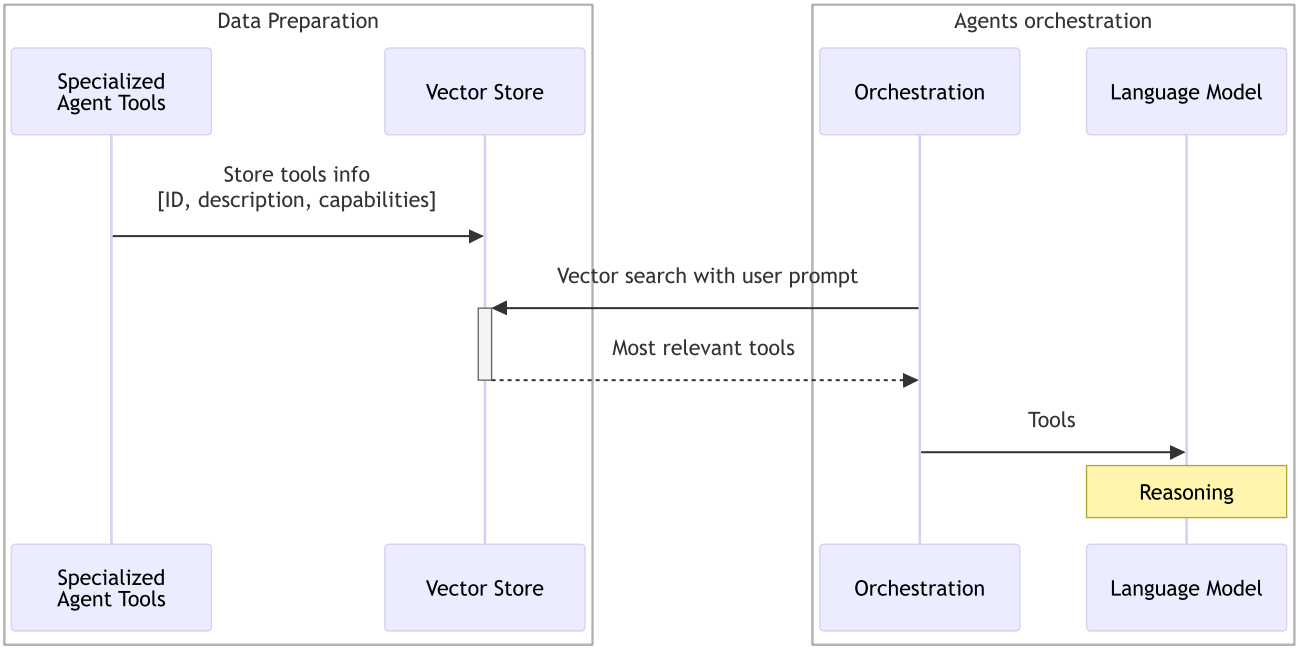
Tradeoffs

- **Orchestration effectiveness**: Large number of specialized agent tools may affect the language model's ability to select the appropriate ones to solve the problem. For instance, the OpenAI Function Calling Guide suggests between 10-20 tools per request.
- **Scalability**: Large number of specialized agent tools may exceed model context limits.
- Adding or removing agents requires a new orchestration agent deployment.

---

2. Vector Similarity Search over Static Tools

Store specialized agent tools information in an in-memory or external vector database. When orchestration is needed, perform a vector search using the user prompt to retrieve the most relevant agents.



> Modern vector stores such as Azure AI Search offers additional features on top of traditional vector search to improve search relevance scores. For more info, see semantic ranking.

Key characteristics

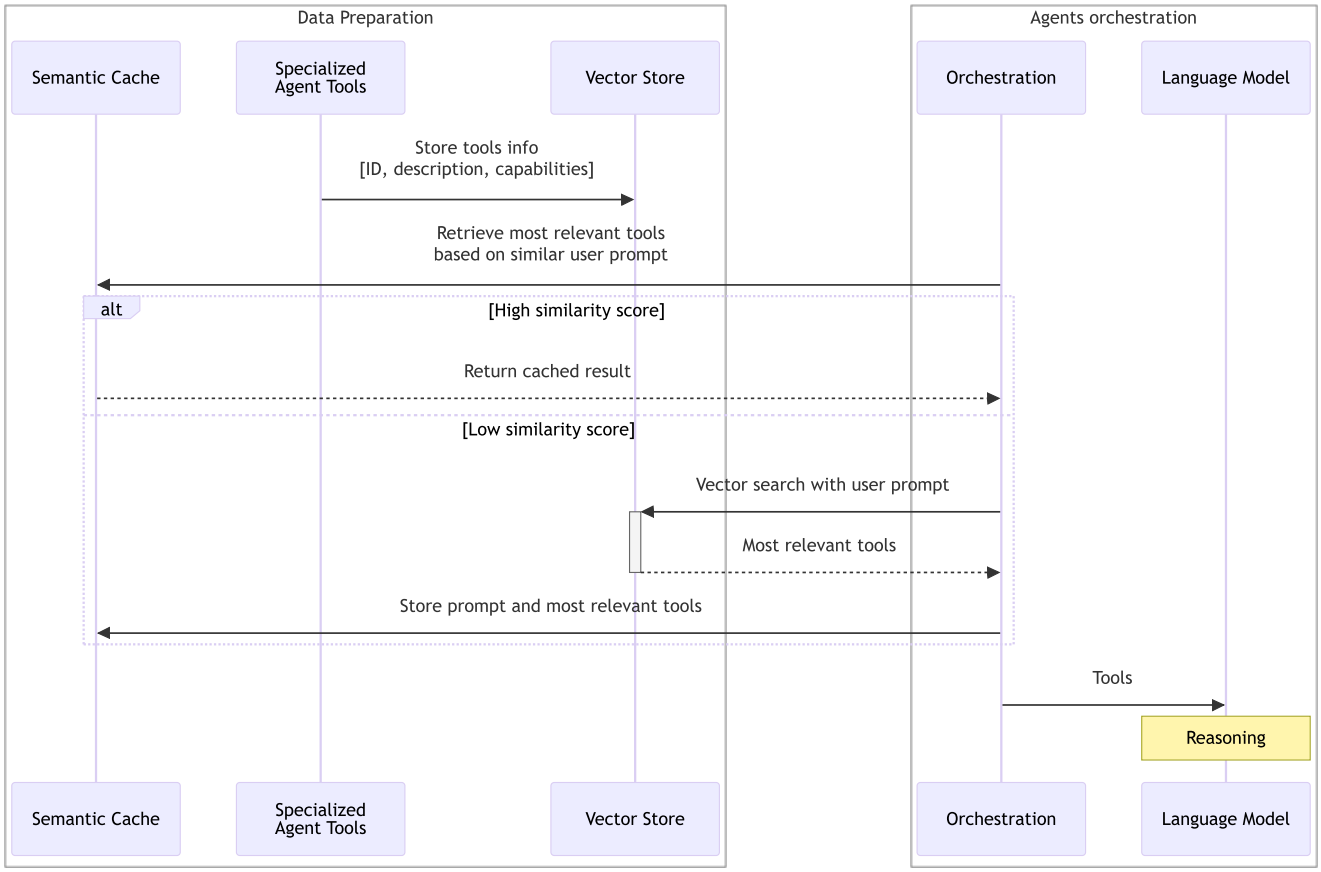
- **Optimized context**: Only the most relevant specialized agents are considered as context for the language model, reducing AI costs and improving the language model's ability to efficiently select agents.
- **Scalability**: Supports large numbers of specialized agent tools without overloading the language model context.

Tradeoffs

- **Operational overhead**: Vector search requires an additional overhead of provisioning, scaling, updating and monitoring the database, selecting the Approximate Nearest Neighbor (ANN) algorithm, selecting embedding models and formats, as well as tuning data and queries for better precision and performance over time.
- **Data retrieval mismatch**: Vector search introduces potential risks of *false positives*, retrieving irrelevant yet semantically close specialized agent tools in vector space (typically

not harmful to the model's reasoning in small numbers but increasing token usage), and *false negatives*, where relevant tools are missed due to content quality issues.

Optionally, a semantic cache can use prior user prompts and their corresponding most relevant tools to address similar user prompts without the need of executing vector search queries on every request.



Key characteristics

- **Reduced Latency**: By utilizing cached data for similar user prompts, the need for executing vector search queries for every request is minimized, leading to faster response times.
- **Cost Efficiency**: Avoids repeated database querying for similar prompts, reducing operational costs.

Tradeoffs

- **Data consistency management**: Managing cache consistency can be challenging, as updates to the vector database may not immediately reflect in the cache. Using a time-to-live (TTL) for cached items helps to minimize this risk.
- **Storage Overhead**: Maintaining a semantic cache requires additional storage resources, which can grow with the number of cached items. Using a time-to-live (TTL) for cached items helps avoid it growing to an enormous size.

- **Potential for Staleness**: Cached results might become outdated if the underlying agent tools' information changes, leading to irrelevant or incorrect results.

References

- Semantic Kernel Decisions: Context-based function selection
- Semantic Kernel: Multi-agent Orchestration
- Semantic ranking in Azure AI Search
- Introduction to semantic cache

Tools Context for Agents

Last updated: 2025-07-14

The ability to perform tasks and interact with external systems is what transforms language models from passive responders into active problem solvers. The design of those tools directly determines how effectively specialized agents can reason about them and coordinate their use in multi-agent workflows.

This section provides recommended practices for designing tools that are AI-friendly (descriptive and concise purposes and input/output formats), avoiding common pitfalls such as over-granularity or hidden side effects that can significantly influence reliable outcomes. These recommendations are crucial for both orchestrator agents and specialized agents.

Recommended practices

1. Handling Deterministic Logic

Contextual data that is static or deterministic (i.e. doesn't rely on the user prompt), should be handled outside of language model interactions. For example:

- **Preloading user data and preferences**: Ensures context consistency and eliminates redundant queries.
- **Data validation, sorting, or filtering**: Simplifies downstream processing and ensures reliable data.
- **Computing results for business rules or domain-specific calculations**: Optimizes execution by reducing repetitive reasoning within the language model.

> Data can be dynamically injected into the language model instructions, or injected in the tool(s) within the lifecycle of the request.
>
> Some model operations, such as OpenAI Chat Completions, offer a `metadata` field only used for conversation history storage, useful for evaluation and distillation (technique for training models with the outcomes of another model).

This approach provides the following benefits:

- **Enhanced Performance**: By preloading static and dynamic data, agents avoid unnecessary computation during runtime, ensuring faster execution.
- **Simplified Tool Design**: Tools become simpler and more focused on handling dynamic interactions.
- **Operational Cost Reduction**: Minimizes resource usage by avoiding redundant operations.
- **Scalability and Flexibility**: Both static and dynamic data can be efficiently managed across multiple agents or systems, enabling dynamic data updates without embedding them into language model interactions.

2. Balancing Tool Granularity and Responsibility

Developers tend to adopt the Single Responsibility Principle (SRP), a common practice in traditional software development, to design tools that perform one specific task. While this approach promotes clarity and reusability, it can introduce processing overhead for language models, including increased reasoning complexity, higher token consumption, and additional network latency due to multiple tool invocations.

On the other side of the spectrum, designing a single tool to handle multiple responsibilities can reduce overhead but often leads to decreased reusability, increased implementation complexity, and a higher risk of parameter mismatches or incorrect assumptions by the model.

Recommendations

- **Group closely related responsibilities:** Combine tasks that are frequently used together or share similar input/output structures into a single tool to minimize overhead.
- **Avoid excessive generalization:** Do not overload tools with unrelated responsibilities, as this can make them harder to maintain and use correctly.
- **Optimize for model reasoning:** Consider how the language model selects and uses tools. Overly granular tools may increase reasoning steps, while overly broad or generic tools may confuse parameter mapping.
- **Monitor usage patterns:** Analyze tool invocation logs to identify bottlenecks or frequent multi-tool workflows that could be streamlined.
- **Iterate and test:** Continuously refine tool boundaries based on real-world usage and model performance, aiming for a balance between efficiency and clarity. The previous Iterative Optimization Loop section offers a plan to do this in a structured way.

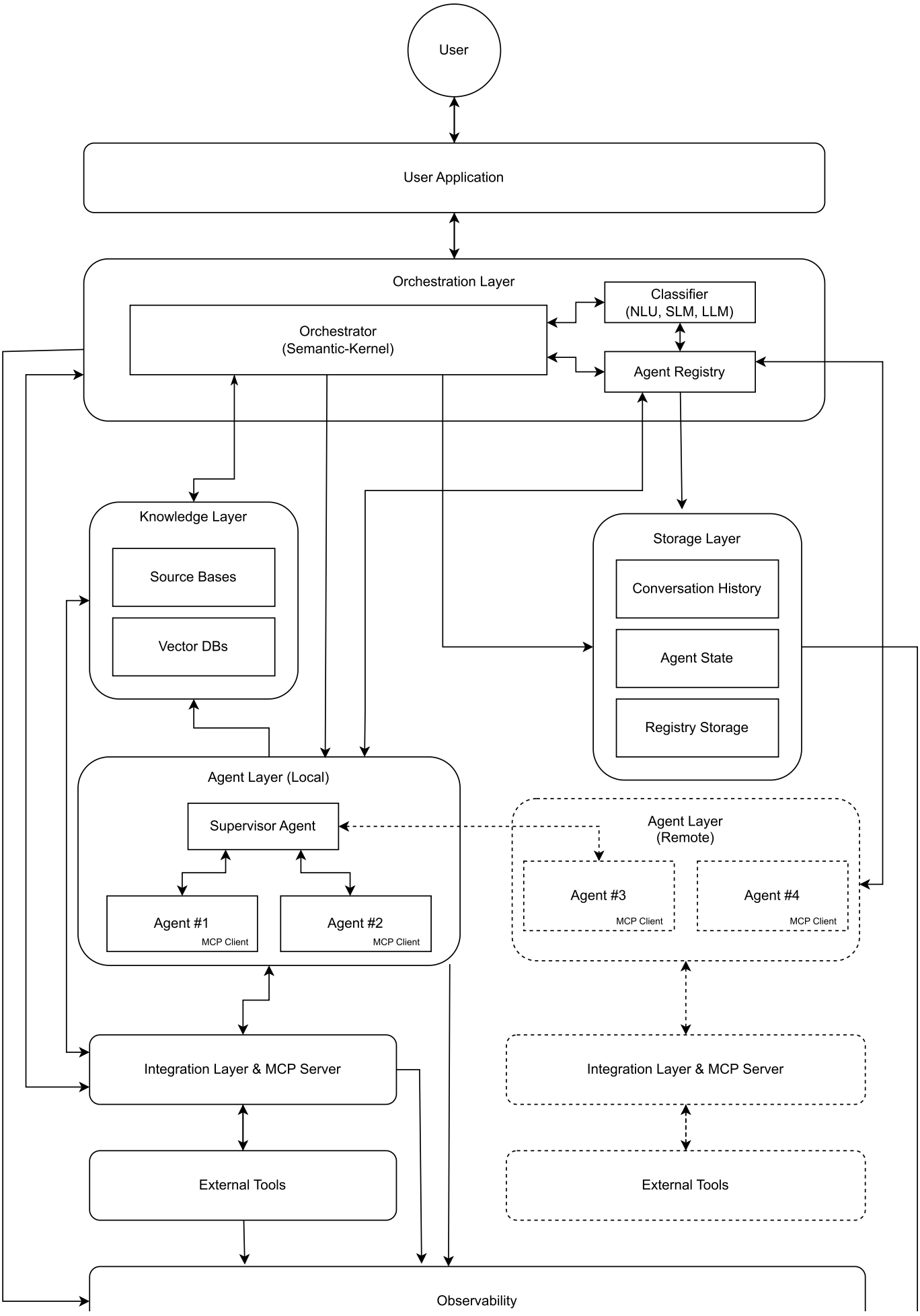
> The Semantic Kernel: Make plugins AI-friendly documentation gives an agnostic view of recommendations to design clear and concise tool schemas for language models.

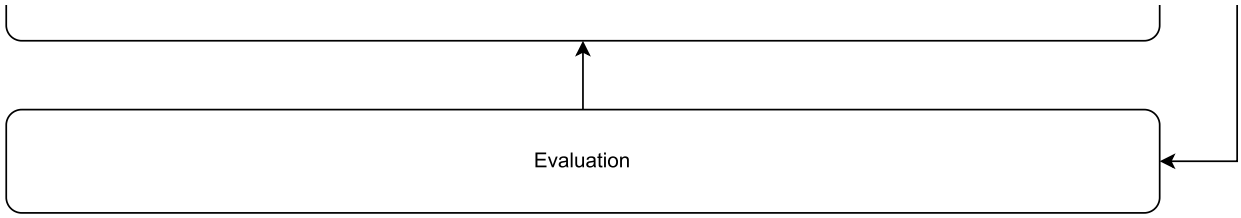
Discuss this page

Reference Architecture

Last updated: 2025-08-18

The architecture below illustrates a modular and governed multi-agent system, supporting both local and remote agents through a central orchestration layer. At its core, the Orchestrator (e.g., Semantic Kernel) coordinates agent interactions, consults a classifier for intent routing, and uses a registry for agent discovery and lifecycle management. The system integrates with knowledge bases and vector databases, and maintains context and state through a persistent storage layer. Integration with external tools is supported through an MCP (Model Context Protocol) server. This design ensures flexibility, extensibility, and strong control boundaries between components, allowing seamless onboarding of new models, tools, and communication patterns.


User
User Application
Orchestration Layer
Classifier
(NLU, SLM, LLM)
Orchestrator
(Semantic-Kernel)
Agent Registry
Knowledge Layer
Source Bases
Vector DBs
Storage Layer
Conversation History
Agent State
Registry Storage
Agent Layer (Local)
Supervisor Agent
Agent #1
MCP Client
Agent #2
MCP Client
Agent Layer
(Remote)
Agent #3
MCP Client
Agent #4
MCP Client
Integration Layer & MCP Server
Integration Layer & MCP Server
External Tools
External Tools
Observability



Components' breakdown

User Application

- The interface layer that facilitates user interaction with the multi-agent system.
- It abstracts the complexity of the underlying architecture and provides a consistent, user-friendly experience.

How it works: Typically implemented as web applications, mobile apps, chat interfaces, APIs, or embedded widgets in other applications. It handles:

- User authentication and session management
- Input formatting and validation
- Response rendering and formatting
- Error handling and user feedback
- User preference management

Orchestrator (Semantic Kernel)

The central coordination component that manages the flow of requests and responses throughout the system. It provides unified management, ensuring appropriate routing, maintaining context, and handling the lifecycle of requests.

How it works: The orchestrator receives requests from the User Application, determines how to process them, coordinates with the appropriate components, maintains state, and eventually returns responses.

Implementation

Often structured as a core orchestration service with:

- Request/response lifecycle management
- Context preservation across interactions
- It determines the appropriate functions, plugins, or agents to invoke using planners or semantic functions.

- Fallback and error recovery mechanisms

Semantic Kernel provides:

- Semantic function orchestration (organizing and sequencing AI functions)
- Memory and context management
- Plugin architecture for extensibility
- Planners that can decompose complex tasks into simpler steps
- Integration with various AI models and services
- Native support for multi-agent scenarios
- Cross-platform compatibility (multiple programming languages, check the features compatibility)

Classifier (NLU, SLM, LLM)

The component responsible for understanding user inputs and determining the appropriate routing within the system. It ensures that user requests are properly understood and directed to the most suitable agent, improving response quality and system efficiency.

How it works: Analyzes the content, context, and intent of user inputs to categorize them and determine appropriate handling.

> The approach involves using options ranging from less to more expensive ones, NLU -> SLM -> LLM | SML based on certainty to determine the use of intent or continuation. If no intent is detected by the end of the process, return "IDK" (I Don't Know).

Implementation:

It can be implemented using a variety of technologies, including but not limited to:

- NLU (Natural Language Understanding): Extracts intent and entities
- SLM (Statistical Language Models): Used for pattern recognition and classification
- LLM | SLM (Large Language Models | Small Language Models): Provide sophisticated understanding of complex inputs

Agent Registry

See Agent Registry Page for more detail

Knowledge Layer

Repositories of structured and semi-structured knowledge that agents can reference. They provide domain-specific information that enhances agent capabilities beyond what's possible with generic AI models alone.

How it works: Organizes knowledge into accessible formats, often including taxonomies, ontologies, and semantic relationships.

Implementation:

- Document databases
- Knowledge graphs
- Content management systems with APIs
- Specialized domain knowledge bases
- FAQ systems and support documentation
- Expert-curated information repositories

> Recommended practices: Knowledge should be properly structured, tagged, versioned, and regularly updated to maintain accuracy and relevance.

Supervisor Agent

A specialized agent responsible for coordinating the activities of other agents to solve complex tasks. It enables decomposition of complex tasks into subtasks that can be handled by specialized agents, then synthesizes their outputs into coherent responses.

How it works: Receives high-level tasks, breaks them down, delegates to appropriate specialized agents, monitors progress, aggregates results, and ensures overall task completion.

Implementation:

- Task planning algorithms
- Dependency tracking systems
- Agent selection and routing logic
- Result aggregation and synthesis capabilities
- Error handling and retry mechanisms
- Conflict resolution for contradictory inputs
- High-level reasoning models (usually LLMs or SLMs)

Recommended practices

- Monitor agent overlap in terms of knowledge domain and action scope to prevent redundancy and confusion.
- Avoid keeping highly similar agents separate, as this can degrade the performance of the orchestrator or intent classifier.
- Refactor or group similar agents under a shared interface or capability to streamline classification and routing.
- Introduce agent supervisors as the architecture scales across domains—these components help manage and abstract groups of related agents.
- Use hierarchical organization (e.g., supervisor → agent group) to maintain clarity, scalability, and ease of intent resolution.

Agent #1, #2, #3, #4 (with MCP Client)

Specialized AI agents designed to handle specific domains, tasks, or capabilities. Domain specialization allows for deeper expertise and better performance in specific areas compared to general-purpose agents.

How it works: Each agent focuses on a particular domain (e.g., finance, healthcare, coding) or function (e.g., summarization, research, creative writing), applying specialized knowledge, models, or techniques to user requests.

Implementation:

- Domain-specific LLM fine-tuning
- RAG (Retrieval-Augmented Generation) with domain knowledge
- Specialized algorithms for domain-specific tasks
- Local or remote execution depending on resource requirements
- Self-assessment of capability and confidence levels

MCP Client Component

- Enables standardized communication with external tools via MCP
- Manages discovery of available tools and capabilities
- Handles authentication and authorization for tool access
- Maintains connection state and manages re-connections
- Formats requests and responses according to MCP specification
- Provides tool usage analytics and error handling

Differences between Local and Remote Agents

- Local agents run within the same environment as the orchestrator
- Remote agents operate across network boundaries
- Remote agents require additional security and reliability considerations
- Communication patterns differ (in-memory vs. network protocols)
- Deployment and scaling strategies vary significantly
- Resource management approaches differ substantially

Conversation History

A persistent store of user-agent interactions and conversation flows. It enables context-aware responses, supports learning from past interactions, and provides an audit trail of system behavior.

How it works: Records each turn in a conversation, maintaining user inputs, agent responses, and associated metadata in a structured, queryable format.

Implementation:

- Specialized conversation stores or time-series databases
- Hierarchical data models (user → session → conversation → turn)
- Indexing for efficient context retrieval
- Compression and archiving strategies for older data
- Query optimization for contextual lookups
- Privacy controls and data retention policies

Agent State

Persistent storage of agent operational status, configuration, and runtime state. It enables continuity across sessions, recovery from failures, and adaptation based on past experiences.

How it works: Maintains both static configuration and dynamic runtime state for each agent, allowing them to resume operations and maintain learned behaviors.

Implementation:

- Key-value stores for fast state access
- State versioning for consistency
- Snapshot mechanisms for point-in-time recovery
- Caching strategies for performance optimization
- Conflict resolution for concurrent updates
- State migration for version compatibility

Registry Storage

Specialized storage for the Agent Registry, maintaining agent metadata, capabilities, and operational history. It provides the persistent data layer for the Agent Registry, ensuring consistent agent information across system restarts and updates.

How it works: Stores comprehensive information about each agent, including capabilities, endpoints, security credentials, performance metrics, and version history.

Implementation:

- Structured database (relational or document)
- Query-optimized schema for capability lookups
- Transaction support for consistent updates
- Versioning for agent evolution tracking
- Audit logging for security compliance
- Backup and recovery mechanisms

> Best practices: Implementing appropriate access controls, regular backup procedures, and efficient querying patterns for agent discovery

---

Integration Layer & MCP Server

A standardized interface layer that connects agents to external tools, services, and data sources. It provides a consistent way for agents to access external capabilities without needing to implement custom integrations for each tool.

How it works: Implements the Model Context Protocol (MCP) to expose tools as a standardized service that agents can discover and invoke.

Implementation:

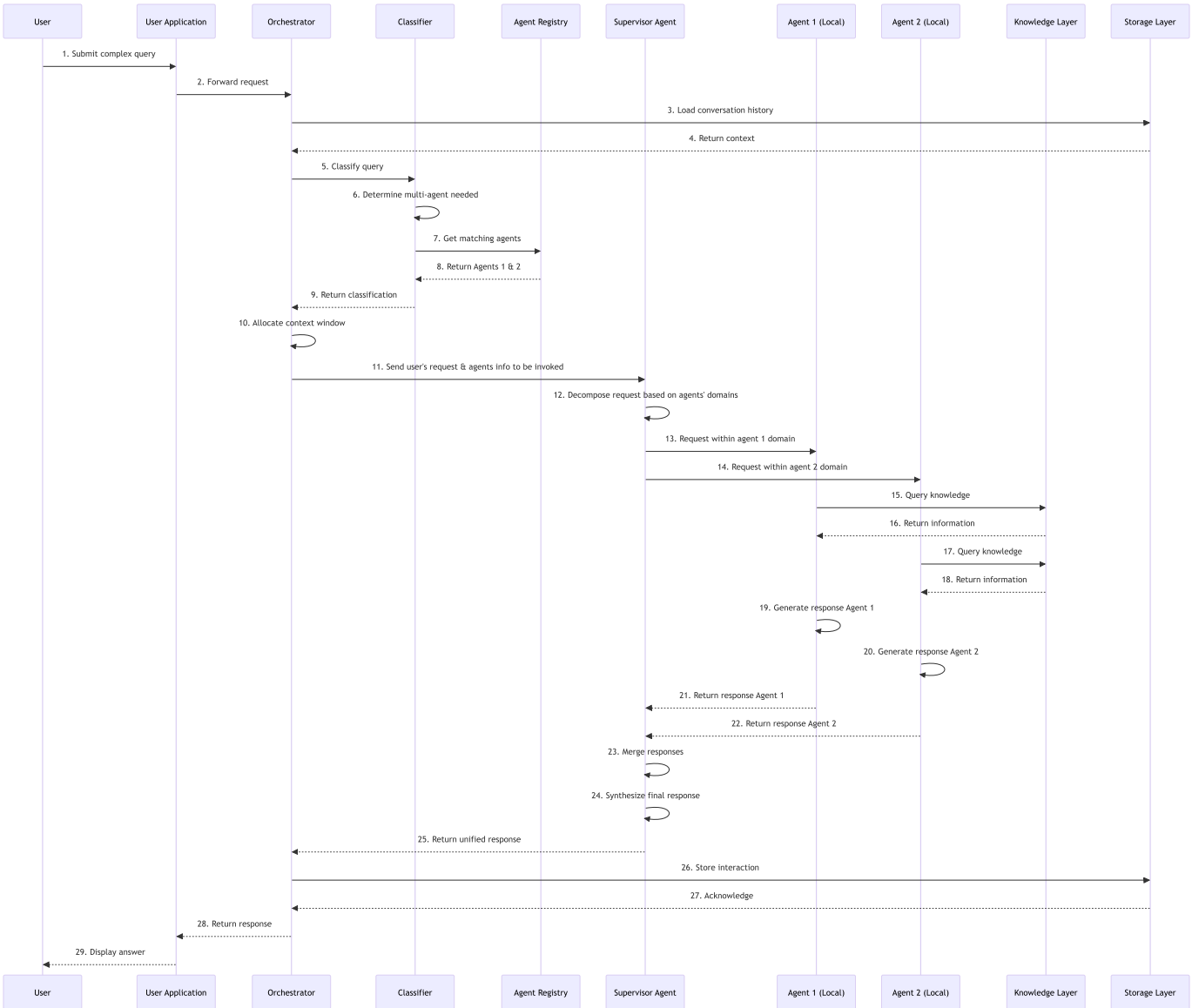
- MCP server implementation exposing tool APIs
- Authentication and authorization controls
- Request validation and error handling
- Tool discovery and capability description
- Usage monitoring and rate limiting
- Versioning and backward compatibility

Key sub components

1. Tool Adapters: Convert native tool APIs to MCP-compatible formats
2. Security Gateway: Manages authentication and authorization
3. Request Router: Directs requests to appropriate tools
4. Response Formatter: Ensures consistent response formats
5. Monitoring System: Tracks usage, performance, and errors

Best practices: Implementing robust security controls, comprehensive monitoring, and maintaining clear documentation of exposed tool capabilities.

Sequence diagram



Patterns

Below are the most foundational patterns that shaped this architecture. For a complete catalog of design patterns used across scenarios, visit the Full Pattern Reference.

List of patterns that guided the proposed architecture.

1. Semantic Router + LLM Fallback
2. Dynamic Agent Registry (Service Mesh for Agents)
3. Semantic Kernel Orchestrator with Skills
4. Local & Remote Agent Execution
5. Separation of Concerns Across Layers (Onion Architecture for Agent Systems)
6. MCP Integration for Agent-Tool Communication
7. RAG (Retrieval-Augmented Generation) Pipeline
8. Conversation-Aware Agent Orchestration (Contextual state + history memory)
9. Agent to Agent communication Agent to Agent Communication Patterns

---

Multi-Agent Patterns Reference

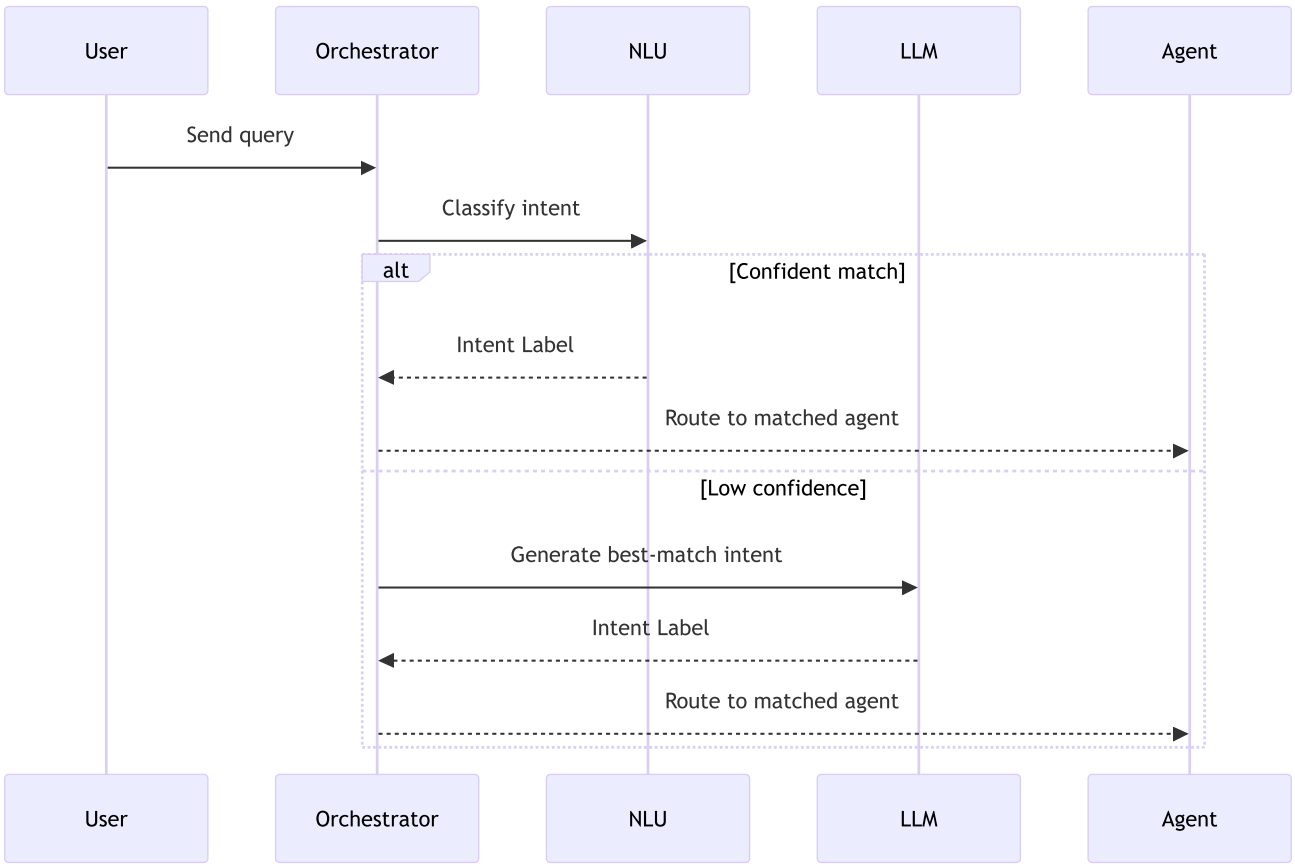
Last updated: 2025-05-14

This document catalogs key design patterns used in the Multi-Agent Reference Architecture. Each pattern contributes to the system's modularity, scalability, governance, or performance.

1. Semantic Router with LLM Fallback

Intent-based routing optimized for cost and performance.

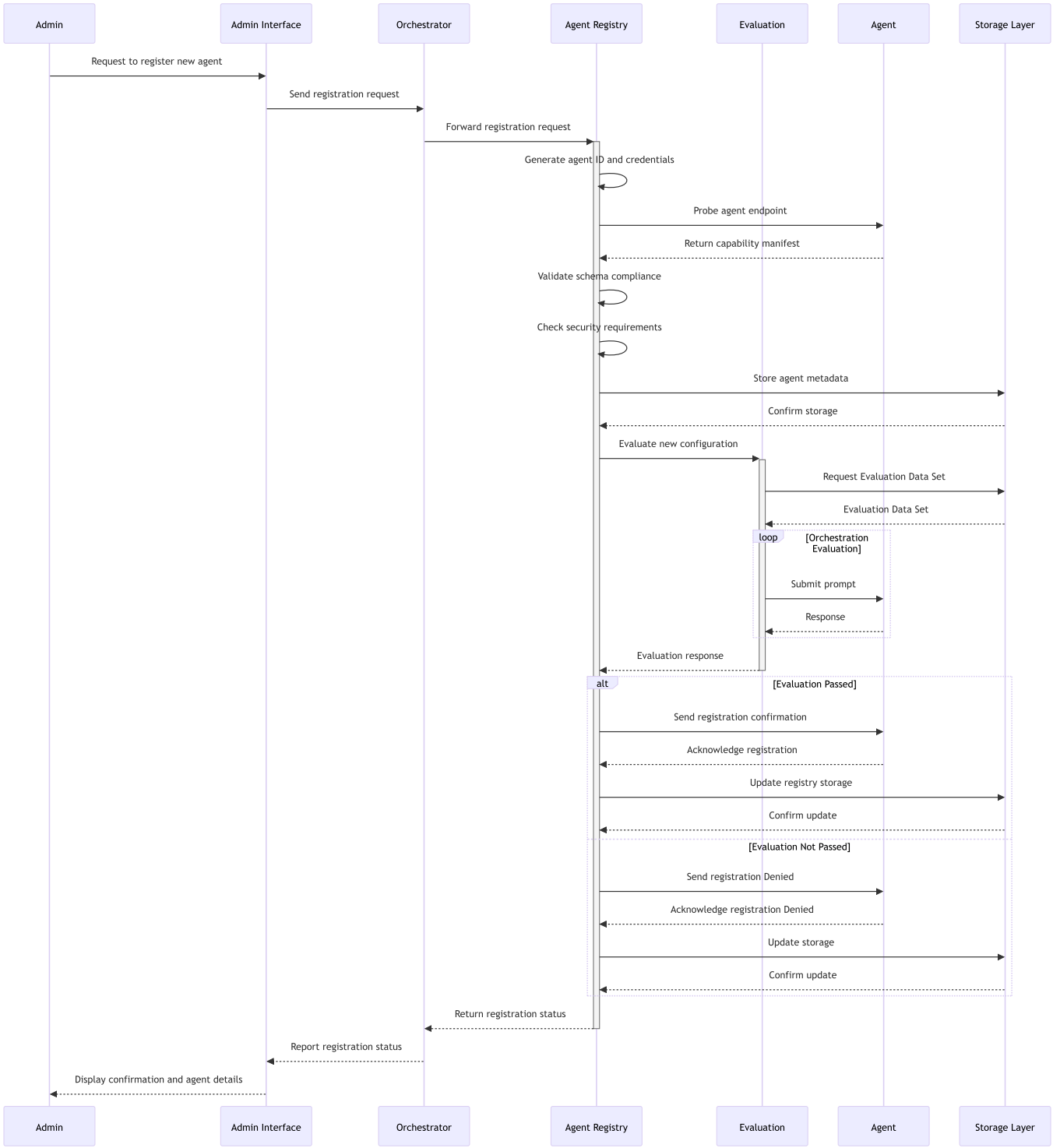
- Use a lightweight NLU or SLM classifier for initial routing.
- If classifier confidence is low, escalate to a more expensive LLM.
- Benefit: Reduces LLM usage while maintaining accuracy.



2. Dynamic Agent Registry (Service Mesh for Agents)

Agent discovery based on capabilities and metadata.

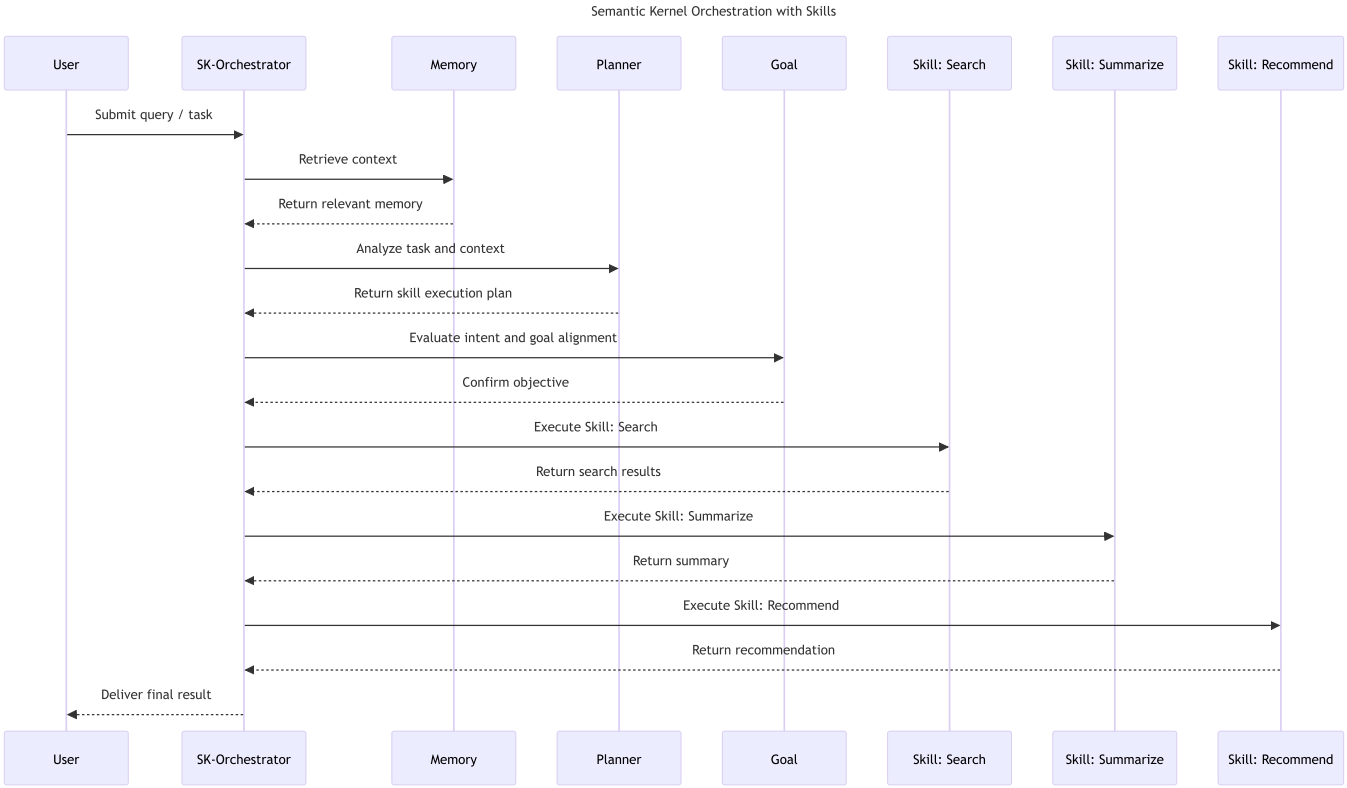
- Agents register with descriptors (capabilities, tags, embeddings).
- Registry supports runtime resolution by orchestrator.
- Benefit: Supports plug-and-play extensibility and self-healing behavior.



3. Semantic Kernel Orchestration with Skills

Composable orchestration using reusable agent capabilities.

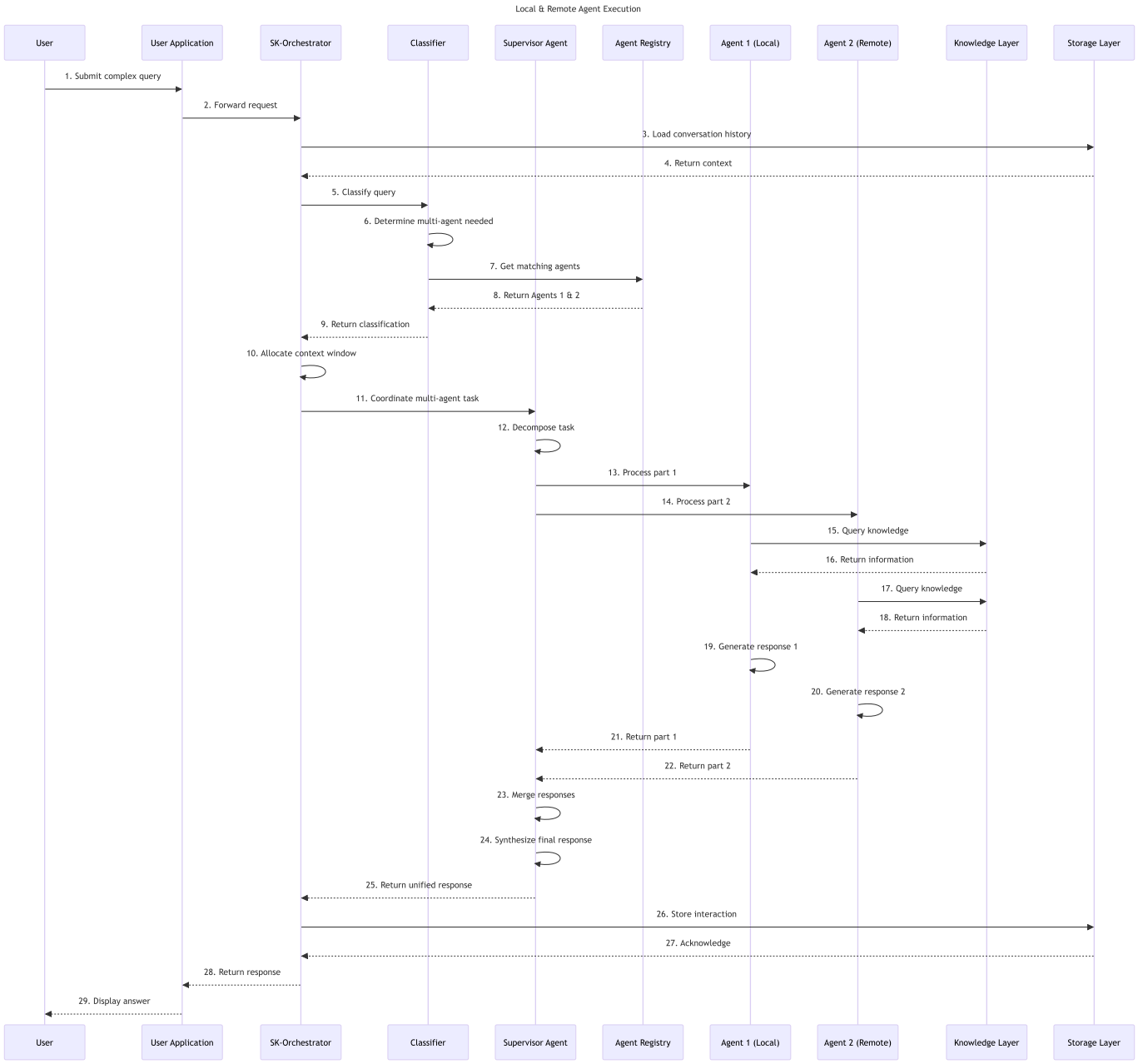
- Each "skill" encapsulates an agent function.
- Orchestrator chains skills with memory, planning, and goals.
- Benefit: Encourages modular and context-aware execution.



4. Local & Remote Agent Execution

Federated agent model with supervisor coordination.

- Local supervisor delegates tasks to local or remote agents.
- Secure channels maintain observability and traceability.
- Benefit: Enables scalability across networks or geographies.



5. Layered (Onion) Architecture

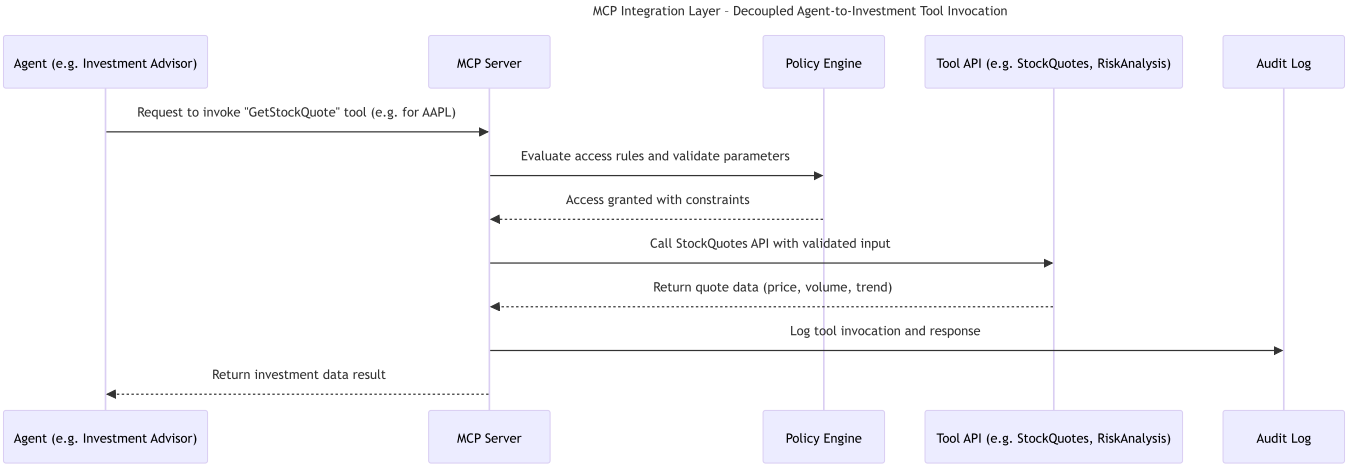
Separation of concerns by functional domain.

- Layers include Orchestration, Agent, Knowledge, Storage, Integration.
- Each layer has bounded responsibilities and APIs.
- Benefit: Improves maintainability, scalability, and testability.

6. MCP Integration Layer

Decoupled agent-to-tool invocation with governance.

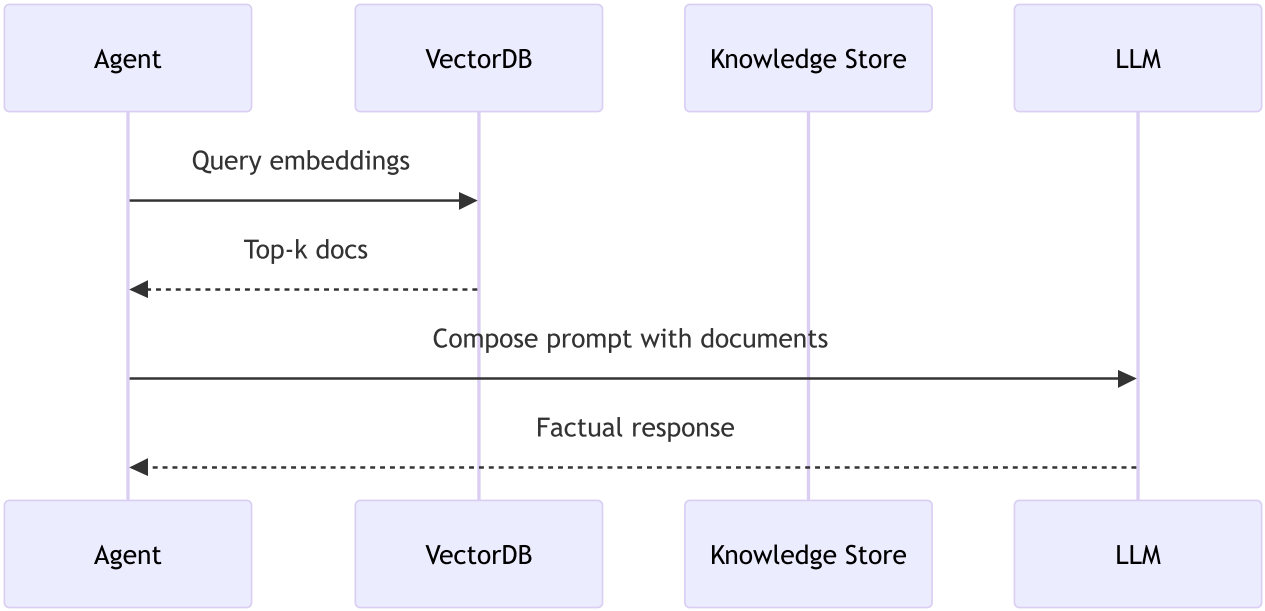
- MCP server abstracts tool APIs from agents.
- Policies control access, parameters, and invocation flow.
- Benefit: Adds auditability, policy enforcement, and centralized logic.



7. RAG (Retrieval-Augmented Generation)

Enhancing responses with contextual data from vector stores.

- Pre-indexed content stored in vector DBs.
- Orchestrator and agents query for grounding facts.
- Benefit: Improves factual accuracy and reduces hallucination.



8. Conversation-Aware Orchestration

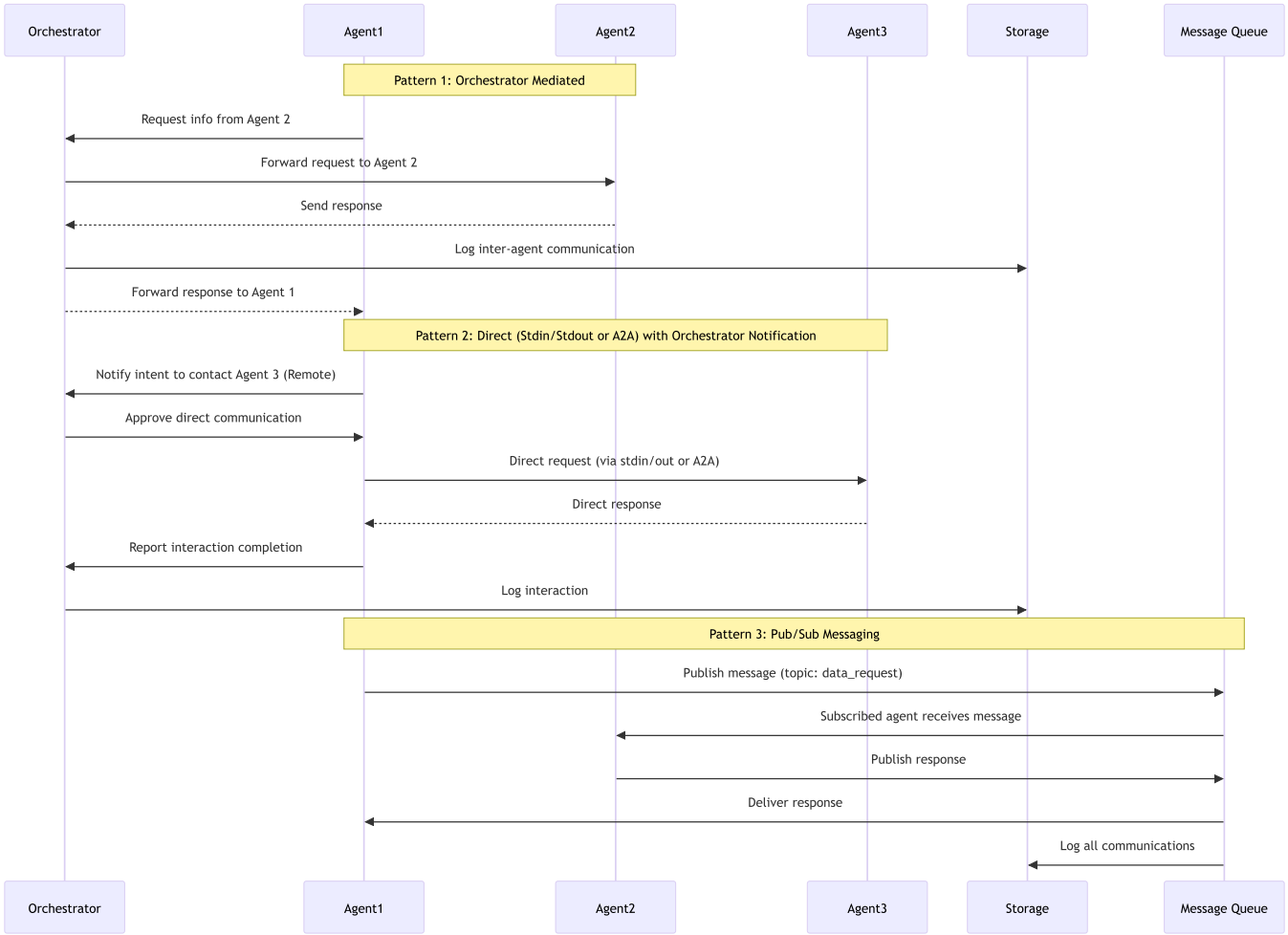
Adaptive behavior based on memory and context.

- Conversation history is stored and retrieved by orchestrator.
- Agents can use long-term and short-term memory cues.
- Benefit: Supports personalization, continuity, and context-awareness.

9. Agent-to-Agent Communication

Cooperative task delegation between agents.

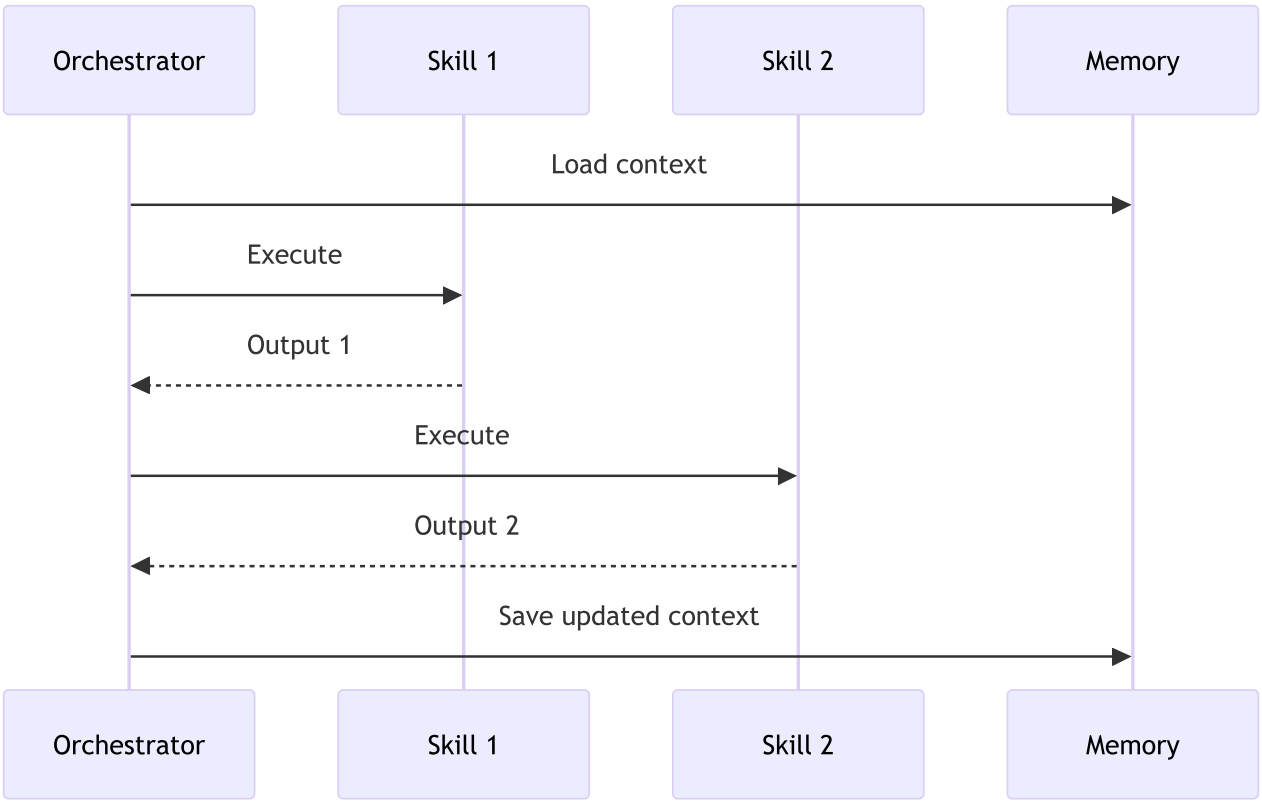
- Agents interact via orchestrator or directly through scoped protocols.
- Registry tracks active agents and routing preferences.
- Benefit: Supports delegation, specialization, and parallelization.



10. Skill Chaining with Planning Support

Goal-oriented execution via automatic chaining of capabilities.

- Planner creates execution path from available skills.
- Each skill is stateless, composable, and memory-aware.
- Benefit: Unlocks complex multi-step interactions.



Contributors

Last updated: 2025-05-14

Contributor
Ana Franco
Hao Luo
Fernando de Oliveira
Thiago Rotta
Vinicius Souza

Discuss this page

References

Last updated: 2025-06-30

Orchestration

- Semantic Kernel: Multi-agent Orchestration

Microsoft

- Microsoft Responsible AI Standard (v2)
- Monitoring Generative AI applications
- AI Red Teaming agent
- Continuously evaluate system behavior
- Prompt Shields
- Azure AI Content Safety
- Multi-Agent Solution Accelerator
- Azure AI Foundry Playground

Open AI

- Monitoring Generative AI applications
- A practical guide to build agents

Memory

- How Microsoft Copilot scales to millions of users with Azure Cosmos DB

Observability

- What is OpenTelemetry?

- Monitoring Generative AI applications
- OpenTelemetry – an open standard for collecting telemetry data.
- Open Telemetry Signals Concepts
- Observability defined by CNCF
- Observability in Semantic Kernel
- What are evaluators? - Azure AI Foundry
- Agent Evaluation in 2025: Complete Guide
- AI Agent Observability and Evaluation

Evaluation

- Evaluating Multi-Agent Systems
- LLM Evaluation
- Agent evaluators
- ToolCallAccuracyEvaluator (Microsoft / .NET)

Architecture design

- Monolith First by Martin Fowler
- Microservices for Greenfield?
- The Onion Architecture
- Microservices Guide
- Microservices Patterns
- Agentic Services

Communication Patterns & Protocols

Request-based

- Asynchronous Request-Reply pattern
- A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)
- Model Context Protocol (MCP)
- Unleash the Power of Model Context Protocol

- Agent-to-Agent Protocol (A2A)
- Google's Public A2A Agent Registry
- Semantic kernel A2A integration
- A2A Agent Card Specification
- Agent Network Protocol (ANP)
- Agent Communication Protocol (ACP)
- ACP Agent Detail Specification
- ACP Agent Detail Spec
- ACP Documentation on Agent Registry

Message-driven

- Choose between Azure messaging services
- A Distributed State of Mind: Event-Driven Multi-Agent Systems

Context Engineering

- OpenAI Function Calling Guide
- Semantic Kernel Decisions: Context-based function selection
- Introduction to semantic cache
- Semantic ranking in Azure AI Search
- Semantic Kernel: Make plugins AI-friendly
