Agentic AI patterns and workflows on AWS

AWS Prescriptive Guidance



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Prescriptive Guidance: Agentic AI patterns and workflows on AWS

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction **1**
Intended audience 1
Objectives 1
About this content series 2
Agent patterns **3**
Basic reasoning agents 4
Architecture 4
Description 5
Capabilities 6
Limitations 6
Common use cases 6
Implementation guidance 7
Summary 7
Architecture 7
Description 8
Capabilities 9
Common use cases 9
Implementation guidance 9
Summary 10
Tool-based agents for calling functions 10
Architecture 10
Description 11
Capabilities 12
Common use cases 12
Implementation guidance 12
Summary 13
Tool-based agents for servers 13
Architecture 13
Description 14
Capabilities 15
Common use cases 15
Implementation guidance 15
Summary 16
Computer-use agents 16

Architecture 16
Description 17
Capabilities 18
Common use cases 18
Implementation guidance 19
Summary 19
Coding agents 19
Architecture 19
Description 20
Capabilities 21
Common use cases 21
Implementation guidance 22
Summary 22
Speech and voice agents 22
Architecture 22
Description 23
Capabilities 24
Common use cases 24
Implementation guidance 24
Summary 25
Workflow orchestration agents 25
Architecture 25
Description 26
Capabilities 27
Common use cases 27
Implementation guidance 27
Summary 28
Memory-augmented agents 28
Architecture 28
Description 29
Capabilities 30
Common use cases 30
Implementing memory-augmented agents 30
Implementing memory-injected prompting 31
Summary 31
Simulation and test-bed agents 32

Architecture ... 32
Description ... 33
Capabilities ... 34
Common use cases ... 34
Implementation guidance ... 35
Summary ... 35
Observer and monitoring agents ... 36
Architecture ... 36
Description ... 36
Capabilities ... 37
Common use cases ... 37
Implementation guidance ... 38
Summary ... 38
Multi-agent collaboration ... 39
Description ... 40
Capabilities ... 41
Common use cases ... 41
Implementation guidance ... 41
Summary ... 42
Conclusion ... 42
Takeaways ... 42
LLM workflows ... 44
Overview of LLM-augmented cognition ... 44
Workflow for prompt chaining ... 45
Description ... 46
Capabilities ... 46
Common use cases ... 46
Workflow for routing ... 46
Capabilities ... 48
Common use cases ... 48
Workflow for parallelization ... 48
Capabilities ... 49
Common use cases ... 49
Workflow for orchestration ... 50
Capabilities ... 51
Common use cases ... 51

Workflow for evaluators and reflect-refine loops ... 51
Common use cases ... 52
Capabilities ... 52
Conclusion ... 52
Agentic workflow patterns ... 54
From event-driven to cognition-augmented systems ... 54
Event-driven architecture ... 54
Cognition-augmented workflows ... 55
Core insights ... 57
Prompt chaining saga patterns ... 57
Saga choreography ... 57
Prompt chaining pattern ... 58
Agent choreography ... 58
Takeaways ... 61
Routing dynamic dispatch patterns ... 61
Dynamic dispatch ... 62
LLM-based routing ... 63
Agent router ... 64
Takeaways ... 65
Parallelization and scatter-gather patterns ... 65
Scatter-gather ... 66
LLM-based parallelization (scatter-gather cognition) ... 68
Agent parallelization ... 68
Takeaways ... 69
Saga orchestration patterns ... 69
Event orchestration ... 70
Role-based agent system (orchestrator) ... 71
Supervisor ... 71
Takeaways ... 73
Evaluator reflect-refine loop patterns ... 73
Feedback control loop ... 74
Feedback control loop (evaluator) ... 75
Evaluator ... 75
Takeaways ... 76
Designing agentic workflows on AWS ... 77
Conclusion ... 77

Document history 79

Agentic AI patterns and workflows on AWS

Aaron Sempf and Andrew Hooker, Amazon Web Services

Organizations are adopting large language models (LLMs) and software agents to solve dynamic, multidomain problems using a new architectural discipline called agentic patterns. Agentic patterns are foundational blueprints and modular constructs that are used to design and orchestrate goal-oriented AI agents across many contexts.

Intended audience

This guide is intended for architects, developers, and product leaders who want to build intelligent applications that go beyond static logic, symbolic logic, and deterministic automation.

Objectives

This guide provides a design framework and implementation approach for AI agent systems that operate autonomously while remaining controllable and aligned with your goals. It connects event-driven architectural patterns with various agentic alternatives, demonstrating how to build production-grade agent systems using cloud-native architectures. The following subjects are discussed in this guide:

- **Agent patterns –** Agent patterns are reusable design templates that describe the structure and behavior of individual agents. This includes reasoning agents, retrieval-augmented agents, coding agents, voice interfaces, workflow orchestrators, and collaborative multi-agent systems. Each pattern illustrates how agents perceive, reason, act, and learn, mapped to AWS services.
- **LLM workflows –** Workflows focus on how agents use LLMs for reasoning. They explore prompting strategies and planning mechanisms, and outline how LLMs are used not only to generate text but also to drive structured, interpretable, and reliable behaviors within an agent loop.
- **Agentic workflow patterns –** Workflow patterns describe how multiple agents, tools, and environments interact to form autonomous systems. This includes patterns for task orchestration, subagent delegation, event-based coordination, observability, and control. These aspects promote scalable, composable, and auditable AI architectures.

About this content series

This guide is part of a set of publications that provide architectural blueprints and technical guidance for building AI-driven software agents on AWS. The AWS Prescriptive Guidance series includes the following guides:

- Operationalizing agentic AI on AWS
- Foundations of agentic AI on AWS
- *Agentic AI patterns and workflows on AWS* (this guide)
- Agentic AI frameworks, protocols, and tools on AWS
- Building serverless architectures for agentic AI on AWS
- Building multi-tenant architectures for agentic AI on AWS

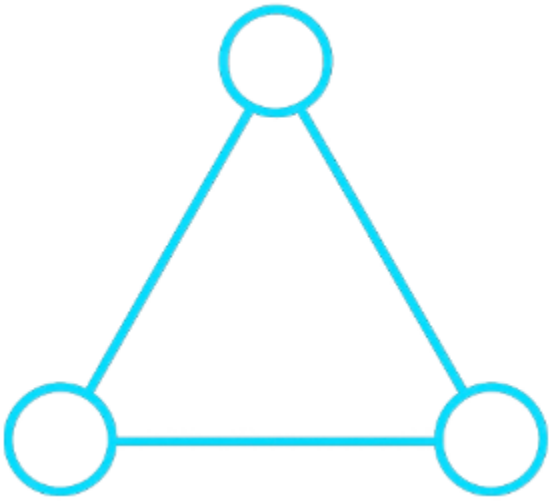
For more information about this content series, see Agentic AI.

Agent patterns

Agent patterns are reusable, composable building blocks that can be tailored to specific domains, use cases, and levels of complexity. Agentic systems differ, however, from traditional applications. At the heart of all AI agent designs is a conceptual model anchored in the following three foundational principles:

- **Asynchronous –** Agents operate in loosely coupled, event-rich environments
- **Autonomy –** Agents act independently, without human or external control
- **Agency –** Agents act with purpose, on behalf of a user or system, toward specific goals

The triangle in the following diagram represents the core building blocks of a software agent: perception, reason, and action. This enables an agentic system to observe, make decisions, and act within its environment.



By design, agentic patterns provide a modular design language for building AI systems, which means they're accessible, operational, extensible, and production ready. Designing these systems requires careful attention to the following three interrelated dimensions, which are further discussed later in this guide.

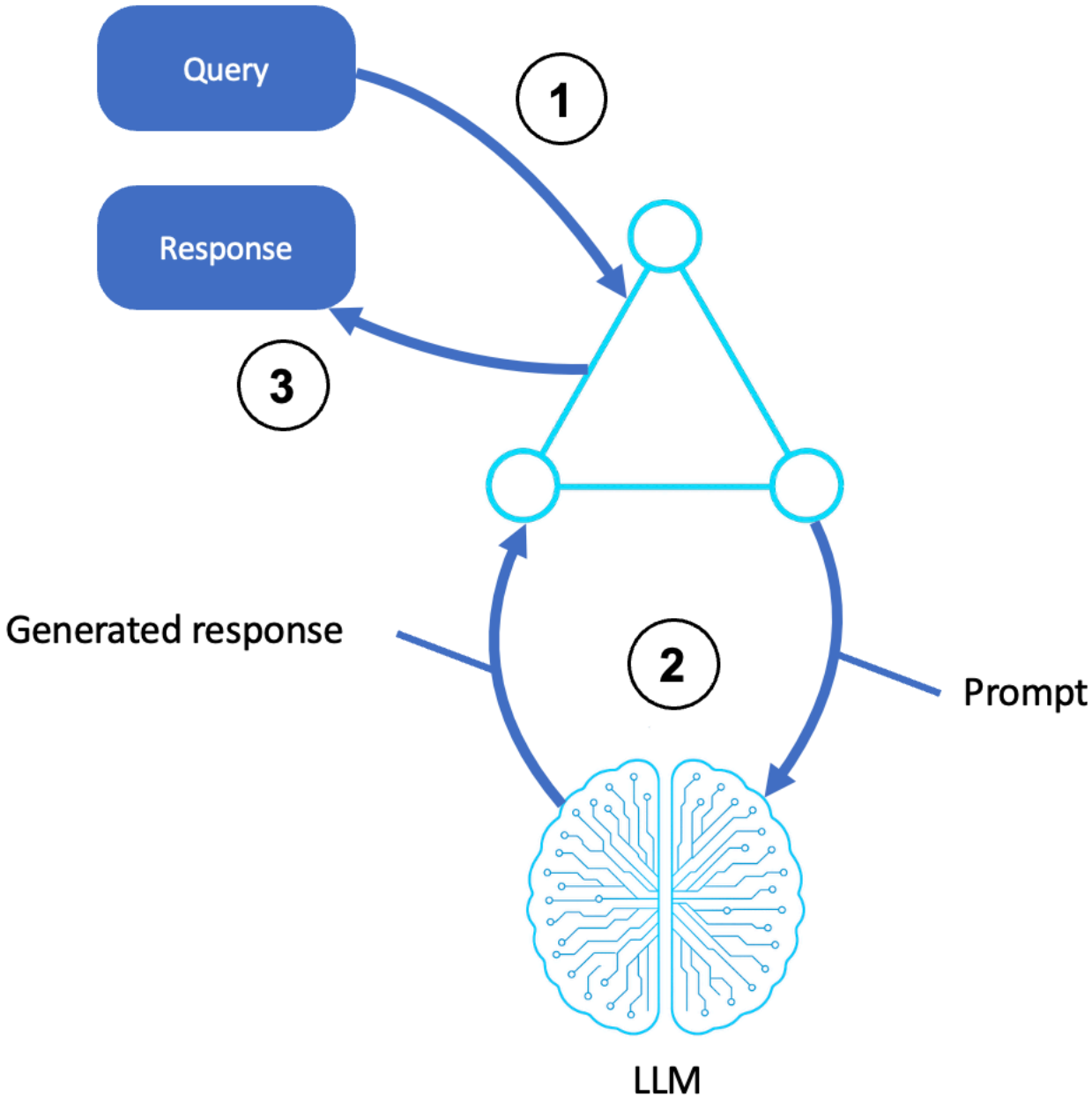
Basic reasoning agents

A basic reasoning agent is the simplest form of agentic AI that performs logical inference or decision-making in response to a query. It accepts input from a user or system and processes queries and generates responses using structured prompts.

This pattern is useful for tasks that require single-step reasoning, classification, or summarization based on a given context. It doesn't use memory, tools, or state management, which makes it stateless, lightweight, and highly composable across large workflows.

Architecture

The flow of a basic reasoning agent is shown in the following diagram:



Description

1. Receives an input
   - A user, system, or upstream agent submits a query or instruction.
   - The input is handed off to the agent shell or orchestration layer.
   - This step includes any preprocessing, prompt templating, and goals identification.
2. Invokes the LLM

- The agent transforms the query into a structured prompt and sends it to an LLM (for example, through Amazon Bedrock).
- The LLM generates a response based on the prompt using pretrained knowledge and context.
- The generated output may include reasoning steps (chain-of-thought), final answers, or ranked options.

3. Returns a response
   - The generated output is relayed to the agent's interface.
   - This may include formatting, postprocessing, or an API response.

Capabilities

- Supports natural language or structured input
- Uses prompt engineering to guide behavior
- Stateless and scalable
- Can be embedded into UI, CLI, APIs, and pipelines

Limitations

- No memory or historical awareness
- No interaction with external tools or data sources
- Limited to what the LLM knows at the time of inference

Common use cases

- Conversational questions and answers
- Policy explanations and summaries
- Guidance for making decisions
- Lightweight and automated chatbot flows
- Classification, labeling, and scoring

Implementation guidance

You can use the following tools and services to create a basic reasoning agent:

- Amazon Bedrock for LLM invocation (Anthropic, AI21, Meta)
- Amazon API Gateway or AWS Lambda to expose it as a stateless microservice
- Prompt templates stored in Parameter Store, AWS Secrets Manager, or as code

Summary

The basic reasoning agent is foundational because of its simple structure. It has core capabilities that turn goals into reasoning paths that lead to intelligent outputs. This pattern is often a starting point for advanced patterns, such as tool-based agents and agents that use retrieval-augmented generation (RAG). It's also a reliable and modular component of large workflows.

Agent RAG

Retrieval-augmented generation (RAG) is a technique that combines information retrieval with text generation to create accurate and contextual responses. RAG enables agents to retrieve relevant external information before engaging the LLM. It extends an agent's effective memory and reasoning accuracy by grounding its decisions in up-to-date, factual, or domain-specific information. In contrast to stateless LLMs that rely solely on pretrained weights, RAG has an external knowledge search layer that dynamically enhances prompts with context.

Architecture

The logic of the RAG pattern is illustrated in the following diagram:

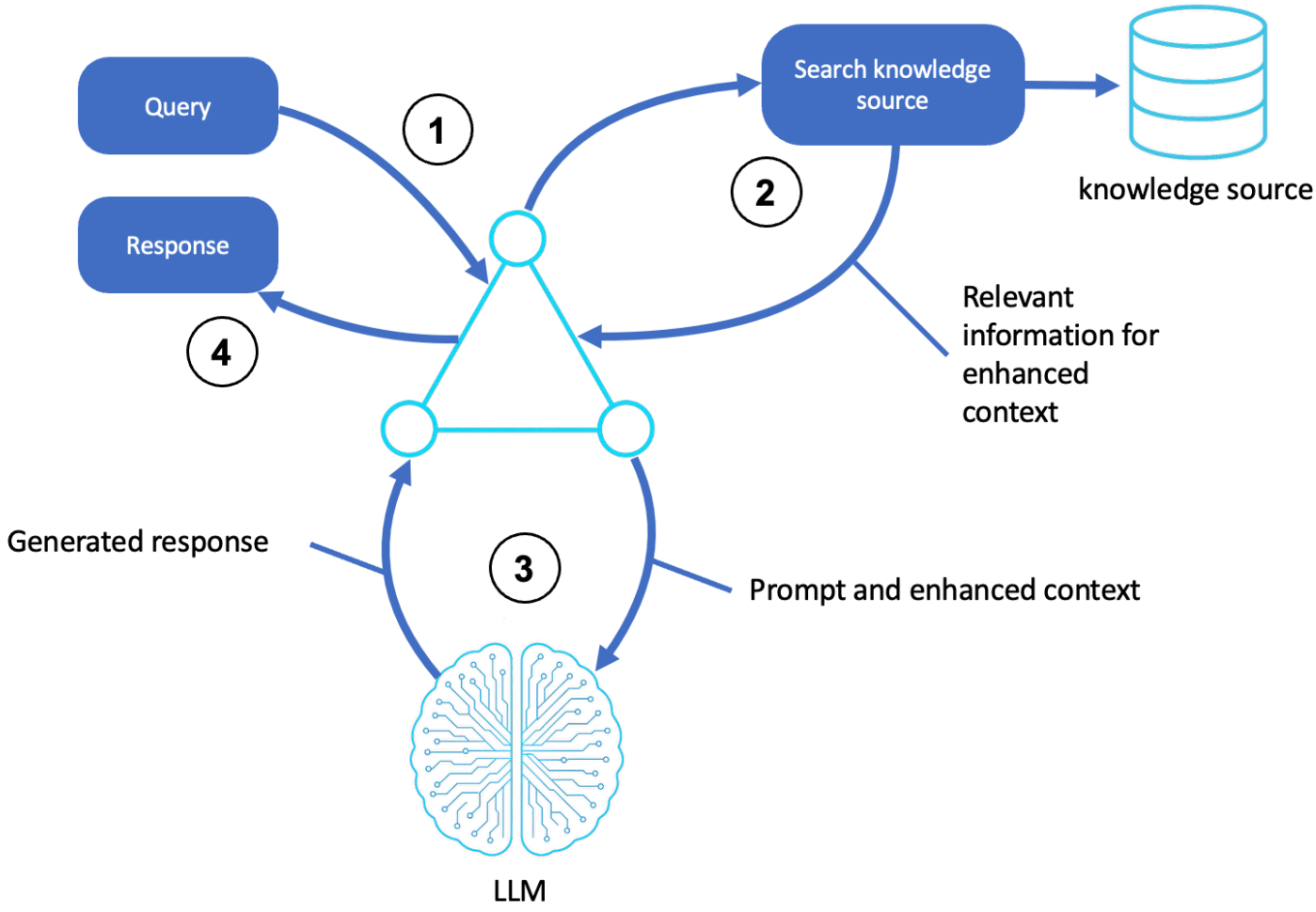


Figure 3. Agent RAG

Description

1. Receives a query
   - A user or upstream system submits a query or goal to the agent.
   - The agent shell accepts the request and formats it as a prompt for reasoning.
2. Searches an external source
   - The agent identifies concepts and intent from the query.
   - It queries a knowledge source, such as a vector store, database, or document index using semantic search or keyword matching.
   - The most relevant passages, documents, or entities are retrieved for use in the next step.
3. Generates a contextual response

- The agent augments the prompt with the retrieved information, forming a context-enhanced input for the LLM.
- The LLM processes any inputs using generative reasoning (for example, chain-of-thought or reflection) to produce an accurate response.

4. Returns the final output
   - The agent prepares the output by wrapping it in any communication headers or required formatting and then returns it to the user or calling system.
   - (Optional) The retrieved documents and LLM output may be logged, scored, and stored in memory for future queries.

Capabilities

- Fact-grounded output even in long-tail or enterprise-specific domains
- Memory extension without fine-tuning the model
- Dynamic context based on each query and user state
- Fully compatible with vector databases, semantic indexes, and metadata filtering

Common use cases

- Enterprise knowledge assistants
- Regulatory compliance bots
- Customer support copilots
- Search-enhanced chatbots
- Developer documentation agents

Implementation guidance

Use the following tools and services to create an agent that uses RAG:

- Amazon Bedrock for LLM invocation
- Amazon Kendra, OpenSearch, or Amazon Aurora for documentation or a structured data search
- Amazon Simple Storage Service (Amazon S3) for document storage
- AWS Lambda to orchestrate search, prompt, and LLM inference

- Knowledge-based integrations with agents (by using memory plugins, semantic retrievers, or Amazon Bedrock)

Summary

Agent RAG connects static model reasoning to dynamic, real-world intelligence. It equips agents with the ability to look up what they don't know, synthesize answers from retrieved knowledge, and produce high-confidence, auditable responses.

RAG patterns are a foundation for building intelligent agents that scale knowledge access without retraining. It is often a precursor to more complex orchestration patterns involving tool use, planning, and long-term memory.

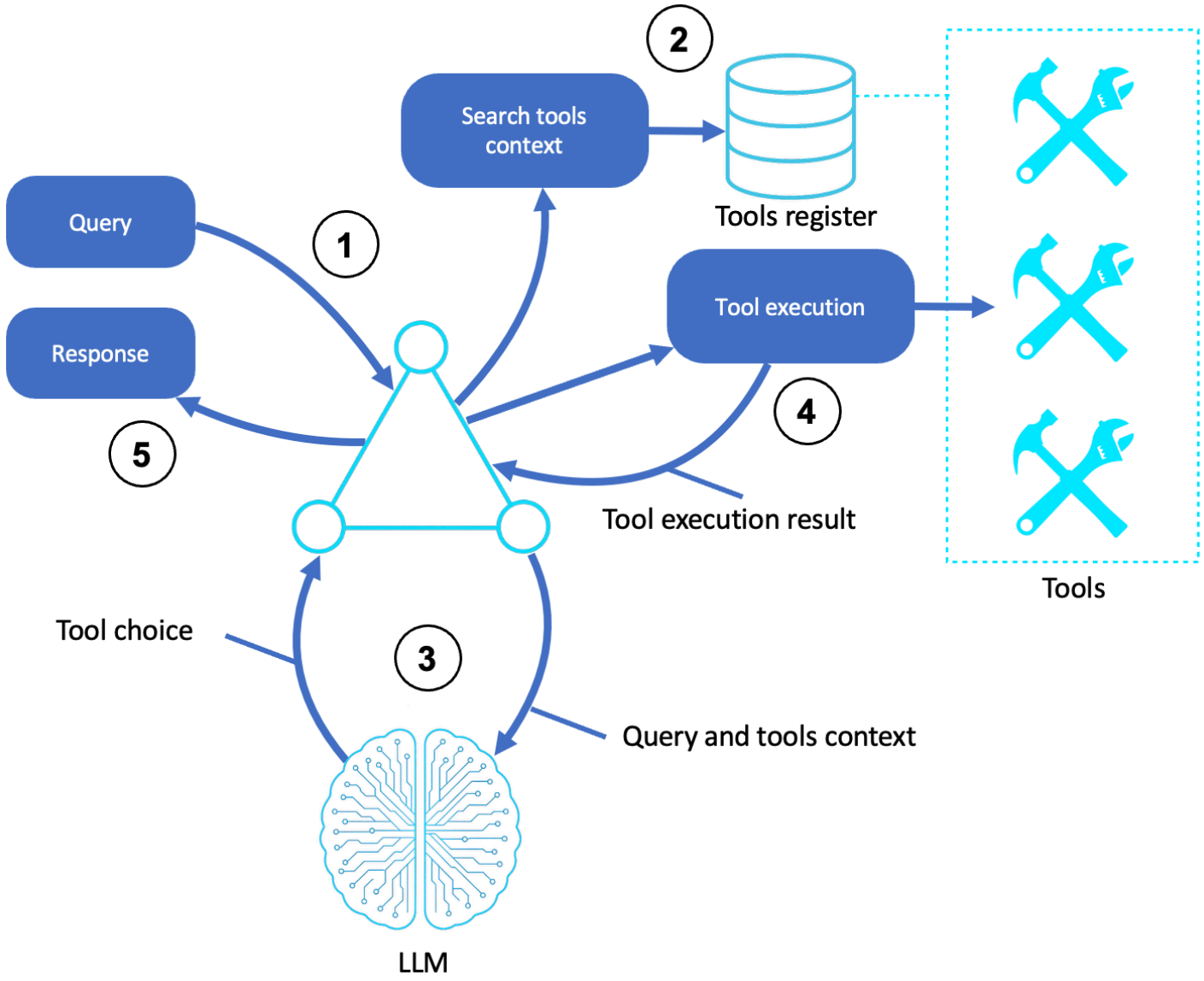
Tool-based agents for calling functions

Tool-based agents extend the capabilities of reasoning agents by invoking external functions or APIs to complete tasks that go beyond language-only reasoning. This pattern uses an LLM to decide which tool to use and then generates call arguments and incorporates a tool's output into its reasoning loop.

This pattern enables agents to act rather than just providing responses. The tool interface represents any callable capability, ranging from arithmetic calculations and database lookups to external APIs and cloud services.

Architecture

A tool-based agent for calling functions is shown in the following diagram:



Description

1. Receives query
   - The agent receives a natural-language query or task from the user or calling system.
2. Searches for tools
   - The agent uses internal metadata or a tool registry to search for available tools, schemas, and relevant capabilities.
3. Selects and invokes tools
   - The LLM receives the query and tool metadata (for example, function names, input types, and descriptions) in its prompt.

- It chooses the most relevant tool, constructs input arguments, and returns a structured function call.

4. Runs the chosen tool
   - The agent shell or tool runner executes the selected function and returns the result (for example, an API output, database value, or computation).
5. Returns a response
   - The LLM passes results to the agent, either directly or as part of an updated prompt. It then returns a natural-language result.

Capabilities

- Dynamic tool selection based on task context
- Schema-based prompting (OpenAPI, JSON schema, AWS function interface)
- Results interpretation and chaining of outputs into reasoning
- Stateless or session-aware operations

Common use cases

- Virtual assistants with external data access
- Financial calculators and estimators
- API-based knowledge workers
- LLMs that invoke AWS Lambda, Amazon SageMaker endpoints, and SaaS services

Implementation guidance

Use the following to create tool-based agents for calling functions:

- Amazon Bedrock with function-calling support (Anthropic Claude)
- AWS Lambda as a tool-execution backend
- Amazon API Gateway or AWS Step Functions for tool orchestration
- Amazon DynamoDB or Amazon Relational Database Service (Amazon RDS) for context-aware tool metadata
- Amazon EventBridge pipelines or AWS Step Functions that map states to route outputs

Summary

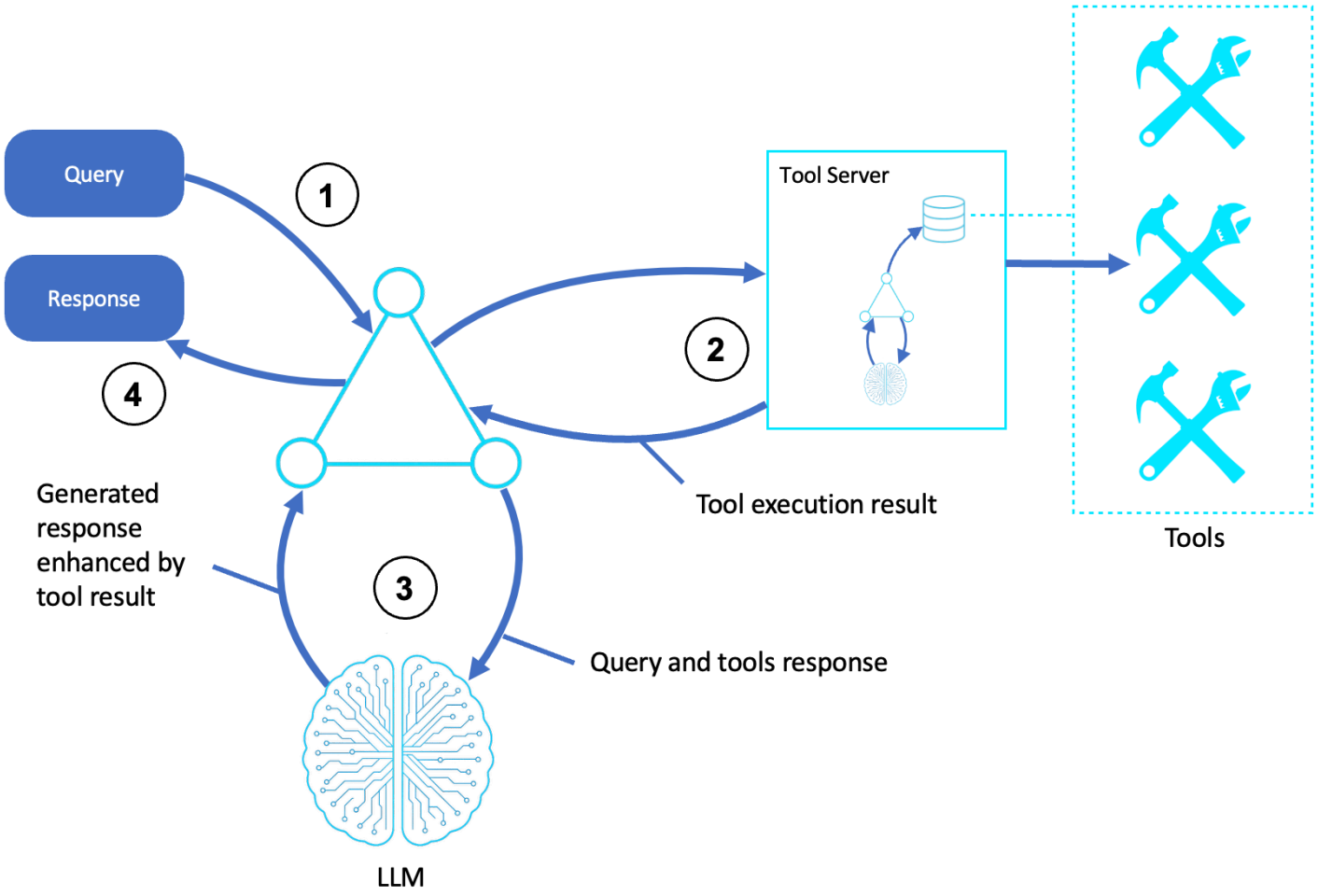
Tool-based function-calling agents represent a shift from understanding language to performing actions. These agents invoke dynamic, context-aware tools while maintaining LLM reasoning, transforming passive assistants into systems that complete tasks, access services, and integrate business operations. This pattern is an important component of agentic AI in enterprise settings, especially when combined with declarative schemas, authorization frameworks, and multi-agent systems.

Tool-based agents for servers

Tool-based agents for servers enhance function-calling agents by delegating tool execution to an external server that has a dedicated runtime environment for tools, scripts, and composite agents. Unlike inline function calls that the agent loop selects and invokes, server-based agents outsource the logic and execution pipeline to other agents or systems. This provides advanced capabilities like multitool chaining, isolated execution, and specialized reasoning. Tool servers are ideal for complex, stateful, or resource-intensive actions where the tools themselves may involve separate AI models, business rules, or environments.

Architecture

The following is a pattern for tool-based agents for servers:



Description

1. Receives query
   - A user or system submits a request to the agent shell.
   - The agent interprets the query and prepares to dispatch it to a tool server.
2. Runs tool server processes
   - The agent sends the task, along with structured parameters, to a tool server.
   - The tool server may then:
     - Run scripts or logic in dedicated compute systems (for example, AWS Lambda, containers, or Amazon SageMaker)
     - Use its own subagent with LLM reasoning to select and run tools
     - Manage dependencies, retries, or multistep execution flows
     - Output results to the primary agent when the task is complete

3. Uses LLM reasoning with tool output
   - The agent invokes an LLM, passing the original query and the tool server result as part of the prompt.
   - The LLM synthesizes a response that incorporates the newly acquired information.
4. Returns a response
   - The agent returns a natural-language or structured response to the user or calling system.
   - (Optional) Results may be stored in memory or audit logs.

Capabilities

- Tools are invoked outside of the primary agent execution loop
- Tool execution may involve LLM calls, logic chains, or subagents
- Agent acts as a controller or dispatcher, not just a tool wrapper
- Enables composability, scalability, and isolation of logic

Common use cases

- Orchestrating model chains (for example, by combining LLM, vision, and code)
- AI-driven automation pipelines
- DevOps assistant agents with script runners
- Complex financial computation, simulation, or optimization agents
- Multimodal tools (for example, by combining audio, documentation, and action)

Implementation guidance

You can build this pattern using the following AWS services:

- Amazon Bedrock (agent host and LLM inference)
- AWS Lambda, Amazon ECS, AWS Fargate, or Amazon SageMaker endpoints as the tool server runtime
- Amazon API Gateway or AWS App Runner to expose tool server APIs
- Amazon EventBridge for decoupled agent-to-tool messaging
- AWS Step Functions or AWS AppFabric for composing multi-agent logic on the tool server

Summary

Tool-based agents that use servers are highly modular and scalable. They decouple decision logic from execution, which allows the primary agent to remain lightweight while offloading complex or sensitive actions to other systems. This is important for enterprise-grade agentic AI, especially in environments that require governance, observability, isolation, dynamic composition, or any combination thereof.

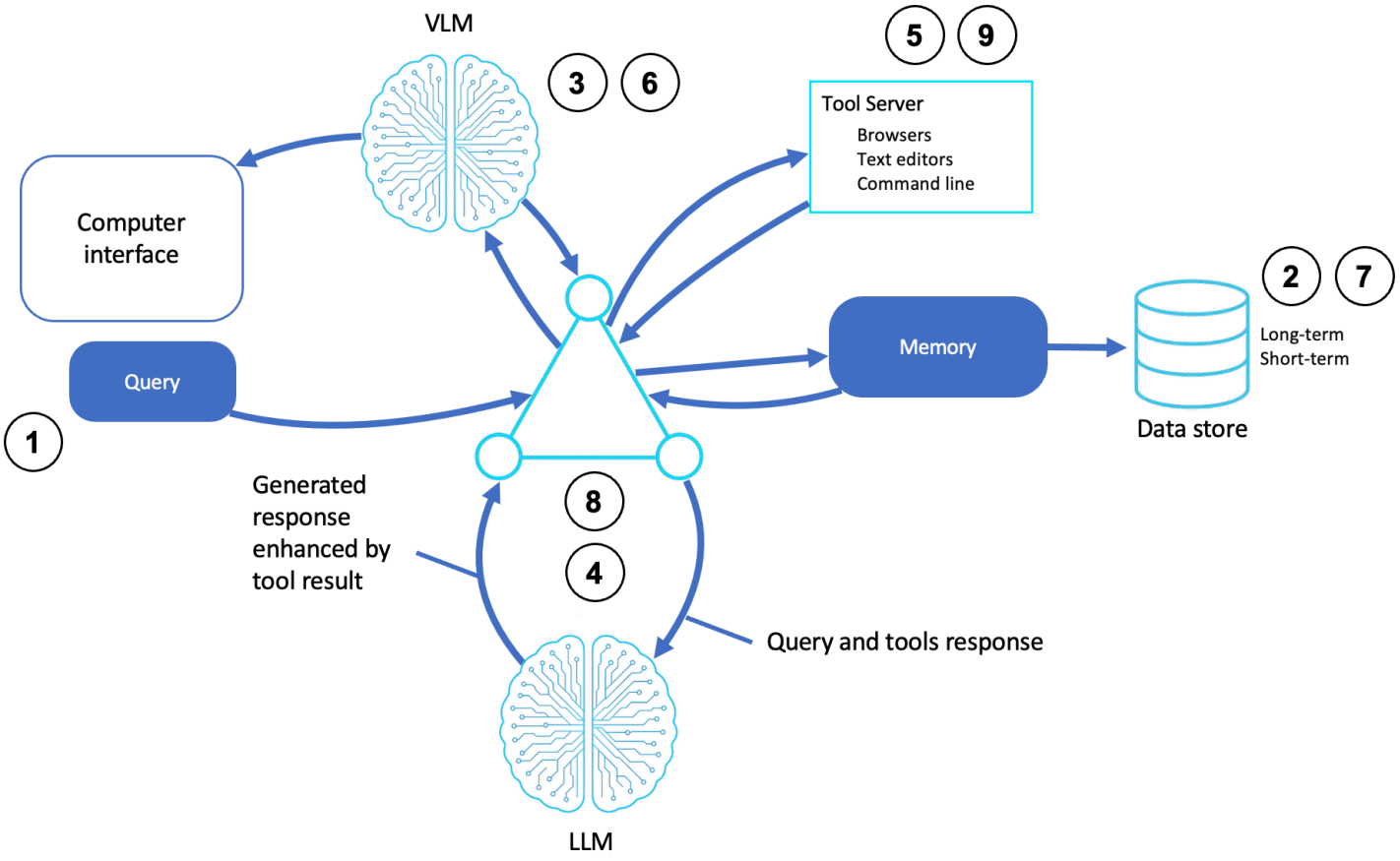
Computer-use agents

Computer-use agents can simulate or control digital environments like browsers, terminals, file systems, and applications. These agents interpret user intent, interact with visual and text interfaces, and perform goal-directed actions by combining LLM reasoning, visual language models (VLMs), and tool servers that execute commands or simulate input events.

This pattern is important for practical AI automations, where the agent functions not just as an assistant but also as a proxy that performs actions as a human would, often by using the same tools and environments.

Architecture

A computer-use agent pattern is shown in the following diagram:



Description

1. Receives a query
   - A task or request is provided through a UI, API, or natural language interface.
2. Accesses memory
   - The agent retrieves short-term and long-term memory to recall past commands, goals, and system states.
3. Analyzes the visual context
   - A VLM observes the computer screen, system state, or UI elements to understand a given context and identify actionable items.
4. Reasons through an LLM
   - The LLM combines the query, memory state, tool, and server response to determine the next action.
5. Interacts with tool server
   - The agent invokes tools that are hosted on a server, which may include the following:

- Browsers (for example, headless Chrome) and shell environments
- Text and code editors
- Custom script interfaces

6. Updates visual inputs
   - If the system UI changes or further observation is needed, the VLM may reanalyze the screen state or text buffers.
7. Updates memory
   - New insights, system states, or user feedback are written to short-term and long-term memory.
8. Formulates final decisions and explanations
   - The LLM synthesizes results or recommends actions based on the query and tool output.
9. Returns a response
   - The agent returns results to the interface (for example, a completed task, confirmation, or generated content).

Capabilities

- Multimodal reasoning with visual and textual inputs
- Control over applications through simulated or API-driven inputs
- Memory management for persistent state
- Autonomy in sequence execution (multistep flows)

Common use cases

- AI developers that write and run code in IDEs
- Computer-use agents for repetitive digital workflows
- Simulated users for software testing and quality assurance
- Accessibility agents for navigating UIs through voice or high-level instructions
- Smart robotic process automation (RPA) that's enhanced with reasoning

Implementation guidance

- You can build this pattern using the following AWS services:
- Amazon Bedrock for LLM-based planning and reasoning
- Amazon Elastic Compute Cloud (Amazon EC2), AWS Lambda, or Amazon SageMaker notebooks to run tool servers with simulated UI environments
- Amazon Simple Storage Service (Amazon S3) or Amazon DynamoDB for memory persistence
- Amazon Rekognition (or custom models) for UI image analysis in hybrid scenarios
- Amazon CloudWatch Logs or AWS X-Ray for observability and audit trails

Summary

Computer-use agents act as autonomous digital operators, bridging the gap between human-computer interactions and AI-driven actions. By incorporating memory, tool orchestration, and VLMs, these agents can adaptively interact with systems designed for humans, execute actions, update files, navigate menus, and generate responses.

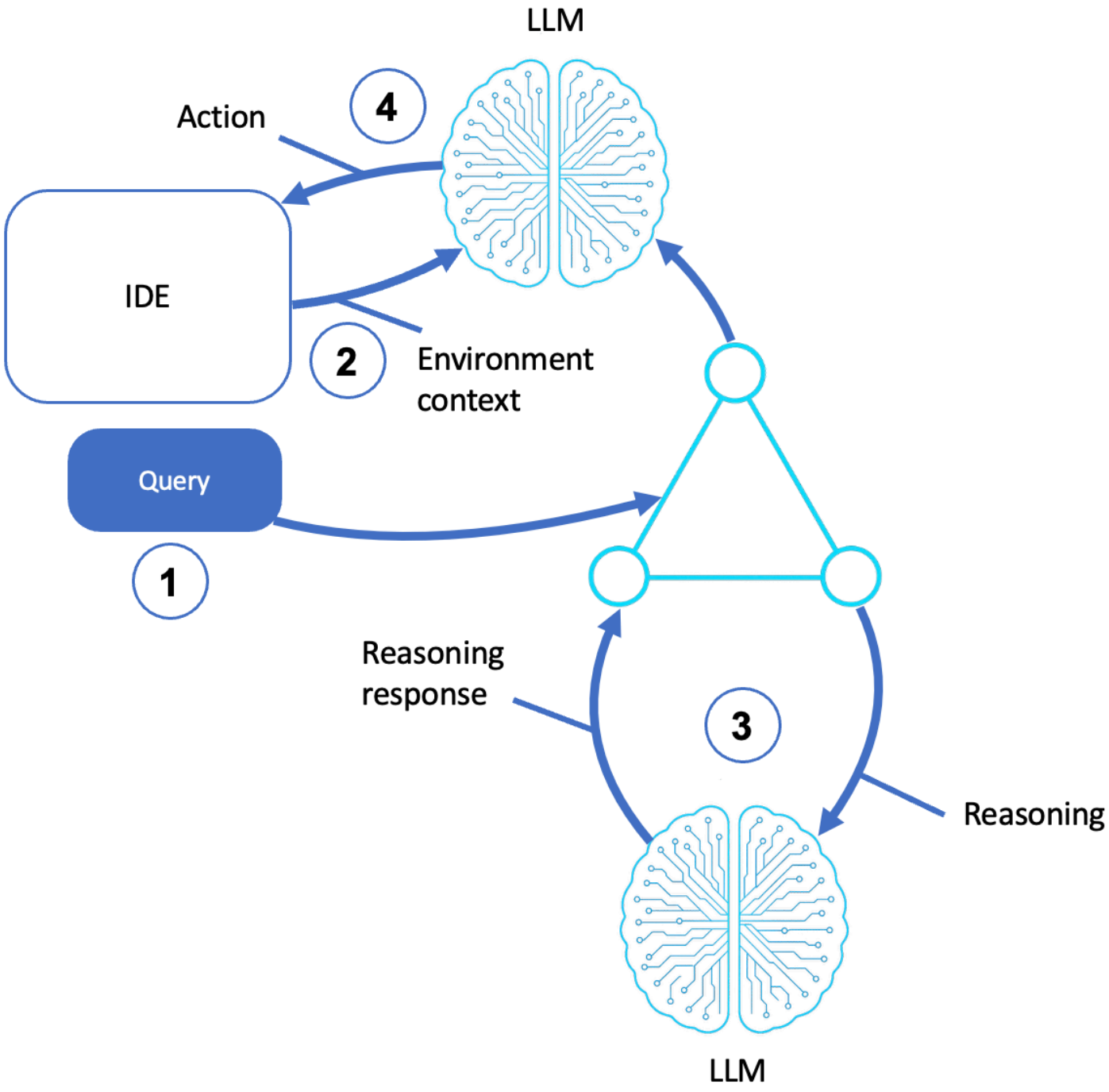
Coding agents

Coding agents can reason about programming tasks, generate or modify code, and interact with developer environments, such as IDEs and CLIs. These agents combine natural-language understanding with structured reasoning to assist, augment, and automate software development, ranging from function generation to bug fixing and test authoring.

Unlike autocomplete tools, coding agents actively interpret user goals, query the development environment for context (for example, it opens files and traces errors), identify requirements, and then propose and perform actions.

Architecture

A coding-agent pattern is shown in the following diagram:



Description

1. Receives query
   - The user provides natural-language instructions through a command palette, chat window, or CLI (for example, "Add logging to this function" or "Refactor for readability").
2. Extracts environment context

- The agent gathers context from the IDE, including active files, cursor position, code snippets and symbol tables.
- It outputs error messages, test results, and outputs from other agents.

3. LLM reasoning
   - The agent sends a prompt, including the query and environmental context, to an LLM.
     - The LLM performs a reasoning pass to determine the following:
     - What needs to change
     - How to generate a solution
     - Any refactoring, rewriting, or coding steps
4. Executes actions
   - The LLM returns the output to the agent and imports it into the IDE or runtime environment.
   - This may include inserting or modifying code, generating comments or documentation, and triggering downstream build, test, and linting tasks.

Capabilities

- High-contextual awareness (for example, IDE state, cursor, and syntax tree)
- Iterative reasoning of goals and feedback
- Optional code planning and action separation (for example, first reason and then act)
- Works in synchronous or asynchronous developer workflows

Common use cases

- Code generation from task descriptions
- Code refactoring and optimization
- Test-case generation and validation
- Error explanations and debugging
- Documentation assistants
- Paired programming copilots

Implementation guidance

- You can build this pattern using the following tools and AWS services:
- Amazon Bedrock for LLM-driven generation and reasoning
- Amazon Q Developer for coding suggestions and completions
- AWS Lambda or Amazon Elastic Container Service (Amazon ECS) for running and testing sandbox environments
- AWS Cloud9, VS Code extensions, or custom IDE integrations to host and evaluate context
- Amazon Simple Storage Service (Amazon S3) for storing intermediate prompts, responses, and revision history

Summary

Coding agents are new AI-powered development tools that are capable of interpreting natural language, analyzing context, generating multistep code changes, and integrating with the software development lifecycle.

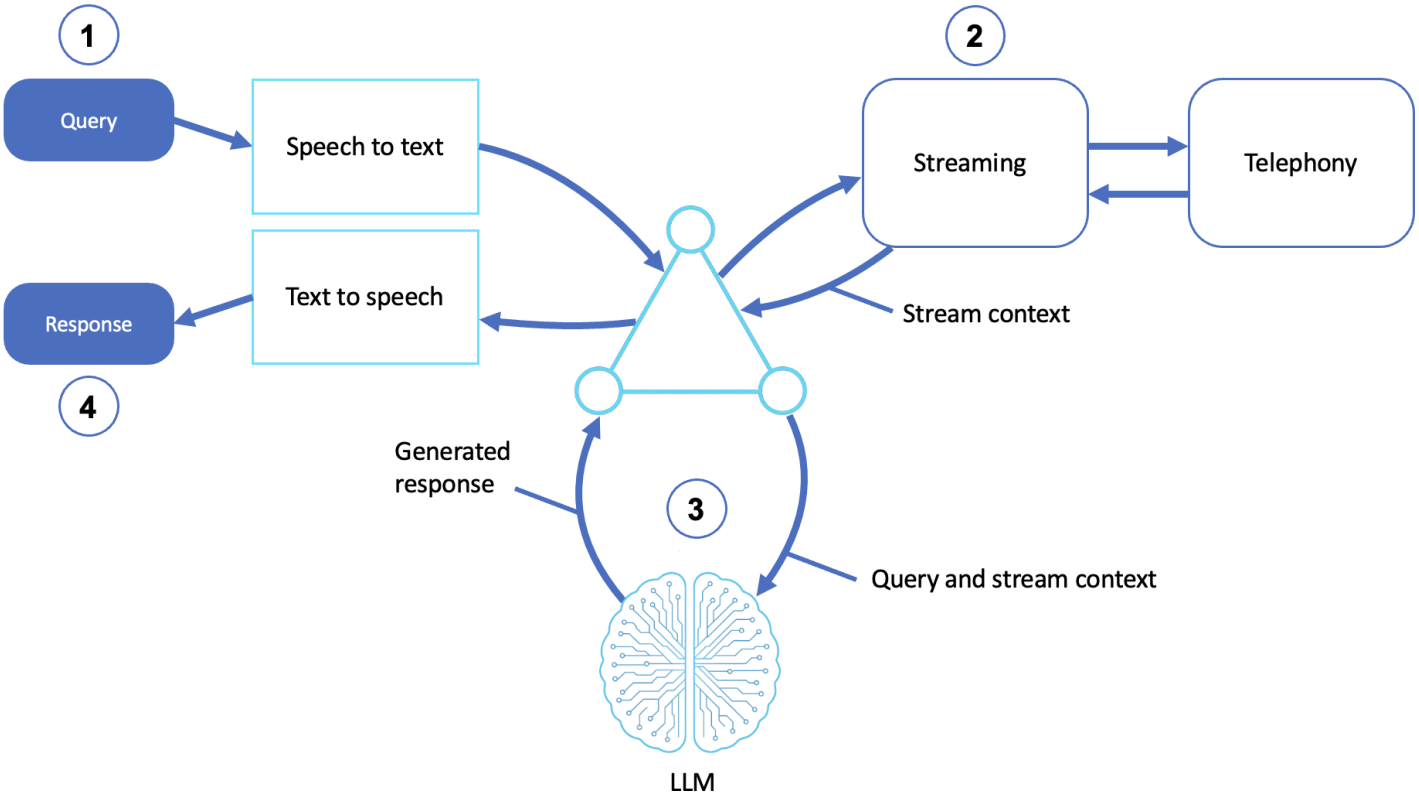
Speech and voice agents

Speech and voice agents interact with users through spoken dialogue. These agents integrate speech recognition, natural-language understanding, and speech synthesis to enable conversational AI across telephony, mobile, web, and embedded platforms.

Voice agents are particularly effective in hands-free, real-time, or accessibility-driven environments. By combining streaming interfaces with LLM-powered reasoning, they facilitate rich, dynamic interactions that feel natural to users.

Architecture

A speech and voice agent is shown in the following diagram:



Description

1. Receives a voice query
   - The user voices a request to a phone, microphone, or embedded system.
   - A speech-to-text (STT) module converts the audio to text.
2. Integrates streaming and telephony context
   - The agent uses a streaming interface to manage audio I/O in real time.
   - If it's deployed in a contact center or telecom context, telephony integration handles session routing, dual-tone multi-frequency (DTMF) input, and media transport.

Note: DTMF refers to the tones generated when you press buttons on a telephone keypad. In the context of streaming and telephony context integration within voice agents, DTMF is used as a signal input mechanism during a phone call, especially in interactive voice response (IVR) systems. DTMF inputs enable the agent to:

- Recognize menu selections (for example, "Press 1 for billing. Press 2 for support.")
- Collect numeric inputs (for example, account numbers, PINs, and confirmation numbers)

- Trigger workflows or state transitions in call flows
- Revert from speech to touch-tone when necessary

1. Reasons through LLM stream context
   - The query is sent to the agent, which passes it, along with any session metadata (for example, caller ID, prior context), to an LLM.
   - The LLM generates a response, possibly using a chain-of-thought strategy or multiturn memory if the interaction is ongoing.
2. Returns a voice response
   - The agent converts its response to speech using text-to-speech (TTS).
   - It returns audio to the user through a voice channel.

Capabilities

- Real-time speech understanding and generation
- Multilingual I/O with STT and TTS support
- Integration with telephony or streaming APIs
- Session awareness and memory handoff between turns

Common use cases

- Conversational IVR systems
- Virtual receptionists and appointment schedulers
- Voice-driven helpdesk agents
- Wearable voice assistants
- Voice interfaces for smart homes and accessibility tools

Implementation guidance

You can build this pattern using the following tools and AWS services:

- Amazon Lex V2 or Amazon Transcribe for STT
- Amazon Polly for TTS

- Amazon Chime SDK, Amazon Connect Customer, or Amazon Interactive Video Service (Amazon IVS) for streaming and telephony
- Amazon Bedrock for reasoning with Anthropic, AI21, or other foundation models
- AWS Lambda to connect STT, LLM, TTS, and session context

(Optional) Additional enhancements may include the following:

- Amazon Kendra or OpenSearch for context-aware RAG
- Amazon DynamoDB for session memory
- Amazon CloudWatch Logs and AWS X-Ray for traceability

Summary

Speech and voice agents are intelligent systems that interact through natural conversations. By integrating speech interfaces with LLM reasoning and real-time streaming infrastructure, voice agents enable seamless, accessible, and scalable interactions.

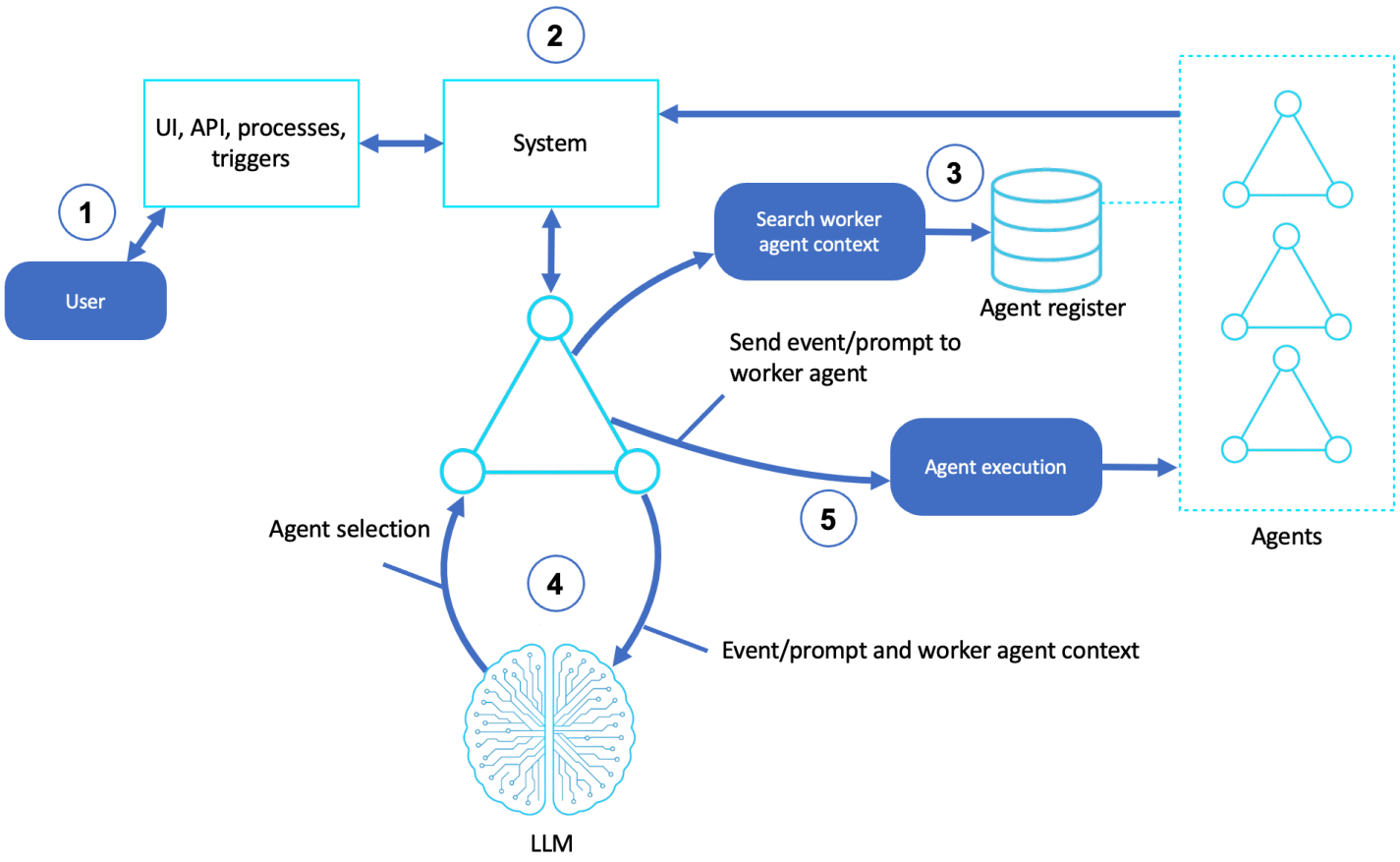
Workflow orchestration agents

Workflow orchestration agents manage and coordinate multistep tasks, processes, and services across distributed systems. Rather than reasoning and acting in isolation, these agents delegate work to subagents or other systems, maintain execution context, and adapt based on intermediate results.

These agents are a fundamental part of automation flows. They are particularly useful when handling long-running tasks, multi-agent compositions, and cross-domain integrations where various agents and tools must be called in sequence or conditionally.

Architecture

A workflow orchestration agent is shown in the following diagram:



Description

1. Receives user input
   - A user (or external trigger) initiates a task through a UI, API, or system event.
2. Handles system events
   - A system component receives the request and emits an event or command that requires orchestration.
3. Retrieves context
   - The workflow agent queries knowledge bases and agent registries to find the right worker agent for the task based on metadata, domain, and prior success rate.
4. Selects an LLM agent
   - An LLM helps to select the best agent or workflow plan by analyzing the task description and available options.
   - It may also formulate task-specific prompts to send to a selected agent.
5. Delegates and executes

- The chosen worker agent receives the event or prompt and begins running commands.
- It can track execution state, retry on failure, and pass intermediate results to the next agent in the sequence.

Capabilities

- Agent composition (for example, supervisors, collaborator agents, and tools)
- Event-driven or scheduled execution
- Memory and state tracking over time
- Hierarchical or parallel task orchestration (synchronous compared with asynchronous workflows)
- Dynamic agent selection and chaining

Common use cases

- Multistep automation (for example, data ingestion and reporting)
- Customer service routing and escalation (for example, agent-as-coordinator)
- AI agents coordinate with humans and bots within the same loop
- Automates enterprise processes using LLM-powered logic
- Hybrid systems combine AI agents and traditional orchestration tools

Implementation guidance

You can build this pattern using the following tools and AWS services:

- Amazon Bedrock for reasoning and agent selection
- AWS Step Functions or Amazon EventBridge for workflow composition
- AWS Lambda as execution units or task runners
- Amazon DynamoDB, Amazon Simple Storage Service (Amazon S3), or Amazon RDS to track states and results
- AWS AppFabric or Amazon AppFlow for cross-system coordination
- (Optional) Use Amazon SageMaker run agent to host domain-specific worker agents

Summary

Workflow agents coordinate, adapt, and align goals in multi-agent environments. This means that AI agents can collaborate, adapt to runtime conditions, and deliver complex outcomes through modular, explainable workflows.

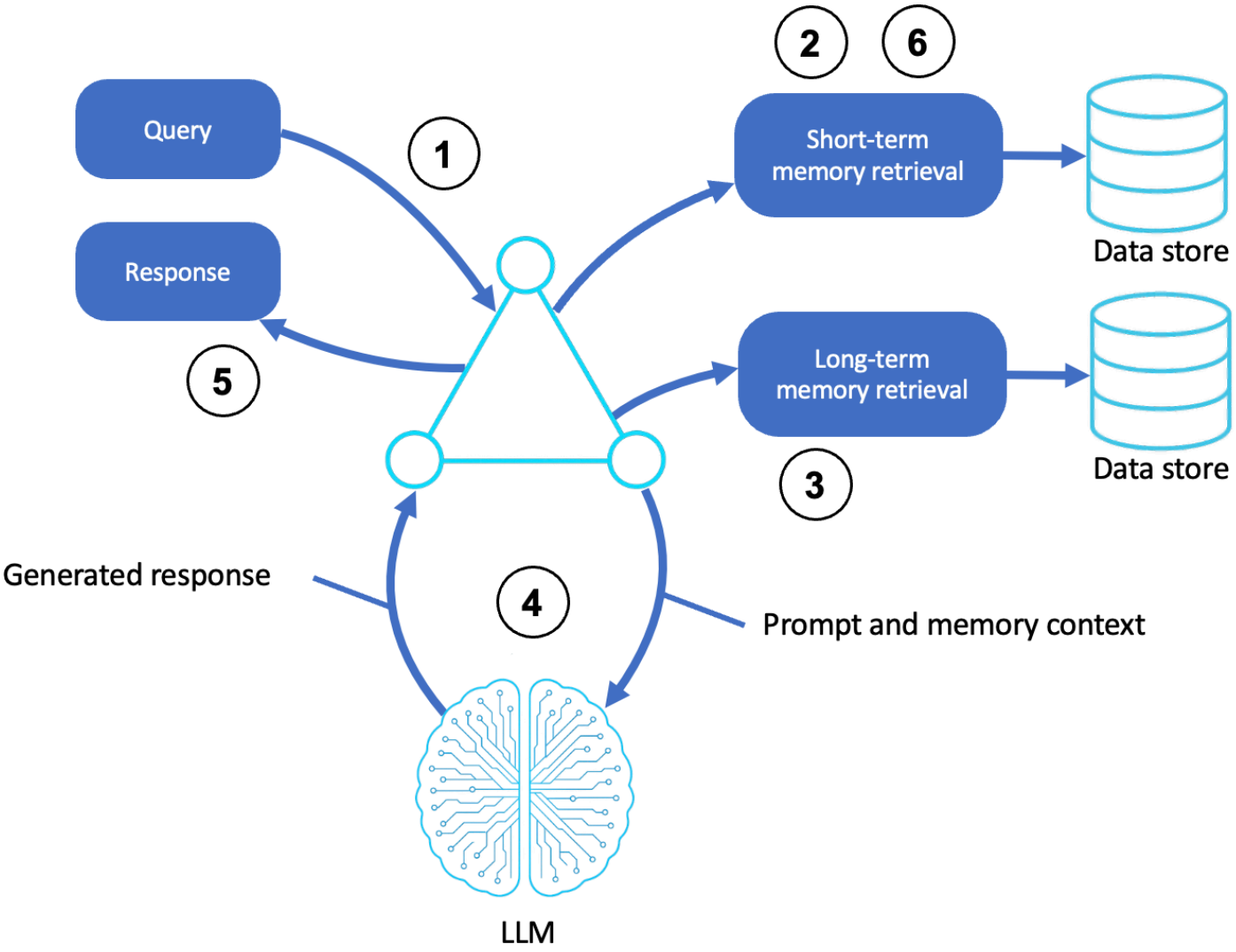
Memory-augmented agents

Memory-augmented agents are enhanced with the ability to store, retrieve, and reason using short-term and long-term memory. This allows them to maintain context across multiple tasks, sessions, and interactions, which produces more coherent, personalized, and strategic responses.

Unlike stateless agents, memory-augmented agents adapt by referencing historical data, learn from prior outcomes, and make decisions that align with the user's goals, preferences, and environment.

Architecture

A memory-augmented agent is shown in the following diagram:



Description

1. Receives input or event
   - The agent receives a user query or system event. This may be a text, API trigger, or environmental change.
2. Retrieves short-term memory
   - The agent retrieves recent conversational history, task context, or the system state that's relevant to the session or workflow.
3. Retrieves long-term memory
   - The agent queries long-term memory (for example, vector databases and key-value stores) for historical insights, such as the following:

    - User preferences
    - Past decisions and outcomes
    - Learned concepts, summaries, or experiences
4. Reasons through the LLM
    - The memory context is embedded into the LLM prompt, allowing the agent to reason based on both current inputs and prior knowledge.
5. Generates outputs
    - The agent produces a contextually aware response, plan, or action that is personalized according to the task history and user's inputs.
6. Updates memory
    - New information, such as updated goals, success and failure signals, and structured responses, are stored for future tasks.

Capabilities

- Session continuity across conversations or events
- Goal persistence over time
- Contextual awareness based on an evolving state
- Adaptability informed by prior successes and failures
- Personalization aligned with user preferences and history

Common use cases

- Conversational copilots that remember user preferences
- Coding agents that track codebase changes
- Workflow agents that adapt according to task history
- Digital twins that evolve from system knowledge
- Research agents that avoid redundant retrievals

Implementing memory-augmented agents

Use the following tools and AWS services for memory-augmented agents:

Memory layer	AWS service	Purpose
Short-term	Amazon DynamoDB, Redis, Amazon Bedrock context	Fast retrieval of recent interaction states
Long-term (structured)	Amazon Aurora, Amazon DynamoDB, Amazon Neptune	Facts, relationships, and logs
Long-term (semantic)	OpenSearch, Pinecone	Embedding-based retrieval (that is, RAG)
Storage	Amazon S3	Storing transcripts, structured memories, and files
Orchestration	AWS Lambda or AWS Step Functions	Managing memory injection and update lifecycle
Reasoning	Amazon Bedrock	Anthropic Claude or Mistral with memory prompts

Implementing memory-injected prompting

To integrate memory into agent reasoning, use a combination of structured state and retrieval-augmented context injection:

- Include the latest agent state and recent dialogue history as structured input when constructing the prompt for the language model, so it can reason with full context.
- Use retrieval-augmented generation (RAG) to pull relevant documents or facts from long-term memory.
- Summarize previous plans, context, and interactions for compression and relevance.
- Inject external memory modules, such as vector stores or structured logs, during inference to guide decision making.

Summary

Memory-augmented agents maintain thought continuity by learning from experience and remembering user context. These agents surpass reactive intelligence by using long-term

collaboration, personalization, and strategic reasoning. In terms of agentic AI, memory allows agents to behave more like adaptive digital counterparts and less like stateless tools.

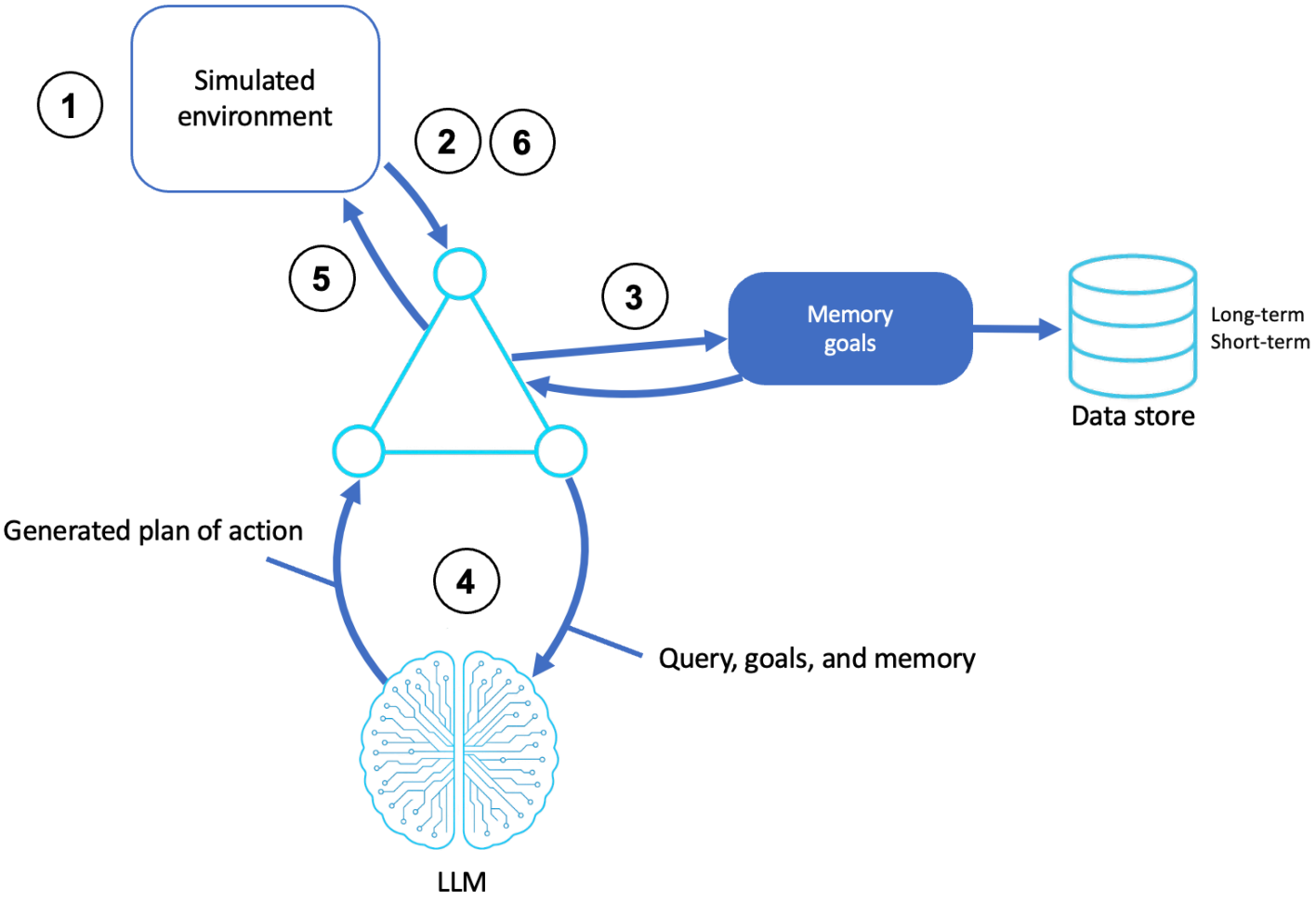
Simulation and test-bed agents

Simulation and test-bed agents operate within virtualized or controlled environments where they reason, act, and learn. These agents simulate behavior, model outcomes, and train strategies in repeatable settings before applying them to real-world environments.

This pattern is useful for iterative development, reinforcement learning (RL), autonomous decision-making evaluation, and emergent behavior testing. Simulation agents often operate in closed loops, receiving feedback from their environment and adjusting their behavior accordingly, making them critical for tasks that involve spatial reasoning, real-time control, or complex system dynamics.

Architecture

The following diagram shows a simulation or test-bed agent:



Description

1. Initiates an environment
   - The agent initiates a simulated environment (for example, a 3D world, physics engine, CLI sandbox, or synthetic data stream).
   - The agent is loaded into the environment with an initial task, goal, or policy.
2. Perceives agent
   - The agent perceives the current state through simulation telemetry (for example, sensor emulation, virtual camera, and structured logs).
3. Retrieves goal and memory
   - The agent retrieves its assigned objective, scenario instructions, or contextual goal.
   - It may also retrieve prior memory, including the following:
     - Long-term strategies or policies

    - Environmental maps or known constraints
    - Past successes or failures from similar simulations
4. Reasons and plans
    - An LLM interprets the simulated state, task objectives, and learned knowledge.
    - It generates a plan of action or control command.
5. Executes simulated actions
    - The agent executes the plan, modifies state, navigates space, or interacts with virtual entities.
6. Learns
    - Agent evaluates action outcomes
    - Depending on the agent's configuration, it may do the following:
        - Perform RL
        - Log outcomes for future fine-tuning
        - Adapt strategies in real time

Capabilities

- Operates within synthetic or virtual environments
- Supports trial-and-error learning, policy refinement, and system modeling
- Low-risk testing for behavior, failure handling, and edge cases
- Enables emergent agent behavior analysis in multi-agent setups
- Supports both closed-loop control and human-in-the-loop exploration

Common use cases

- Reinforcement learning for robotics, drones, and gaming
- Autonomous vehicle training on virtual roads
- Simulated UIs or CLIs for DevOps and test-bed scenarios
- Emergent behavior experiments in social simulations
- Safety validation of decision logic prior to production

Implementation guidance

You can build a simulation and test-bed agent using the following tools and AWS services:

Component	AWS service	Purpose
Environment	AWS RoboMaker, Amazon ECS, Amazon EC2, or a custom simulator in Amazon SageMaker studio lab	Run virtual worlds (Gazebo, Unity, Unreal) or sandbox CLIs
Agent logic	Amazon Bedrock, Amazon SageMaker, or AWS Lambda	LLM-based planners or RL agents
Feedback loop	Amazon SageMaker reinforcement learning, Amazon CloudWatch, or custom logs	Reward tracking, outcome scoring, and behavior logging
Memory and replay	Amazon S3, Amazon DynamoDB, or Amazon RDS	Persistent state, episode history, or scenario data
Visualization	Amazon CloudWatch dashboards or Amazon SageMaker notebooks	Observe policy changes, outcomes, and training metrics

The following are additional applications:

- AWS SimSpace Weaver for large-scale spatial simulations
- AWS IoT Core for testing shadow devices
- Amazon SageMaker Experiments for agent evaluation and benchmarking

Summary

Simulation and test-bed agents are for structured exploration prior to being deployed to production systems. Use these agents to train autonomous navigation policies, test business processes in synthetic environments, and evaluate swarms for coordination patterns.

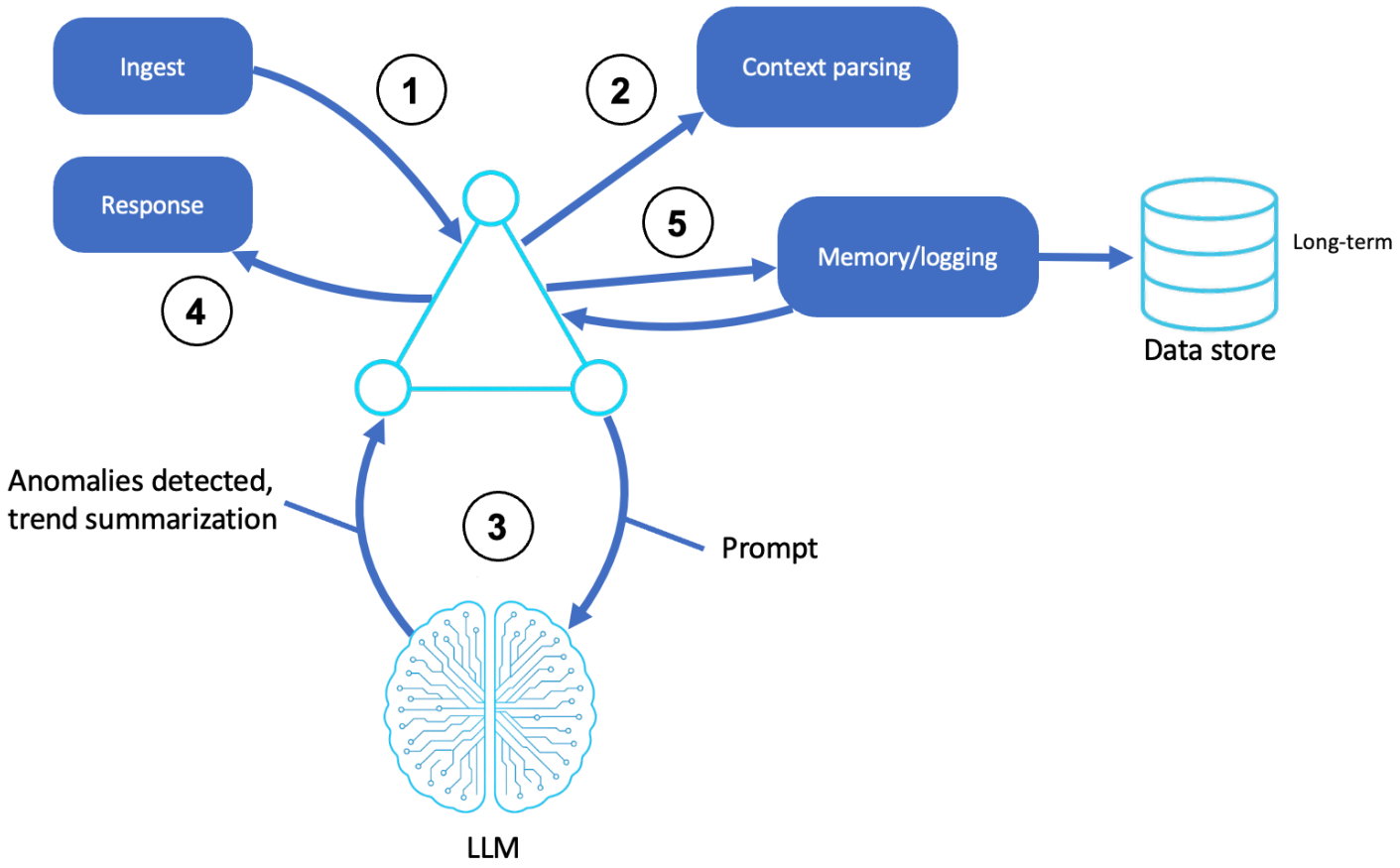
Observer and monitoring agents

Observer and monitoring agents passively observe systems, environments, and interactions to detect patterns, generate insights, and trigger actions. As intelligent watchers, they enhance alerts, diagnostics, and audits without directly initiating behavior.

These agents excel where traditional monitoring lacks adaptability or reasoning, particularly for AI-in-the-loop monitoring, anomaly detection, compliance oversight, and security intelligence. Observer agents are event listeners that continuously monitor system telemetry and user interactions. The agent depends on perception, interpretation, and conditional escalation or reporting.

Architecture

The following diagram shows an observer and monitoring agent:



Description

1. Ingest telemetry

- The agent receives input from one or more system sources, such as the following:
  - Logs (application, infrastructure, security)
  - Metrics (performance, latency, usage)
  - Events (API calls, user actions, sensor data)

2. Parse context
   - Raw input is parsed, structured, and enriched with metadata, such as a timestamp, actor identity, system state, and trace ID.
3. Reasons using LLM
   - The agent uses an LLM or logic module to interpret parsed inputs by identifying anomalies, summarizing trends, and correlating across distributed traces or time windows.
4. Classify or alert
   - The agent determines if the observed behavior warrants the following:
     - An alert or escalation
     - A report or dashboard update
     - A response trigger (for example, automatic remediation and policy enforcement)
5. Log memory or feedback loops
   - The agent stores events and decisions for long-term learning, audits, or future reference for other agents.

Capabilities

- Passive and noninvasive (agent doesn't directly act)
- Highly scalable and asynchronous
- AI-driven correlation across noisy or distributed signals
- Supports audit, compliance, and real-time insight
- Can feed downstream agents or human workflows

Common use cases

- AI-augmented observability for microservices and APIs
- Monitoring for model drift, policy violation, or out-of-band behavior
- Customer activity analysis or interaction summaries

- Code review agents that monitor commits or deployments
- Security or compliance log monitoring using LLM reasoning

Implementation guidance

You can build an observer and monitoring agent using the following tools and AWS services:

Component	AWS service	Purpose
Event ingestion	Amazon EventBridge, Amazon CloudWatch Logs, Amazon Kinesis, Amazon S3	Ingest structured and unstructured telemetry
Preprocessing	AWS Lambda, AWS Glue, AWS Step Functions	Transform raw data into structured prompts
Reasoning engine	Amazon Bedrock, Amazon SageMaker, AWS Lambda	Analyze events, classify behavior, generate insights
Storage and memory	Amazon S3, Amazon DynamoDB, OpenSearch	Persistent observations, summaries, and outputs
Alerting and escalation	Amazon SNS, AWS AppFabric, Amazon EventBridge	Trigger downstream systems or agents

The following are additional applications:

- AWS Security Hub CSPM for security log monitoring
- AWS Quick for visualizing agent outputs

Summary

Observer and monitoring agents track systems and behaviors in real time. They detect anomalies, audit security, and gather operations intelligence by identifying patterns that humans or rules might overlook. This capability helps create systems that can adapt to changing conditions and make decisions based on comprehensive data analysis.

Multi-agent collaboration

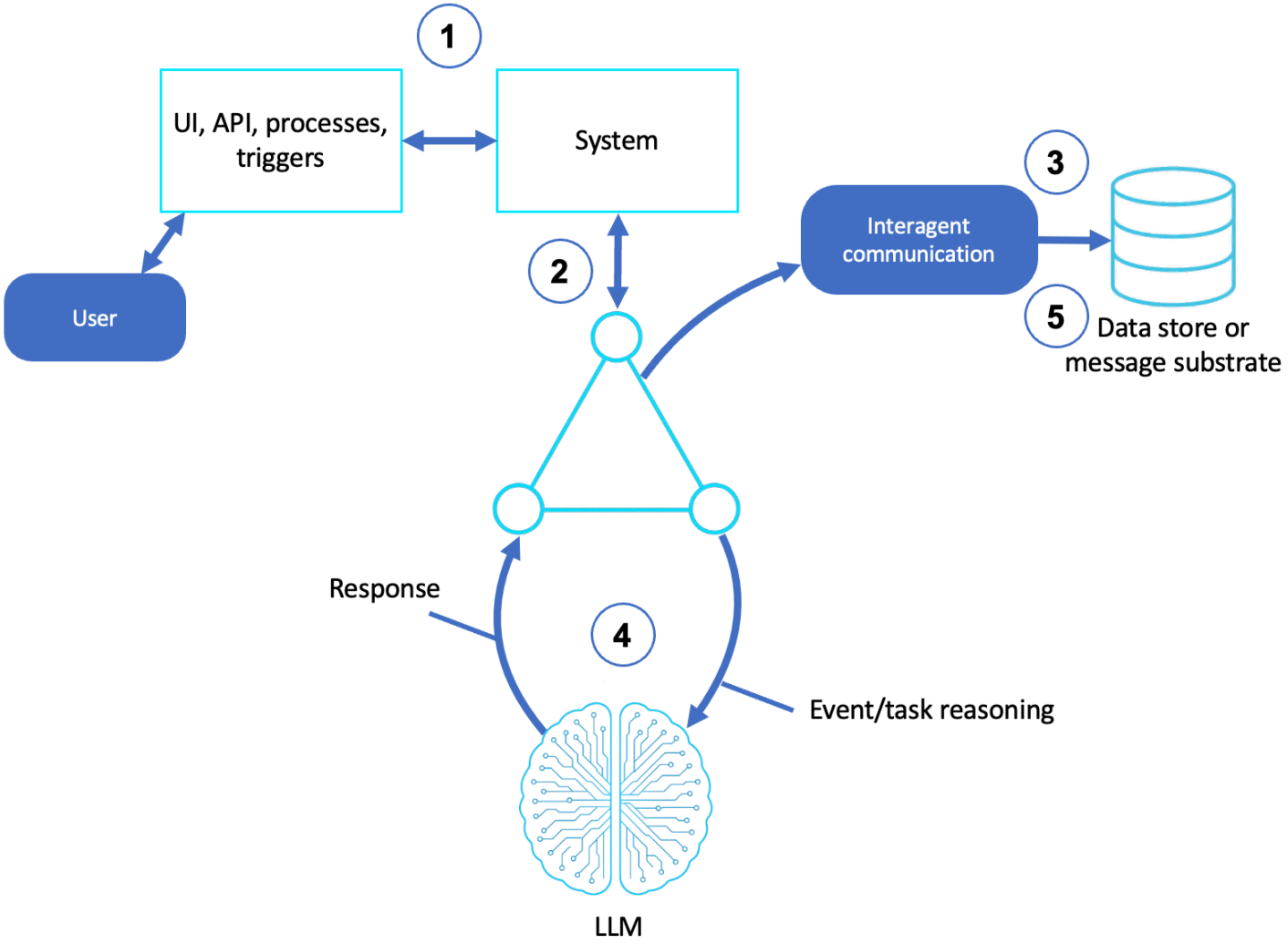
Multi-agent collaboration refers to a pattern in which multiple autonomous agents, each with a distinct role, specialization, or objective, negotiate to solve complex tasks. These agents may operate independently or with other agents by sharing information, dividing responsibilities, and collectively reasoning toward a goal.

This pattern differs from workflow agents, which centrally coordinate and delegate tasks to subordinate agents in a structured flow. In contrast, multi-agent collaboration emphasizes peer-to-peer or emergent coordination by enabling adaptivity, parallelism, and division of cognition. The following table compares multi-agent collaboration with workflow agents:

Feature	Workflow agents	Purpose
Control	Centralized coordinator	Decentralized, distributed, or role-based peers
Interaction	One agent delegates and tracks execution	Multiple agents negotiate, share, and adapt
Design	Predefined sequence of tasks	Emergent, flexible task distribution
Coordination	Procedural orchestration	Cooperative or competitive interactions
Use cases	Enterprise process automatio n	Complex reasoning, explorati on, and emergent strategies

Architecture

The following diagram shows multi-agent collaboration:



Description

1. Initiates a task
   - A user or system emits a high-level goal or problem.
   - A "manager" agent or initiating context defines the objective.
2. Assigns or discovers roles
   - Agents self-assign (symbolic logic or reasoning) or are delegated (event broker) to other roles, such as a planner, researcher, executor, critic, or explainer.
3. Communicates with other agents
   - Agents communicate through shared memory, messaging queues, or prompt chaining.
   - They may debate, query, or propose subtasks to one another.
4. Uses specialized reasoning

- Each agent uses its own model or domain logic to solve its portion of the problem.
- Agents can use LLMs with role-specific prompts and memory.

5. Coordinates outputs or goals
    - The agents synthesize contributions into a final answer, plan, or action.
    - (Optional) A supervising agent may validate or summarize the synthesized output.

Capabilities

- Peer-level agents with specialized roles or skills
- Emergent behavior through communication or negotiation
- Parallel processing of complex or multifaceted problems
- Supports deliberation, self-correction, and reflective iteration
- Model social dynamics, scientific collaboration, or enterprise team roles

Common use cases

- Autonomous research teams (search agent, summarizer, and validator)
- Software development (planner, coder, and tester)
- Business scenario modeling (finance, policy, and compliance)
- Negotiation, bidding, or multiparty reasoning
- Multimodal tasks (image, text, and logic)

Implementation guidance

You can build a multi-agent system using the following tools and AWS services:

Component	AWS service	Purpose
Agent hosting	Amazon Bedrock, Amazon SageMaker, AWS Lambda	Host individual LLM-driven agents
Communication layer	Amazon SQS, Amazon EventBridge, AWS AppFabric	Messaging and coordination between agents

Shared memory	Amazon DynamoDB, Amazon S3, or OpenSearch	Multi-agent memory or blackboard system
Orchestration layer	AWS Step Functions, AWS Lambda pipelines	Kickoff, timeout, fallback, and retry logic
Agent identification	Amazon Bedrock agents (role-defined), AWS AppConfig and Amazon Bedrock converse API (agents outside of Amazon Bedrock)	Role-based tool or agent invocation and boundary enforcement
Emergent interaction	Amazon EventBridge pipelines or agent registries	Enable dynamic task routing or escalation

Summary

Multi-agent collaboration distributes problem-solving tasks across modular, role-driven agents. Unlike workflow orchestration, collaboration patterns use emergent intelligence, resilience, and scalability that mirror how humans solve problems. It's especially valuable for open-ended domains, creative tasks, multimodal reasoning, and environments that benefit from diverse perspectives.

Conclusion

The patterns previously discussed illustrate foundational approaches to real-world implementations of agentic AI. From basic reasoning to memory-augmented intelligence, each pattern is uniquely configured for perception, cognition, and action that is based on autonomy, asynchrony, and agency.

These patterns share vocabularies and technical blueprints for building intelligent, goal-directed systems. Whether a pattern is embedded in a user interface, orchestrated through cloud services, or coordinated across teams of agents, each pattern is adaptable and modular.

Takeaways

- **Agent patterns are composable** – Most real-world agents blend two or more patterns (for example, a voice agent with tool-based reasoning and memory).

- **Agent design is contextual** – Choose patterns based on the interaction surface, task complexity, latency tolerance, and domain-specific constraints.
- **AWS native implementation is achievable** – With Amazon Bedrock, Amazon SageMaker, AWS Lambda, AWS Step Functions, and event-driven architectures, every agent pattern can be delivered at scale.

LLM workflows

In agent patterns, we explored the common AI agent patterns, each built around a set of modular capabilities: perception, action, learning, and cognition. At the heart of the cognitive module in many agent patterns is a large language model (LLM) that is capable of reasoning, planning, and decision-making. However, invoking an LLM alone is not sufficient to produce intelligent, goal-directed behavior.

To perform complex tasks reliably, agents must embed the LLM within a structured workflow, where the model's capabilities are augmented with tools, memory, planning loops, and coordination logic. These LLM workflows allow an agent to break down goals, route subtasks, call external services, reflect on results, and coordinate with other agents.

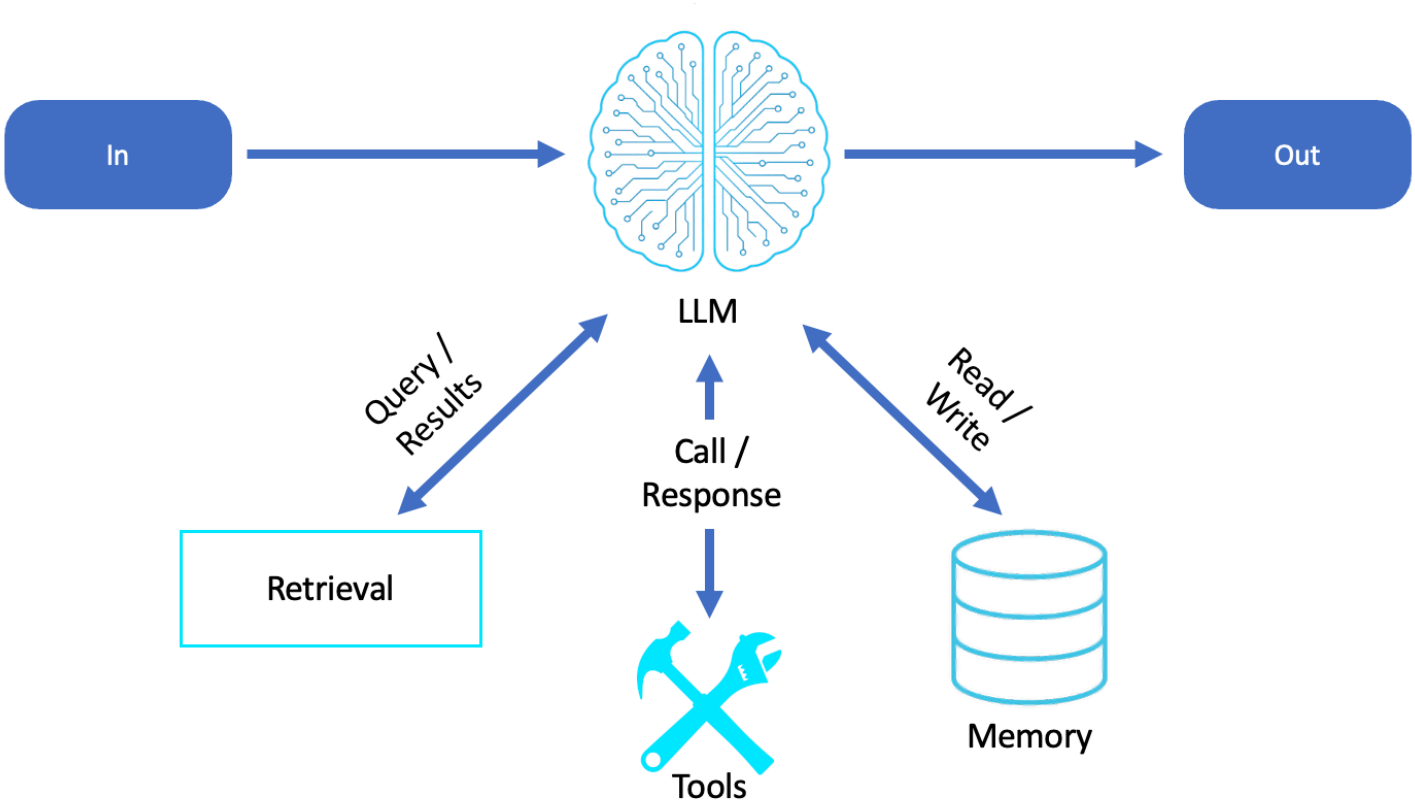
This chapter introduces the core design patterns for building robust, extensible, and intelligent LLM-driven cognitive modules, organized around reusable workflows.

Overview of LLM-augmented cognition

At its core, the cognitive module of a software agent can be viewed as an LLM wrapped in augmentations. The agent can use the following building blocks to reason effectively within its environment:

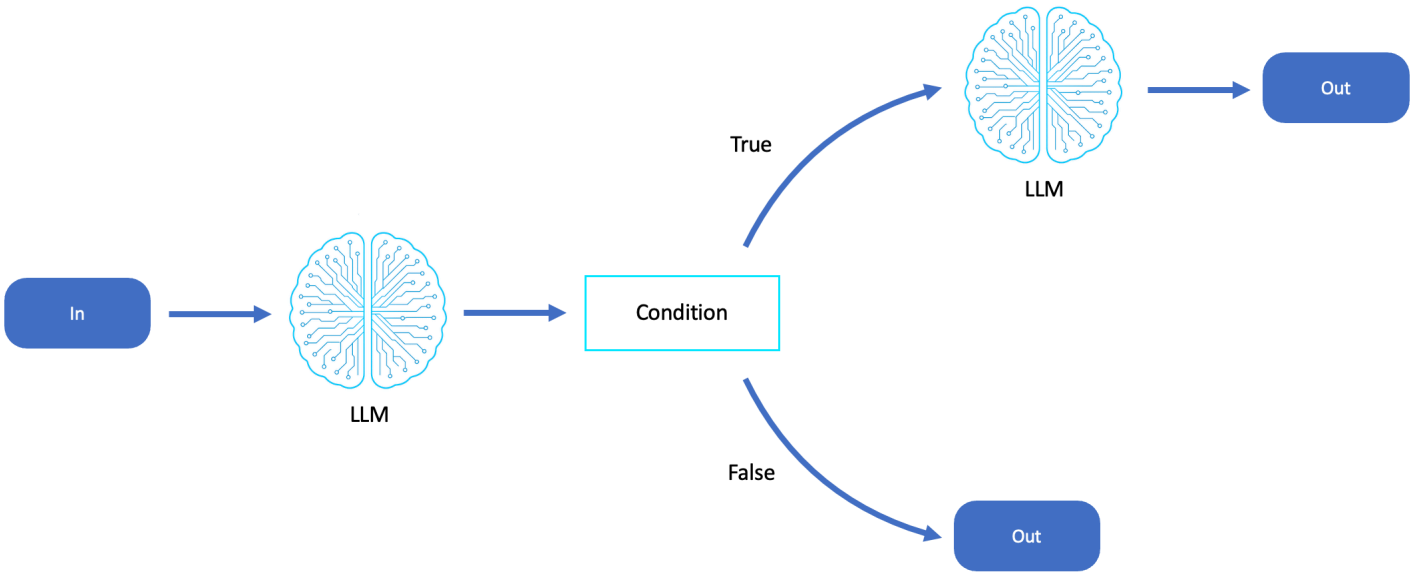
- **Prompting** – Framing input using context, instructions, examples, and memory
- **Retrieval** – Providing up-to-date or domain-specific knowledge to the LLM prompt through vector search or semantic memory, for example, through retrieval-augmented generation (RAG)
- **Tool use** – Enabling the LLM to invoke APIs or call functions to retrieve or act on information
- **Memory** – Incorporating persistent or a session-based state into the reasoning loop, either by using structured databases or contextual summaries

These augmentations are composed of workflows that define how the LLM is used over time and across tasks, transforming it from a stateless engine into a dynamic reasoning agent.



Workflow for prompt chaining

Prompt chaining decomposes complex tasks into a sequence of steps, where each step is a discrete LLM invocation that processes or builds upon the output of the previous one.



The prompt chaining workflow is suited for scenarios where tasks can be logically divided into sequential reasoning steps, and where intermediate outputs inform the next stage. It excels in workflows that require structured thinking, progressive transformation, or layered analysis, such as document review, code generation, knowledge extraction, and content refinement.

Description

- The complexity of the task exceeds the context window or reasoning depth of a single LLM call.
- Outputs from one step (for example, analysis, summarization, or planning) become inputs for a follow-up decision or generation phase.
- You need transparency and control across reasoning stages (for example, auditable intermediate results).
- You want to plug in external validation, filtering, or enrichment logic between steps.
- It's ideal for agents operating in pipeline-style reasoning loops, such as research agents, editorial assistants, planning systems, and multistage copilots.

Capabilities

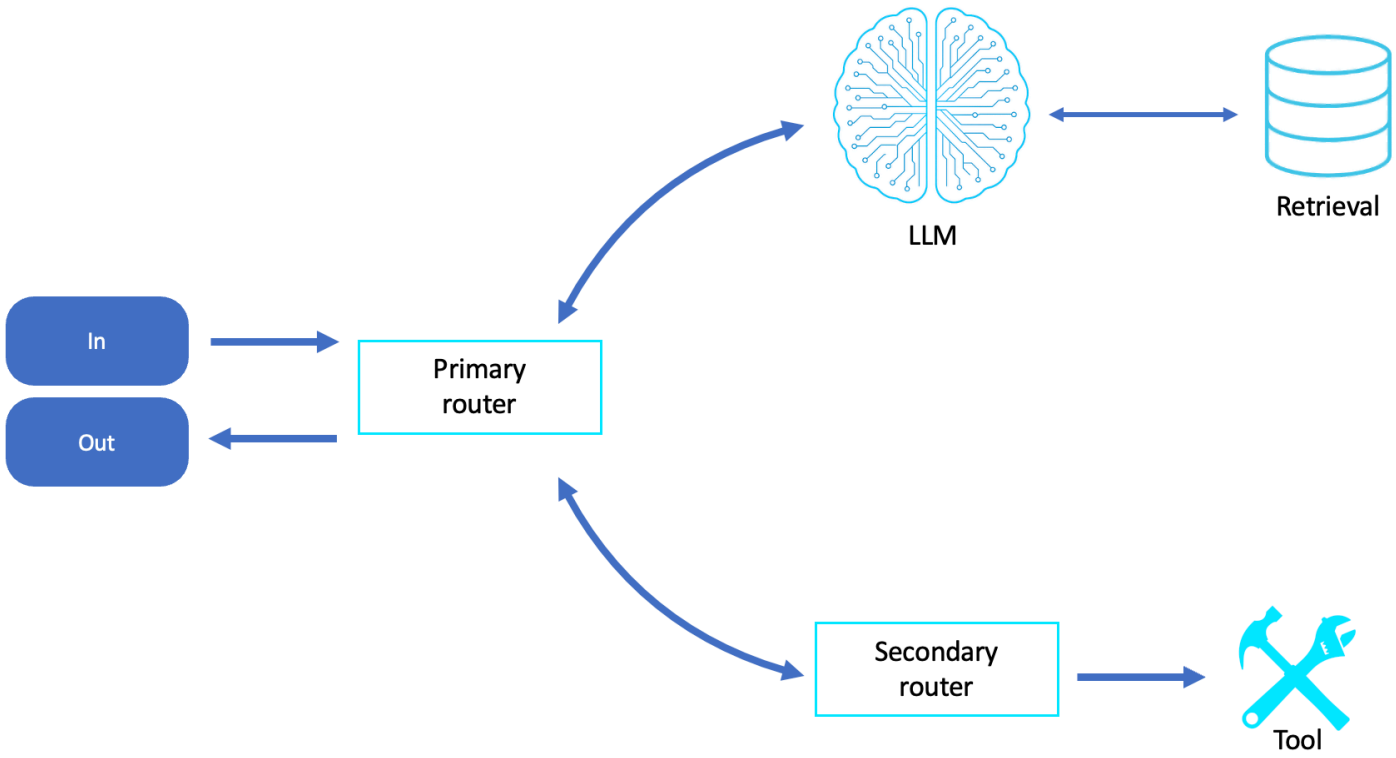
- Linear or branching chains of LLM calls
- Intermediate results passed as structured input or embedded into follow-up prompts
- Can be orchestrated with AWS Step Functions, AWS Lambda, or agent-specific runners

Common use cases

- Multistep reasoning tasks (for example, "summarize critique rewrite")
- Research assistants synthesizing layered outputs (for example, "search extract facts answer question")
- Code generation pipelines ("generate plan write code test code explain output")

Workflow for routing

In the routing pattern, a classifier or router agent uses an LLM to interpret the intent or category of a query, then routes the input to a specialized downstream task or agent.



The Routing workflow is used in scenarios where an agent must quickly classify input intent, task type, or domain, and then delegate the request to a specialized subagent, tool, or workflow. It is especially useful in capability agents, such as those that serve as general assistants, front doors to enterprise functions, or user-facing AI interfaces that span domains.

Routing is particularly effective when:

- Triaging requests across a variety of tasks (for example, search, summarization, booking, calculations).
- Inputs must be preprocessed or normalized before entering more specialized workflows.
- Different input types (for example, images vs. text, structured vs. unstructured queries) require custom handling.
- An agent is acting as a conversational switchboard, delegating tasks to specialized agents or microservices.
- This workflow is common in domain-specific copilots, customer-support bots, enterprise service routers, and multimodal agents, where intelligent dispatching determines both the quality and efficiency of agent behavior.

Capabilities

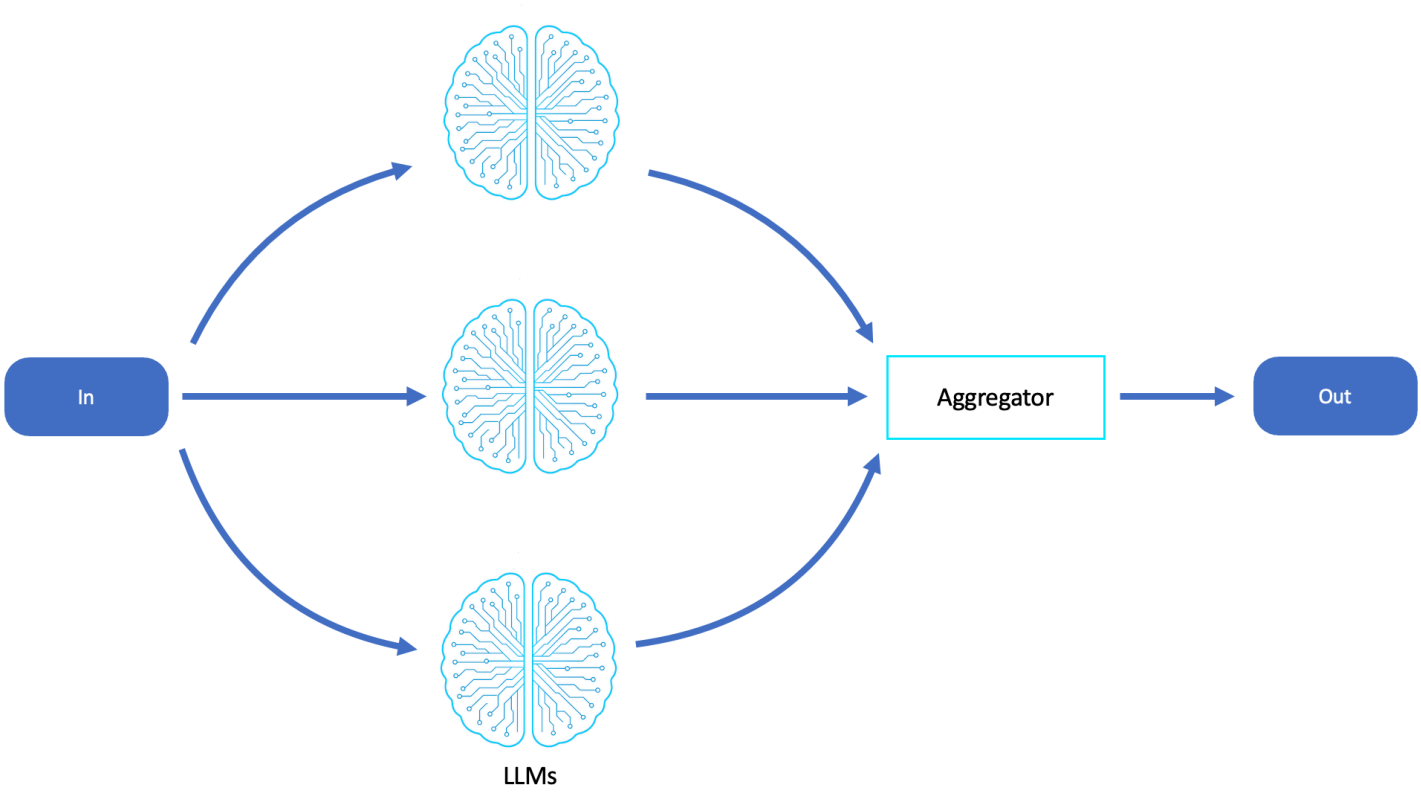
- A first-pass LLM acts as a dispatcher
- Routes can invoke distinct workflows or even other agent patterns
- Supports modular expansion of capabilities

Common use cases

- Multidomain assistants ("is this a legal, medical, or financial question?")
- Decision trees enhanced with LLM reasoning
- Dynamic tool selection (for example, search vs. code generation)

Workflow for parallelization

This workflow involves breaking down a task into independent subtasks that can be handled concurrently by multiple LLM calls or agents. Outputs are then programmatically aggregated and synthesized into a result.



The Parallelization workflow is used when a task can be divided into independent, nonsequential subtasks that can be processed simultaneously, significantly improving efficiency, throughput, and scalability. It is especially powerful in data-heavy, batch-oriented, or multiperspective problem spaces where the agent must analyze or generate content across multiple inputs.

Parallelization is particularly effective when:

- Subtasks do not depend on each other's intermediate results, allowing them to run in parallel without coordination.
- A task involves repeating the same reasoning process across many items (for example, summarizing multiple documents or evaluating a list of options).
- Multiple hypotheses or perspectives are explored in parallel to promote diversity, creativity, or robustness.
- You need to reduce latency for high-volume or high-frequency requests through concurrent LLM execution.
- This workflow is commonly used in document processing agents, survey or comparison engines, batch summarizers, multi-agent brainstormers, and scalable classification or labeling tasks, especially where rapid, parallel reasoning is a performance advantage.

Capabilities

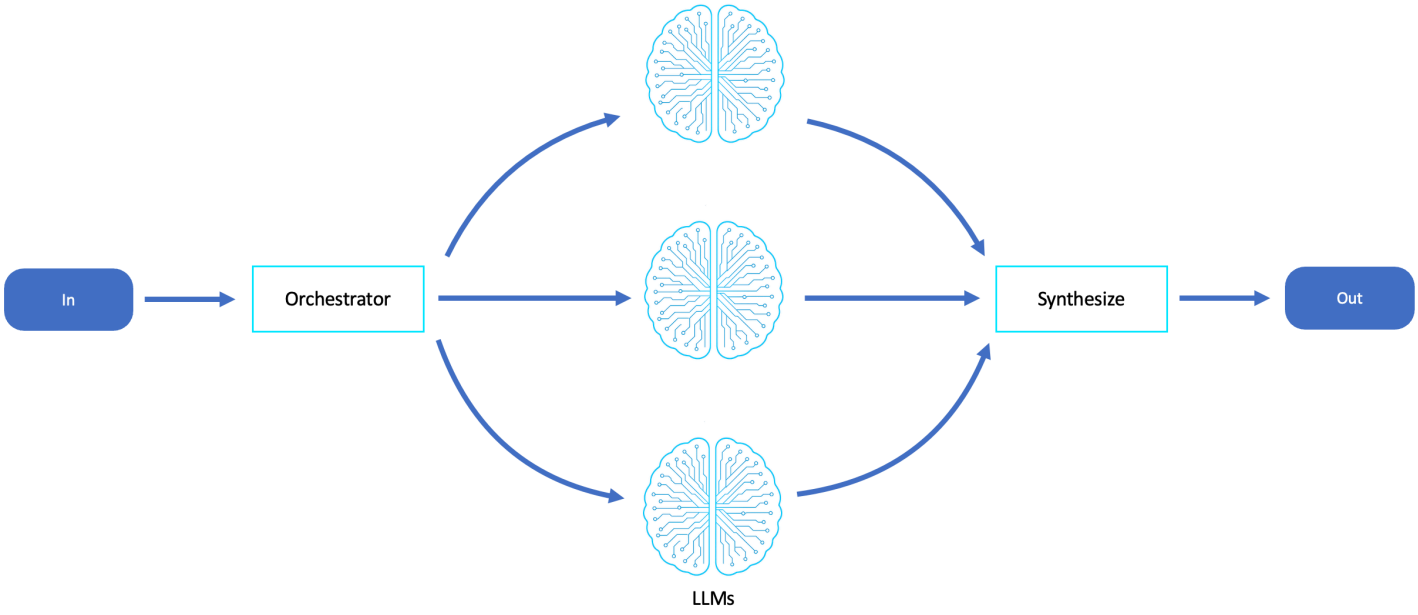
- Parallel execution of LLM tasks (by using AWS Lambda, AWS Fargate, or an AWS Step Functions map state)
- Requires result alignment, validation, or deduplication at the synthesis stage
- Well-suited for stateless agent loops

Common use cases

- Analyzing multiple documents or perspectives in parallel
- Generating diverse drafts, summaries, or plans
- Accelerating throughput across batch jobs

Workflow for orchestration

A central orchestrator agent uses an LLM to plan, decompose, and delegate subtasks to specialized worker agents or models, each with a specific role or domain expertise. This mirrors human team structures and supports emergent behavior across multiple agents.



The orchestration workflow is ideal for scenarios that are complex, hierarchical, or multidisciplinary, requiring structured decomposition and specialized execution. It is particularly well-suited to tasks that require division of labor, where different subcomponents of a task are best handled by agents with distinct capabilities, knowledge, or toolsets.

This workflow is particularly effective when:

- Tasks can be divided into subtasks that vary in scope, type, or reasoning (for example, plan, research, implement, and test).
- An LLM or meta-agent must coordinate other agents, monitor progress, and synthesize results.
- You want to modularize agent responsibilities, enabling scalability, reuse, and specialized tuning.
- The system requires role-based behavior, mimicking how human teams (for example, project managers, developers, and reviewers) operate in collaboration.

Orchestration is ideal for multiturn planning agents, software development copilots, enterprise process agents, and autonomous project executors. It is especially useful when implementing

multi-agent systems that require centralized task breakdown but distributed execution logic, enabling extensibility and more explainable behavior across agent layers.

Capabilities

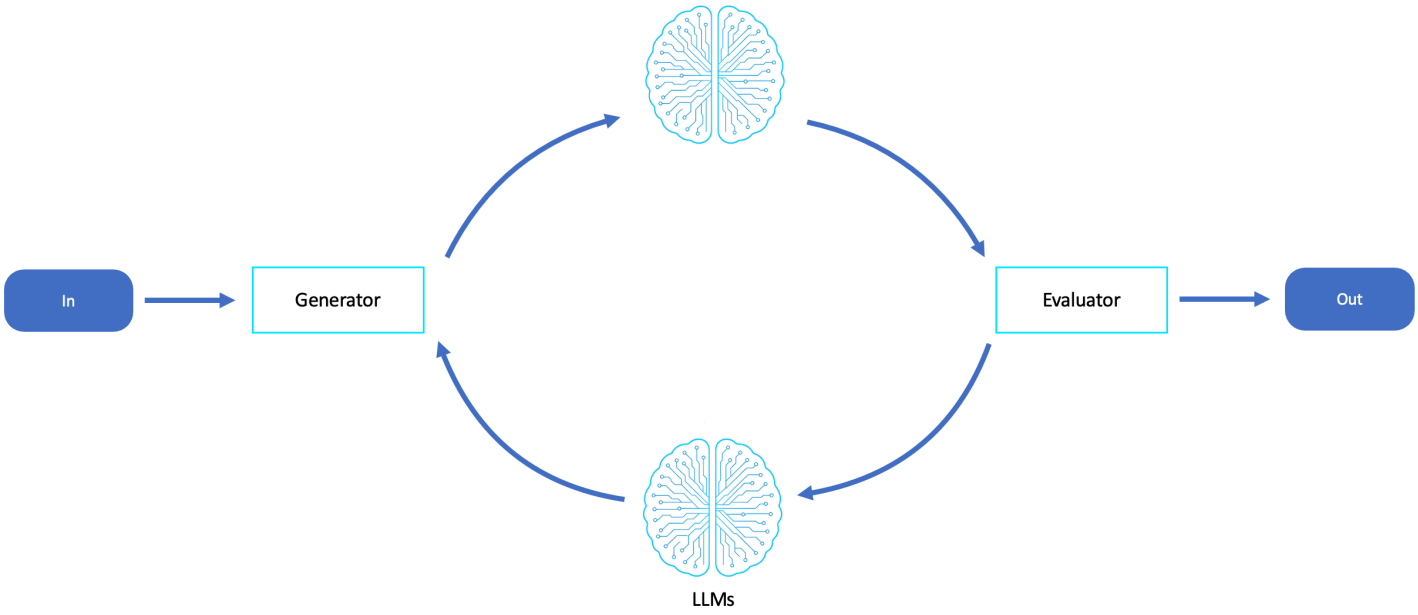
- Orchestrator performs goal meta-reasoning
- Worker agents may include tool access, memory, or domain-specific prompting
- Can be hierarchical (that is, multilevel task delegation)

Common use cases

- Project managers, coordinating researchers, writers, and quality-assurance agents
- Coding copilots that combine planning, executing, and testing
- Agents that supervise toolchains or API access patterns

Workflow for evaluators and reflect-refine loops

This workflow provides a feedback loop where one LLM generates a result, and another evaluates or critiques the result. This promotes self-reflection, optimization, and iterative improvements.



The evaluator workflow is ideal for scenarios where output quality, accuracy, and alignment are important and where single-pass generation is unreliable or insufficient. This workflow excels when

agents must self-critique, iterate, and refine their outputs—either to meet a higher standard of correctness or to explore improved alternatives based on feedback.

This workflow is particularly effective when:

- The output involves subjective quality metrics (for example, style, tone, and readability) or objective criteria (for example, correctness, safety, and performance).
- The agent must reason through trade-offs, evaluate constraints, or optimize toward a goal.
- You require built-in redundancy and quality assurance, especially in regulated, customer-facing, or creative domains.
- Human-in-the-loop review is expensive or unavailable, and autonomous validation is desired.

This workflow is used for content generation, code synthesis and review, policy enforcement, alignment checking, instruction tuning, and RAG postprocessing. It is also useful for self-improving agents, where continuous feedback helps shape better responses over time to build trustworthy, autonomous decision loops.

Common use cases

- Red-team agents compared to blue-team agents
- Agents that generate, evaluate, and revise code or plans
- Quality assurance, hallucination detection, and style enforcement

Capabilities

- Supports decoupled generation and evaluation using different models (for example, Claude for generation and Mistral for evaluation)
- Feedback is structured and used to prompt revised outputs
- Supports multiple iterations or convergence thresholds

Conclusion

LLMs provide the cognitive core of modern software agents, but raw model invocation is not enough to achieve purposeful, robust, and controllable intelligence. To move from output generation to structured reasoning and goal-aligned behavior, LLMs must be embedded in

intentional workflow patterns that define how models process inputs, manage contexts, and coordinate actions.

LLM workflows introduce foundations to build an agent's cognitive module:

- Prompt chaining breaks down complex reasoning into modular, auditable steps.
- Routing enables intelligent task classification and targeted delegation.
- Parallelization accelerates throughput and promotes diverse reasoning.
- Agent orchestration structures multi-agent collaboration through task decomposition and role-based execution.
- Evaluator (reflect-refine loop) enables self-improvement, quality control, and alignment checking.

Each workflow represents a composable pattern that can be adapted to the agent's needs, the complexity of the task, and a user's expectations. These workflows are not mutually exclusive. They are building blocks that are often combined into hybrid architectures that support dynamic reasoning, multi-agent coordination, and enterprise-grade reliability.

As you transition to the next chapter on agentic workflow patterns, these LLM workflows will reappear as embedded structures within larger systems, supporting goal delegation, tool orchestration, decision loops, and lifecycle autonomy. Mastering these LLM workflows is essential to designing software agents that don't just predict text but reason, adapt, and act purposefully.

Agentic workflow patterns

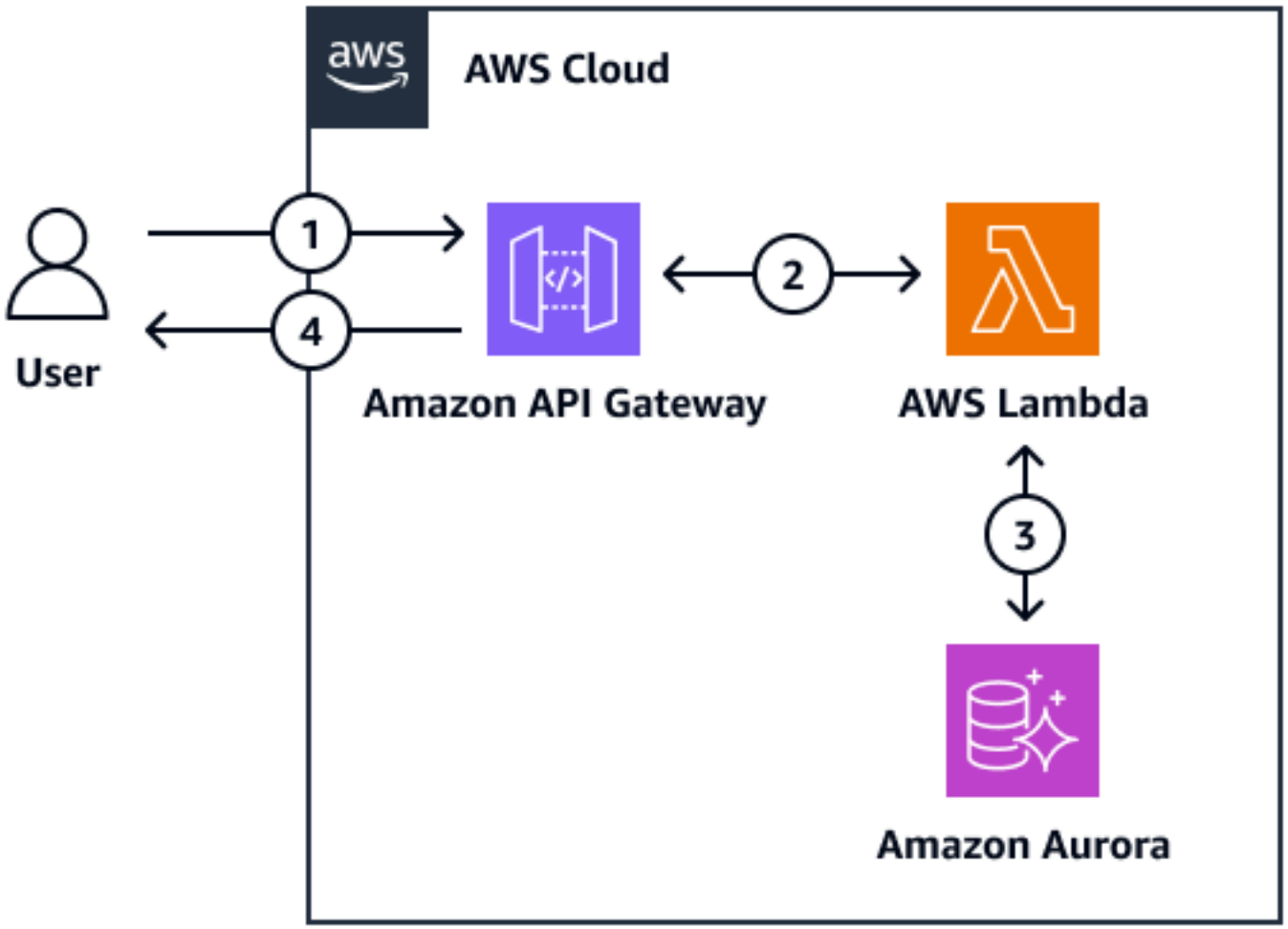
Agentic workflow patterns integrate modular software agents with structured large language model (LLM) workflows, enabling autonomous reasoning and action. While inspired by traditional serverless and event-driven architectures, these patterns shift core logic from static code to LLM-augmented agents, providing enhanced adaptability and contextual decision-making. This evolution transforms conventional cloud architectures from deterministic systems to ones capable of dynamic interpretation and intelligent augmentation, while maintaining fundamental principles of scalability and responsiveness.

From event-driven to cognition-augmented systems

Modern cloud architectures, particularly those built on serverless and event-driven principles, have traditionally relied on patterns like routing, fan-out, and enrichment to create responsive, scalable systems. Agentic AI systems build upon these foundations while reframing them around LLM-augmented reasoning and cognitive flexibility. This approach allows for more sophisticated problem-solving and automation capabilities, potentially revolutionizing how complex tasks are handled in cloud environments.

Event-driven architecture

The following diagram shows a typical distributed system:

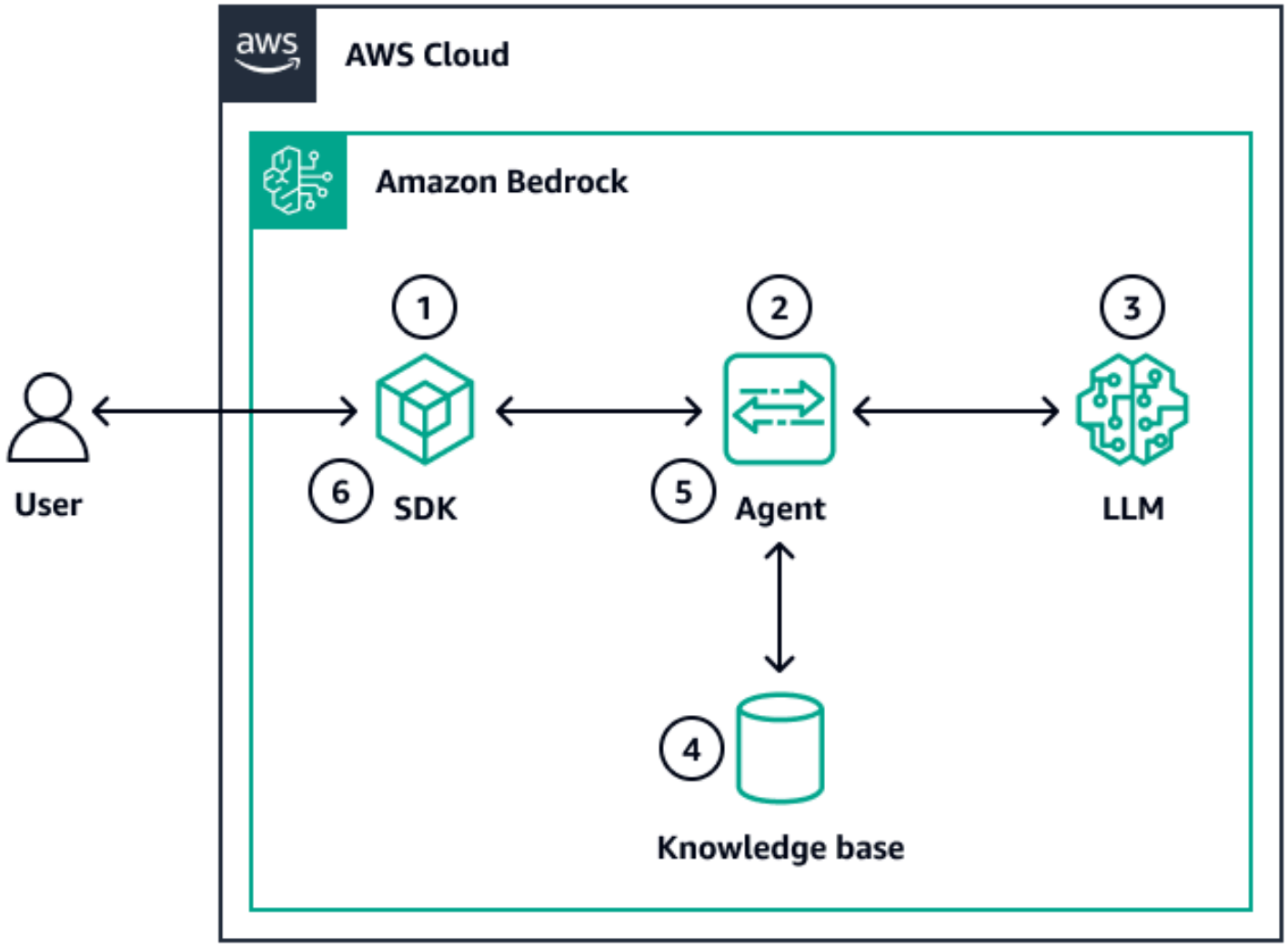


1. A user submits a request to Amazon API Gateway.
2. Amazon API Gateway routes the request to an AWS Lambda function.
3. AWS Lambda performs data enrichment by querying an Amazon Aurora database
4. Amazon API Gateway returns the enriched payload to the caller.

This structure is both reliable and scalable, but it's fundamentally static. Business rules and logic paths must be explicitly coded, and adapting to changing contexts or incomplete information is limited.

Cognition-augmented workflows

Agentic architectures add cognitive augmentation to an event-driven system. The following diagram shows an agentic equivalent:



1. A user submits a query through an SDK or API call.
2. An Amazon Bedrock agent receives the query.
3. The agent interprets the query by invoking an LLM
4. The agent performs semantic enrichment by searching the Amazon Bedrock knowledge base or other external data sources.
5. The LLM synthesizes a context-rich, goal-aligned response.
6. The system returns a synthesized response to the user.

In this flow, the LLM uses logic, understands intent, retrieves and combines relevant context, and then decides how best to respond. This pattern mirrors the traditional enrichment pattern, where messages are augmented with external data before being routed further. In agentic systems,

however, this enrichment is not a static lookup. Instead, the enrichment is dynamic, semantically guided, and driven by purpose.

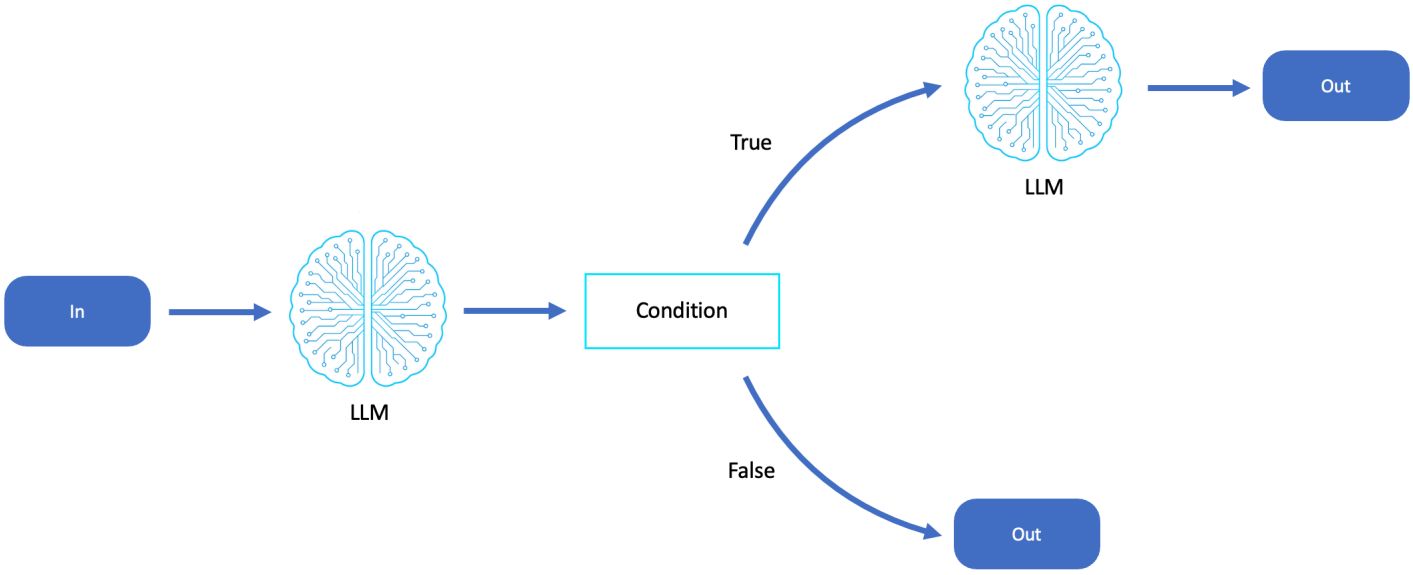
Core insights

Each LLM workflow can be mapped to an agentic workflow pattern, which mirrors and evolves traditional event-driven architecture styles. A basic building block of agentic workflows is the ability to augment an LLM's context with data, tools and memory. This creates a reasoning loop that's informed, adaptive, and aligned with user intent. Where traditional systems enrich messages with lookup data, agentic systems enable software to act less like scripts and more like intelligent collaborators.

Prompt chaining saga patterns

By reimagining LLM prompt chaining as an event-driven saga, we unlock a new operational model: workflows become distributed, recoverable, and semantically coordinated across autonomous agents. Each prompt-response step is reframed as an atomic task, emitted as an event, consumed by a dedicated agent, and enriched with contextual metadata.

The following diagram is an example of LLM prompt chaining:

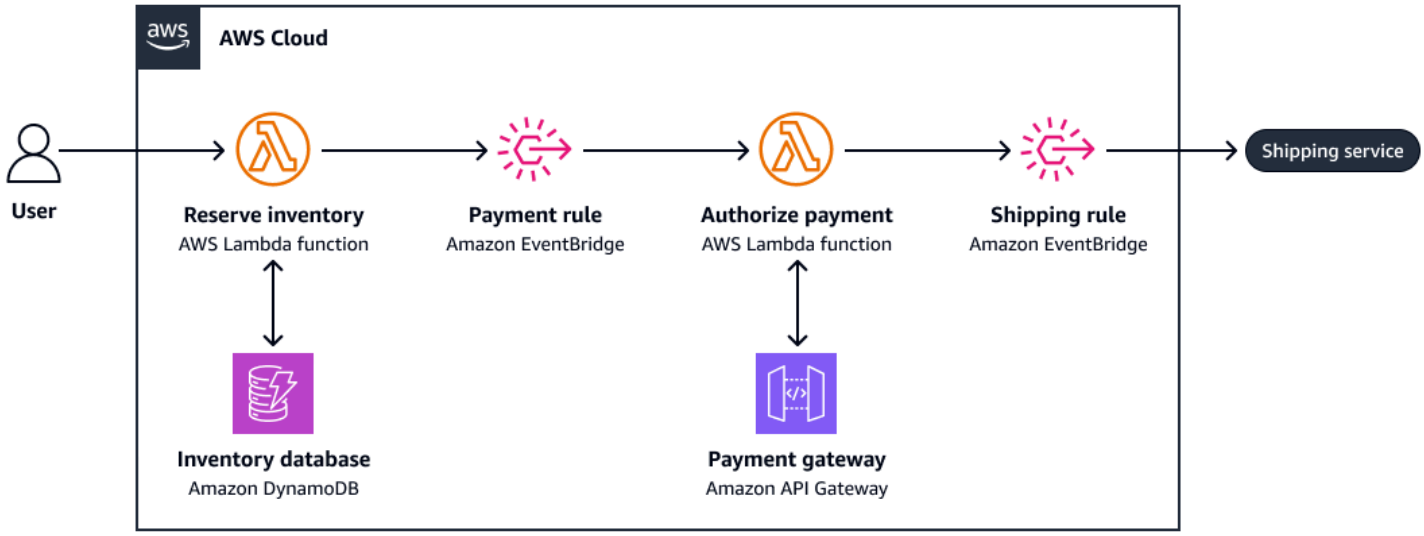


Saga choreography

The saga choreography pattern is an implementation approach in distributed systems that has no central coordinator. Instead, each service or component publishes events that trigger the next

workflow action. This pattern is widely used in distributed systems for managing transactions across multiple services. In a saga, the system runs a series of coordinated local transactions. If one fails, the system triggers compensating actions to maintain consistency.

The following diagram is an AWS architectural example of saga choreography:



1. Reserve inventory
2. Authorize payment
3. Create shipping order

If step 3 fails, the system invokes compensating actions (for example, cancel a payment or release inventory).

This pattern is especially valuable in event-driven architectures where services are loosely coupled and states must be consistently resolved over time, even in the presence of partial failure.

Prompt chaining pattern

Prompt chaining resembles the saga pattern in both structure and purpose. It executes a series of reasoning steps that build sequentially while preserving context and allowing for rollbacks and revisions.

Agent choreography

1. LLM interprets a complex user query and generates a hypothesis
2. LLM elaborates a plan to solve the task

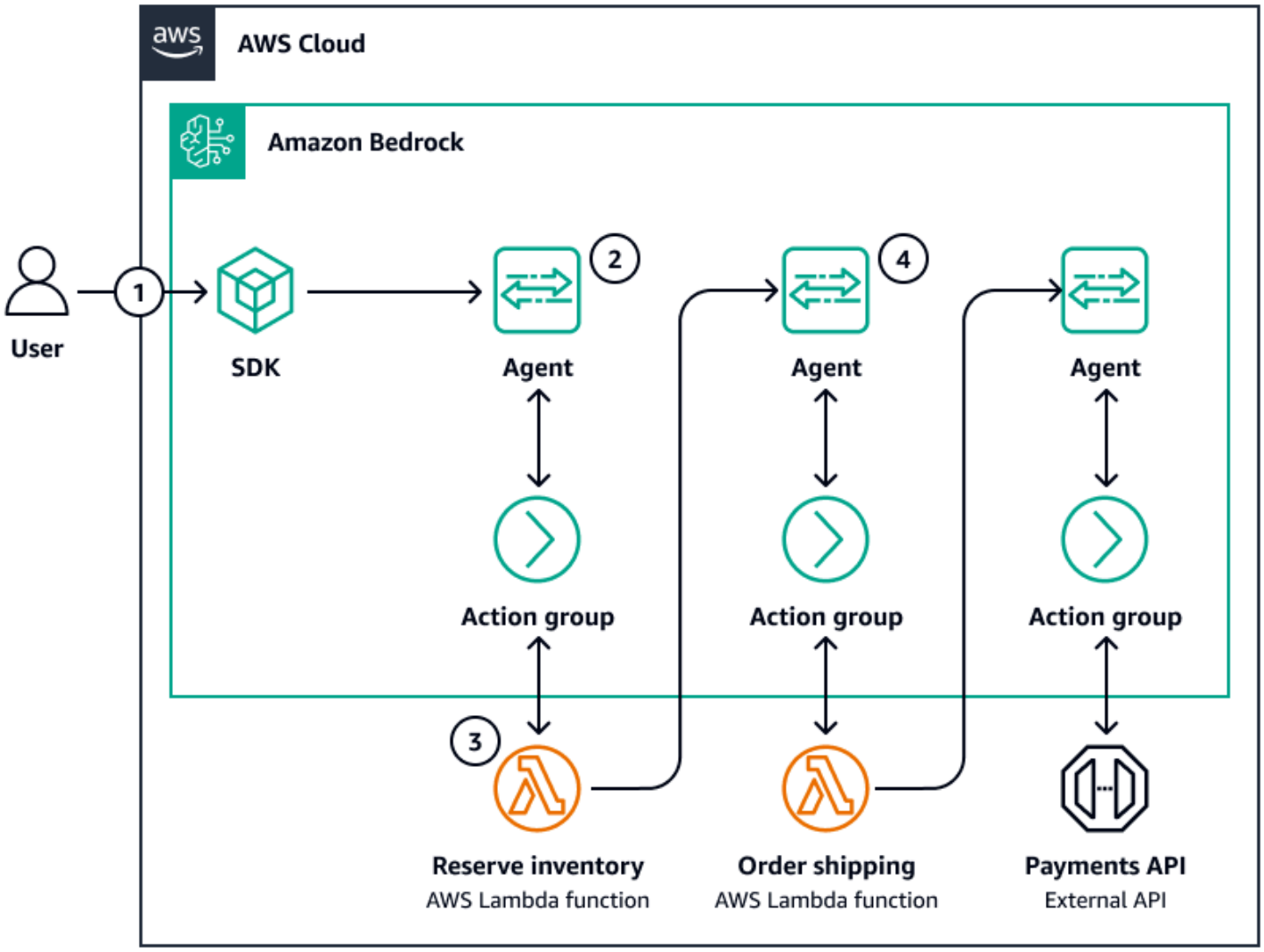
3. LLM executes a subtask (for example, by using a tool call or retrieving knowledge)
4. LLM refines the output or revisits an earlier step if it deems a result unsatisfactory

If an intermediate result is flawed, the system can do one of the following:

- Retry the steps using a different approach
- Revert to a previous prompt and replan
- Use an evaluator loop (for example, from the evaluator-optimizer pattern) to detect and correct failures

Like the saga pattern, prompt chaining allows for partial progress and rollback mechanisms. This happens through iterative refinement and LLM-directed correction rather than through compensating database transactions.

The following diagram is an AWS architectural example of agent choreography:



1. A user submits a query through an SDK.
2. An Amazon Bedrock agent orchestrates reasoning through the following:
   - Interpretation (LLM)
   - Planning (LLM)
   - Execution through a tool or knowledge base
   - Response construction
3. If a tool fails or returns insufficient data, the agent can dynamically replan or rephrase the task.
4. Memory (for example, a short-term vector store) can preserve its state across steps

Takeaways

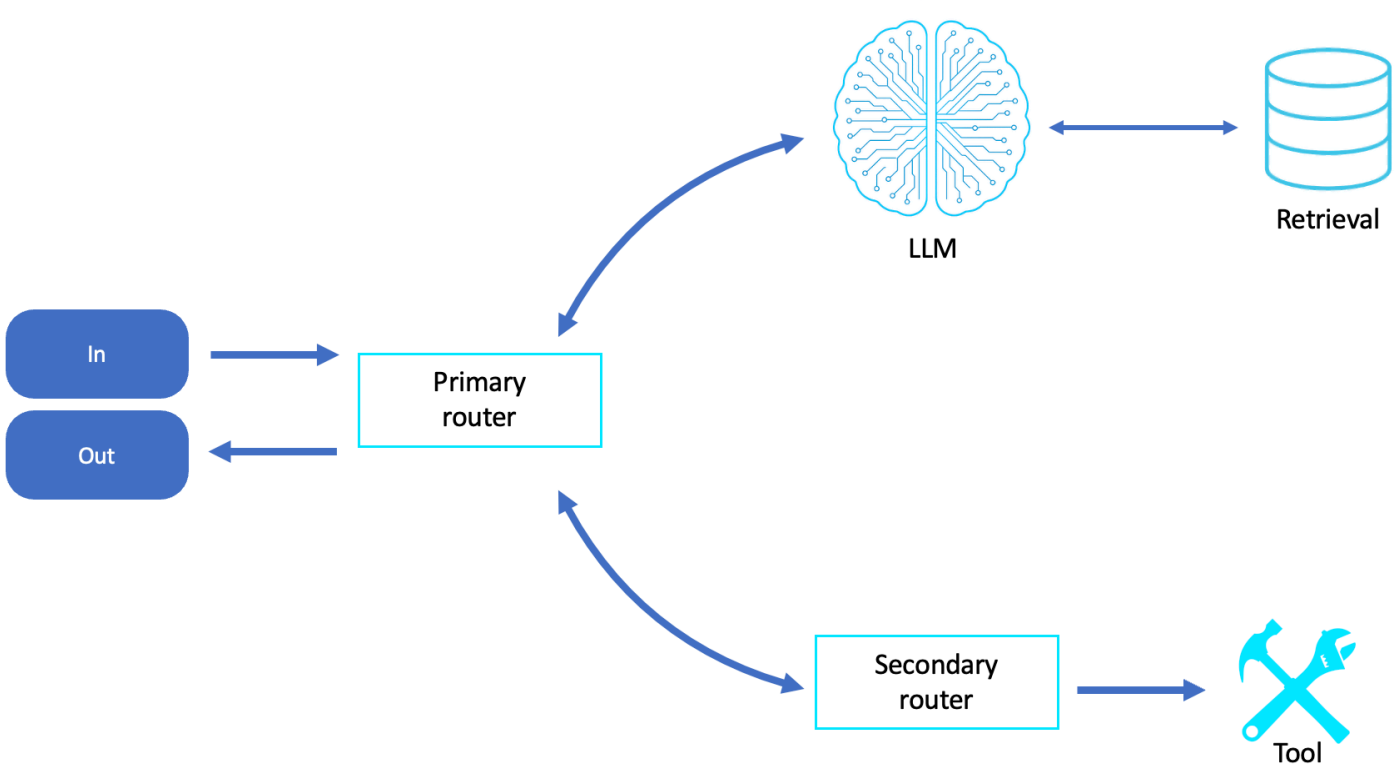
Where the saga pattern manages distributed service calls with compensating logic, prompt chaining manages reasoning tasks with reflective sequencing and adaptive replanning. Both systems allow for incremental progress, decentralized decision points, and failure recovery, and it does all of this through informed reasoning rather than rigid rollback.

Prompt chaining introduces transactional reasoning, which is the cognitive equivalent of sagas. That is, each "thought" is reevaluated, revised, or abandoned as part of a broader goal-directed dialogue.

Routing dynamic dispatch patterns

In modern agentic systems, where tasks range from document parsing to autonomous software generation, the ability to dynamically route requests to the most capable large language model (LLM) or agent becomes critical. Static routing logic, often embedded within orchestration scripts or API layers, lacks the adaptability required for real-time, multi-model, multi-capability environments. To address this, LLM routing workflows can be transformed into an event-driven architecture that leverages a dynamic dispatch pattern, turning LLM calls into intelligently routed, context-aware events.

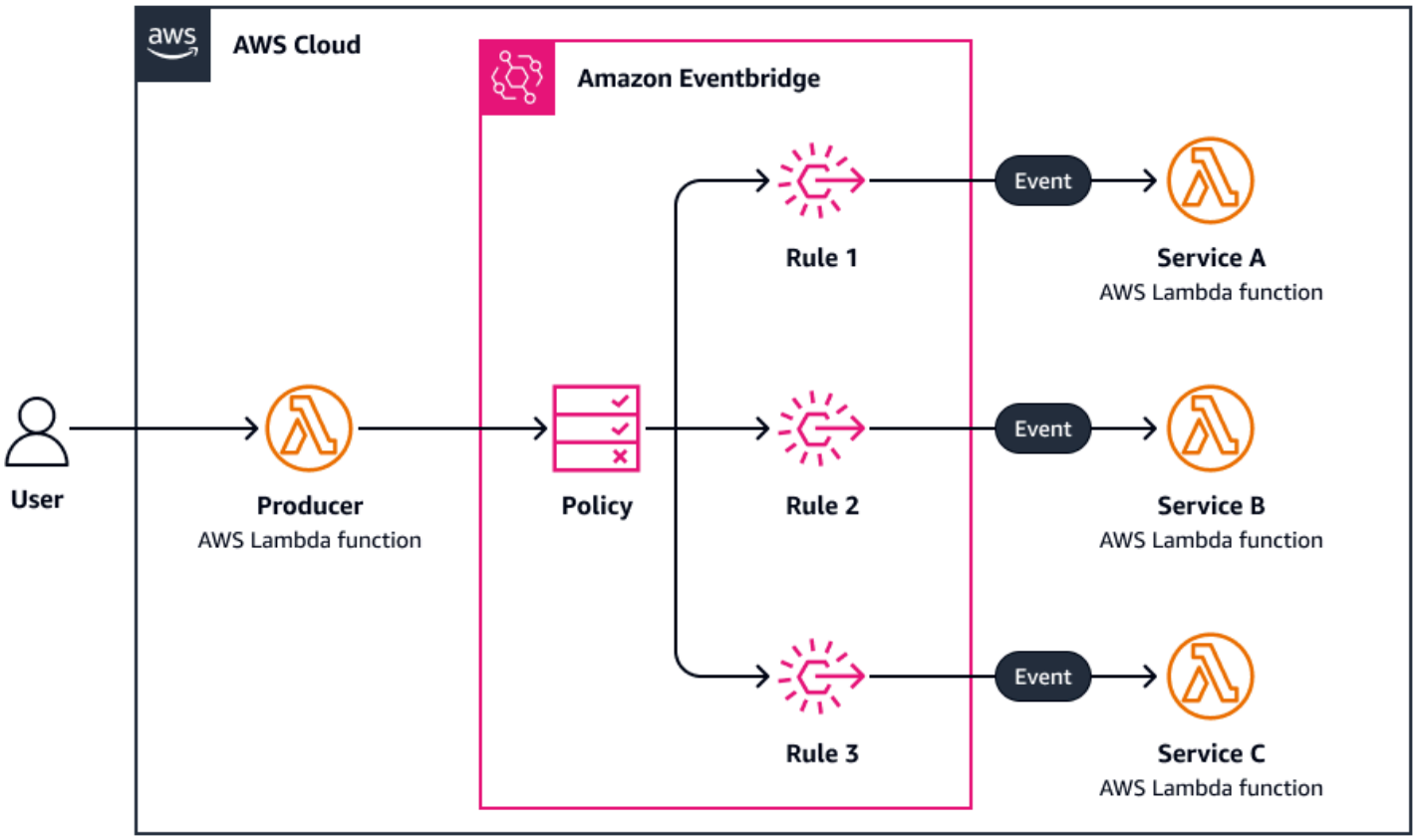
The following diagram is an example of LLM routing:



Dynamic dispatch

In traditional distributed systems, the dynamic dispatch pattern selects and invokes specific services at runtime based on incoming event attributes, such as event type, source, and payload. This is commonly implemented using Amazon EventBridge, which can evaluate and route incoming events to appropriate targets (for example, AWS Lambda functions AWS Step Functions, or Amazon Elastic Container Service tasks).

The following diagram is an AWS architectural example of dynamic dispatch:



1. An application emits an event (for example, {"type": "orderCreated", "priority": "high"}).
2. Amazon EventBridge evaluates the event against its routing rules.
3. Based on an event's attributes, the system dynamically dispatches to the following:
   - `HighPriorityOrderProcessor` (service A)
   - `StandardOrderProcessor` (service B)
   - `UpdateOrderProcessor` (service C)

This pattern supports loose coupling, domain-based specialization, and runtime extensibility. This allows systems to respond intelligently to changing requirements and event semantics.

LLM-based routing

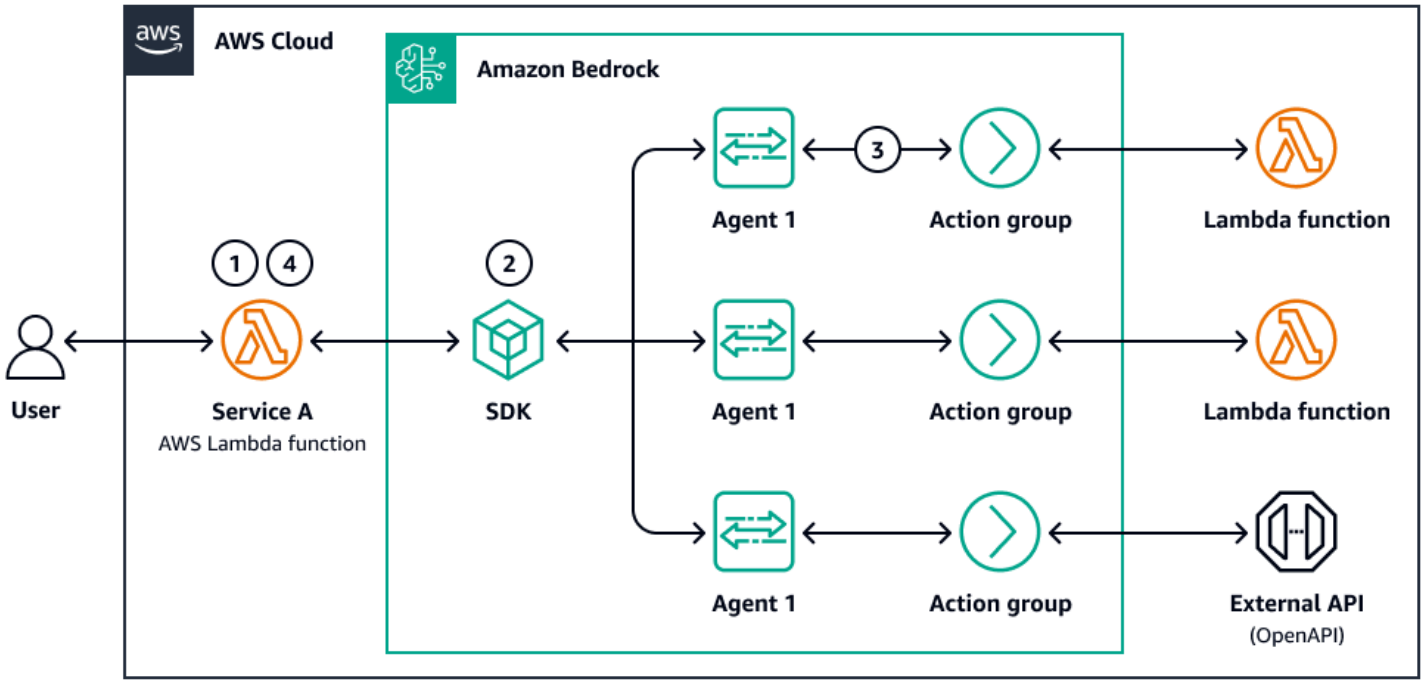
In agentic systems, routing also performs dynamic task delegation – but instead of Amazon EventBridge rules or metadata filters, the LLM classifies and interprets the user's intent through natural language. The result is a flexible, semantic, and adaptive form of dispatching.

Agent router

This architecture enables rich intent-based dispatching without predefined schemas or event types, which is ideal for unstructured input and complex queries.

1. A user submits the request "Can you help me review my contract terms?"
2. The LLM interprets this as a legal document task.
3. The agent routes the task to one or more of the following:
   - Contract review prompt template
   - Legal reasoning subagent
   - Document parsing tool

The following diagram is an AWS architectural example of an agent router:



1. A user submits a natural language request through an SDK.
2. An Amazon Bedrock agent uses an LLM to classify the task (for example, legal, technical, or scheduling).
3. The agent dynamically routes the task through an action group to invoke the required agent:
   - Domain-specific agent
   - Specialized tool chain

- Custom prompt configuration

4. The selected handler processes the task and returns a tailored response.

Takeaways

Where traditional dynamic dispatch uses Amazon Bedrock rules for routing based on structured event attributes, agentic routing uses LLMs to semantically classify and route tasks based on meaning and intent. This expands the system's flexibility by enabling the following:

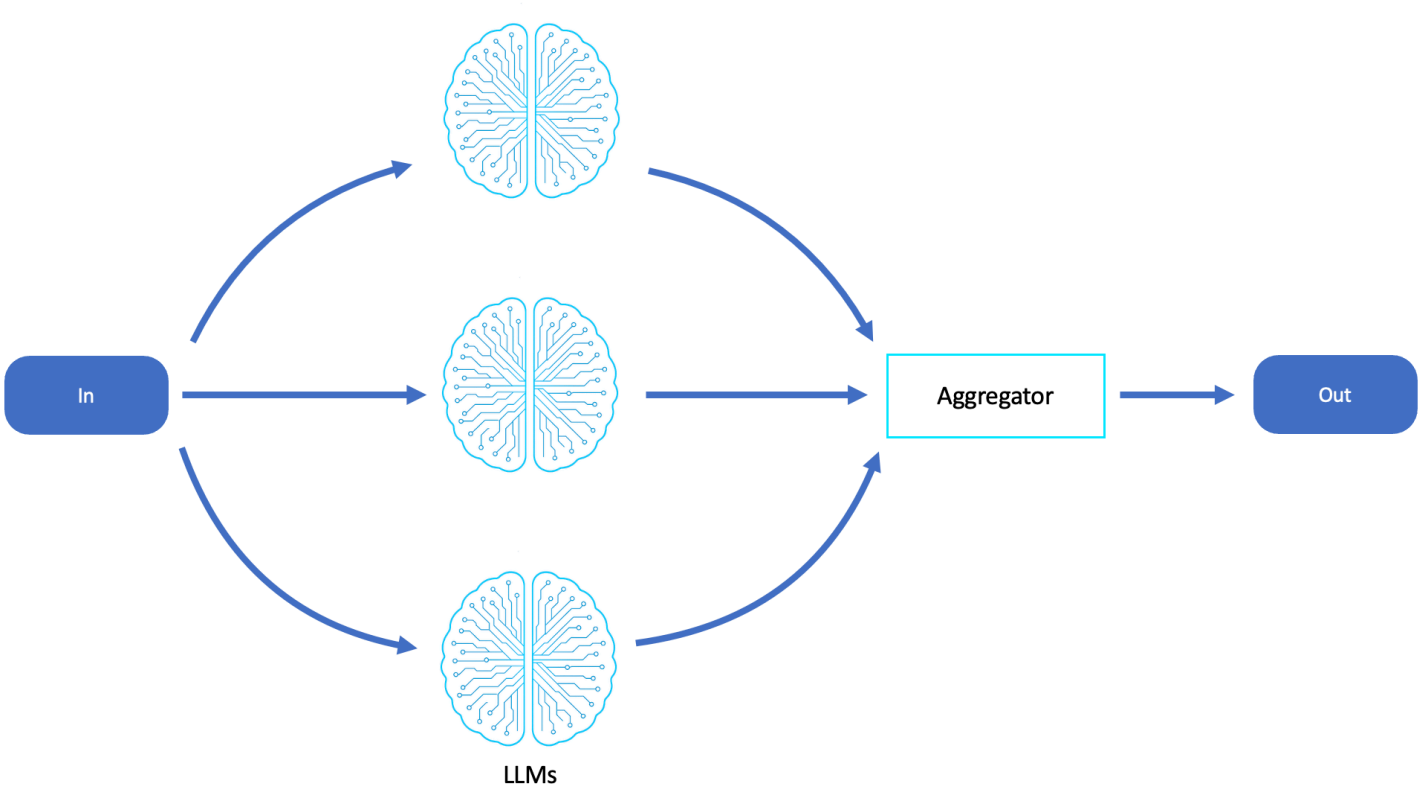
- Broader input understanding
- Intelligent fallback and tool selection
- Natural extensibility through new agent roles or prompt styles

Agentic routing replaces rigid rules with dynamic cognitive dispatching, which allows systems to evolve with language rather than code.

Parallelization and scatter-gather patterns

Many advanced reasoning and generation tasks – such as summarizing large documents, evaluating multiple solution paths, or comparing diverse perspectives – benefit from the parallel execution of prompts. Traditional sequential workflows fall short when scalability, responsiveness, and fault tolerance are required. To overcome this, LLM-based parallelization can be reimagined using an event-driven scatter-gather pattern, where tasks are dynamically fanned out to autonomous agents and results intelligently synthesized.

The following diagram is an example of an LLM parallelization workflow:



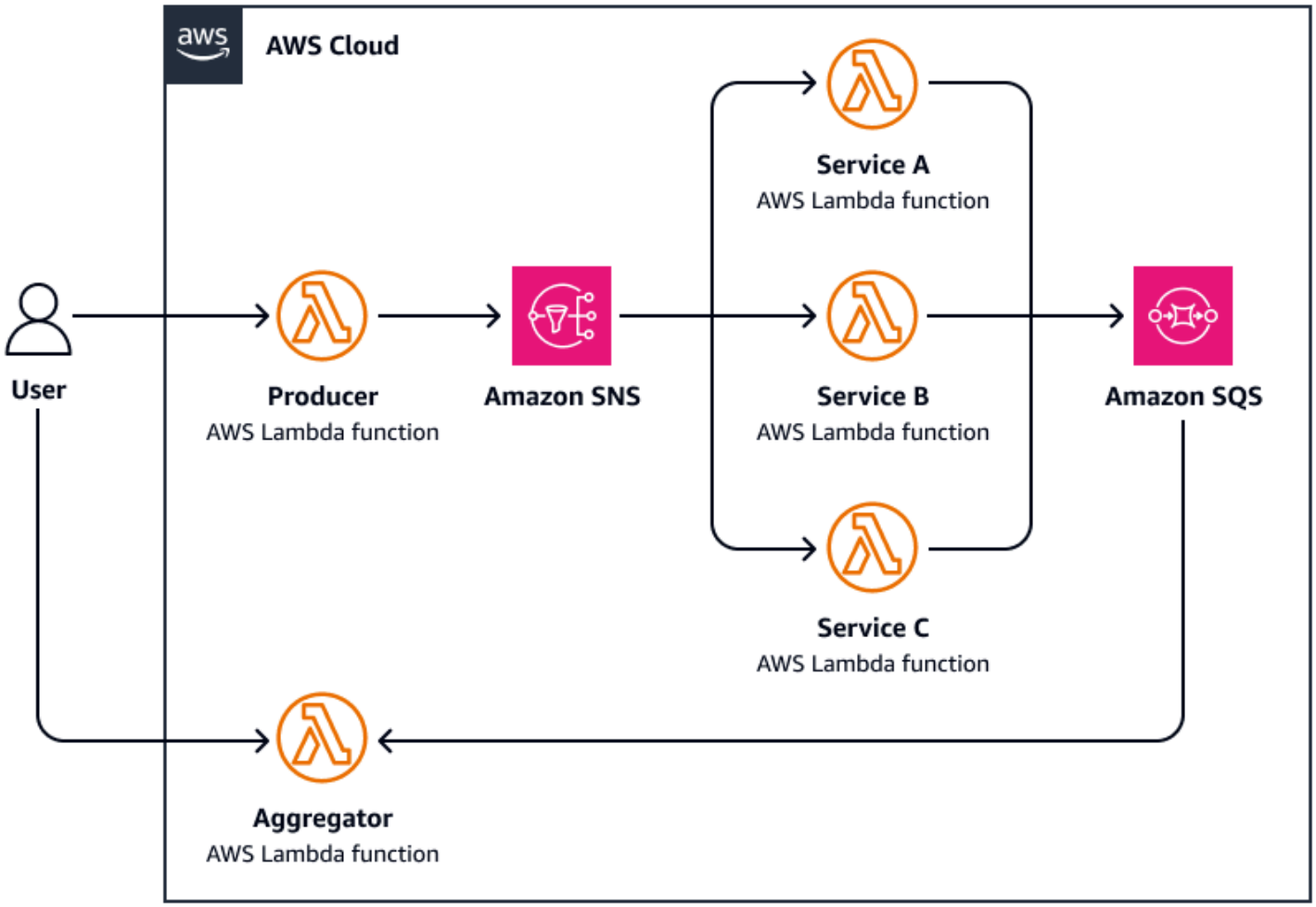
Scatter-gather

In distributed systems, a scatter-gather pattern sends tasks to multiple services or processing units in parallel, waits for their responses, and then aggregates results into a consolidated output. Unlike fan-out, scatter-gather is coordinated because it expects responses and usually applies logic to combine, compare, and select results.

Common implementations for parallelization and scatter-gather include the following:

- AWS Step Functions map a state for parallel task execution
- AWS Lambda with concurrency, coordinating results from multiple invoked functions
- Amazon EventBridge with correlation IDs and aggregation workflows
- Custom controller pattern to manage fan-out and gather results by using Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, or queues

The following diagram is an AWS architectural example of scatter-gather:



1. A user sends a request to a central coordinator function that scatters the task by publishing parallel messages to an Amazon Simple Notification Service (Amazon SNS) topic.
2. Each message includes task metadata and is routed to a specialized worker AWS Lambda.
3. Each worker AWS Lambda independently processes its assigned subtask (for example, querying an external API, processing a document, and analyzing data).
4. Results are written to a common storage layer, such as Amazon Simple Queue Service (Amazon SQS).
5. The aggregator function waits for all responses to be completed, and then it does the following:
   - Gathers and aggregates the results (for example, merges summaries, selects best matches)
   - Sends a final response or triggers a downstream workflow

Common use cases for scatter-gather patterns include the following:

- Federated search
- Price comparison engines
- Aggregated data analysis
- Multimodel inference

LLM-based parallelization (scatter-gather cognition)

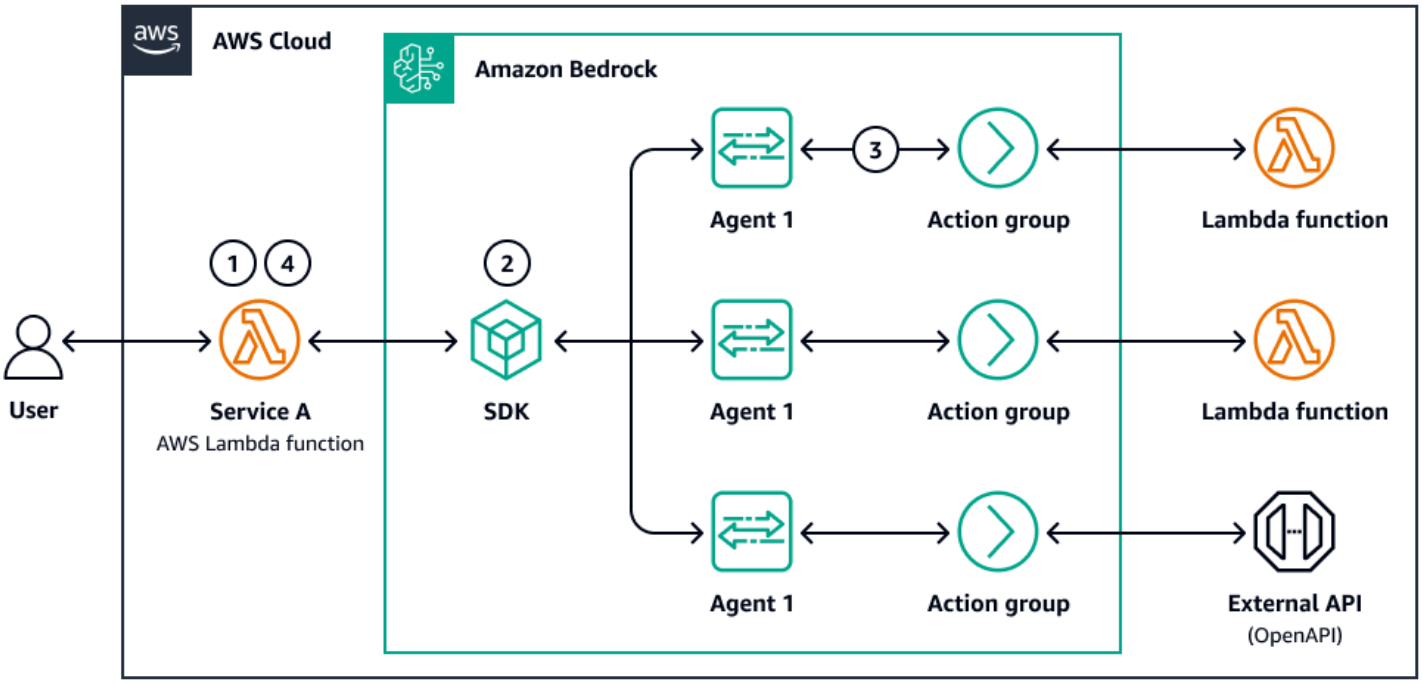
In agentic systems, parallelization closely mirrors scatter-gather by distributing subtasks across multiple LLM calls or agents, each independently reasoning through a portion of the problem. Returned results are gathered and synthesized by an aggregation process, which is often another LLM or controller agent.

Agent parallelization

1. An agent submits a request "Summarize insights across these 10 reports."
2. It scatters the reports to 10 parallel LLM summarization tasks.
3. When it returns all summaries, the agent does the following:
   - Aggregates summaries into a unified briefing
   - Identifies themes or contradictions
   - Sends the synthesized output to the user

This agentic workflow enables scalable, modular, and adaptive parallel reasoning. This is ideal for use cases that require high cognitive throughput.

The following diagram is an AWS architectural example of agent parallelization:



1. A user submits a multipart query or document set.
2. A controller AWS Lambda or step function distributes the subtasks. Each task invokes an Amazon Bedrock LLM call or subagent with its own prompt.
3. When the calls and subtasks are complete, results are stored (for example, in Amazon S3 or memory store), and an aggregation step merges, compares, or filters the outputs.
4. The system returns the final response to the user or downstream agent.

This system has a distributed reasoning loop with traceability, fault tolerance, and optional result weighting or selection logic.

Takeaways

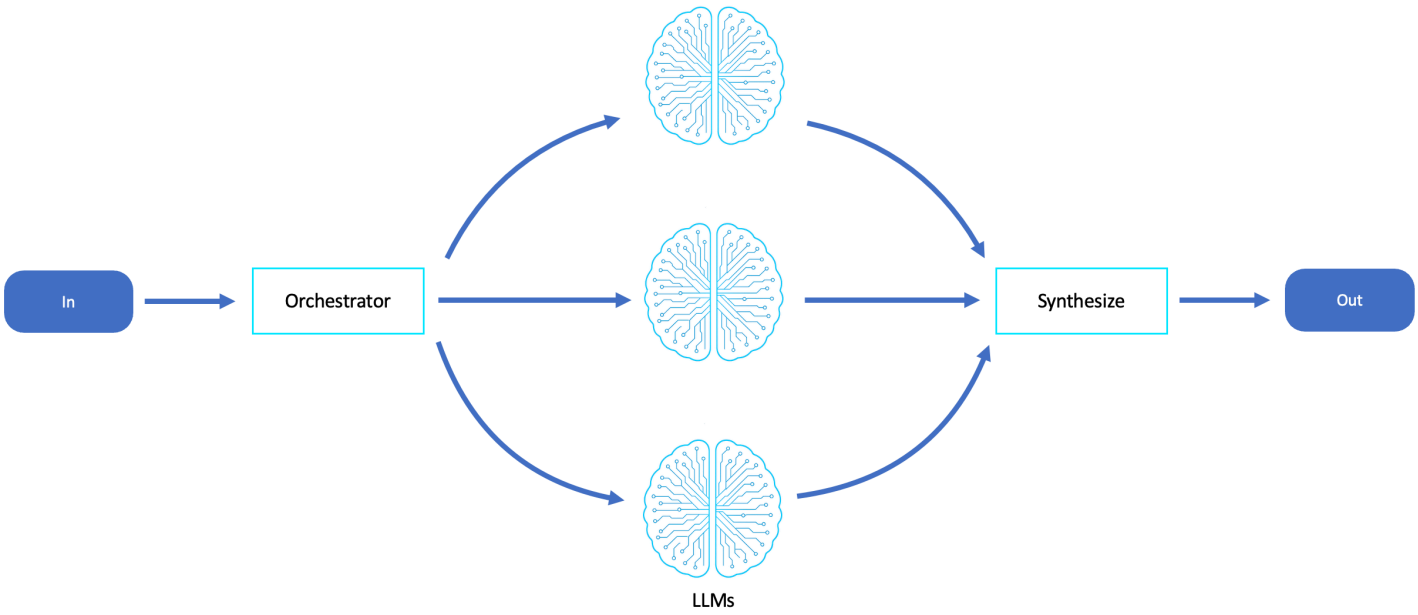
Agentic parallelization uses scatter-gather patterns to distribute LLM tasks, enabling parallel processing and intelligent result synthesis.

Saga orchestration patterns

As workflows driven by LLMs become increasingly complex, spanning prompt chains, data processing steps, tool invocations, and agent collaboration, the need for intelligent orchestration becomes essential. Rather than relying on tightly coupled scripts or static predetermined execution

flows, these workflows can be implemented as event-driven orchestration patterns, enabling LLM-based systems to dynamically coordinate, monitor, and adapt multi-step tasks across autonomous agents.

The following diagram is an example of an orchestrator:



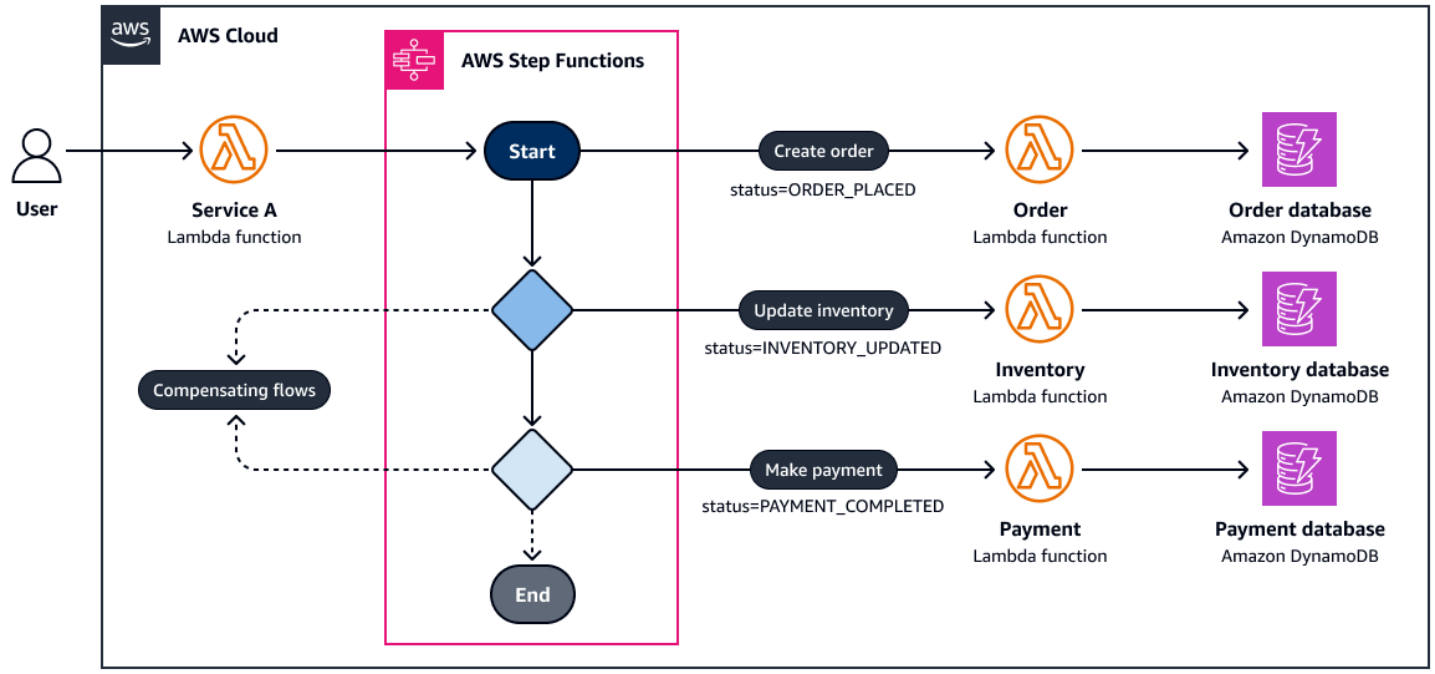
Event orchestration

In traditional distributed systems, event orchestration refers to a pattern in which a central coordinator manages a complex workflow by explicitly directing the flow of control across multiple services or tasks. Unlike event choreography (where each service reacts independently), orchestration provides centralized logic, visibility, and control over the entire process.

This is typically implemented using the following tools:

- **AWS Step Functions** – Define and execute stateful workflows
- **AWS Lambda** – Carry out discrete tasks within the orchestrated flow
- **Amazon SQS** or **Amazon EventBridge** – Triggers asynchronous steps or responses

The following diagram is an AWS architectural example of saga orchestration:



An AWS Step Functions workflow manages a customer order process:

1. Create order (AWS Lambda)
2. Update inventory (AWS Lambda)
3. Make payment (AWS Lambda)

The orchestrator coordinates each step by managing retries, parallel branches, timeouts, and failures.

Role-based agent system (orchestrator)

In agentic systems, the orchestrator pattern mirrors event orchestration but distributes the logic across multiple reasoning agents, each with a defined role or specialization. A central orchestrator agent interprets the overall task, decomposes it into subtasks, and delegates those to worker agents, each optimized for a particular domain (for example, research, coding, summarization, review).

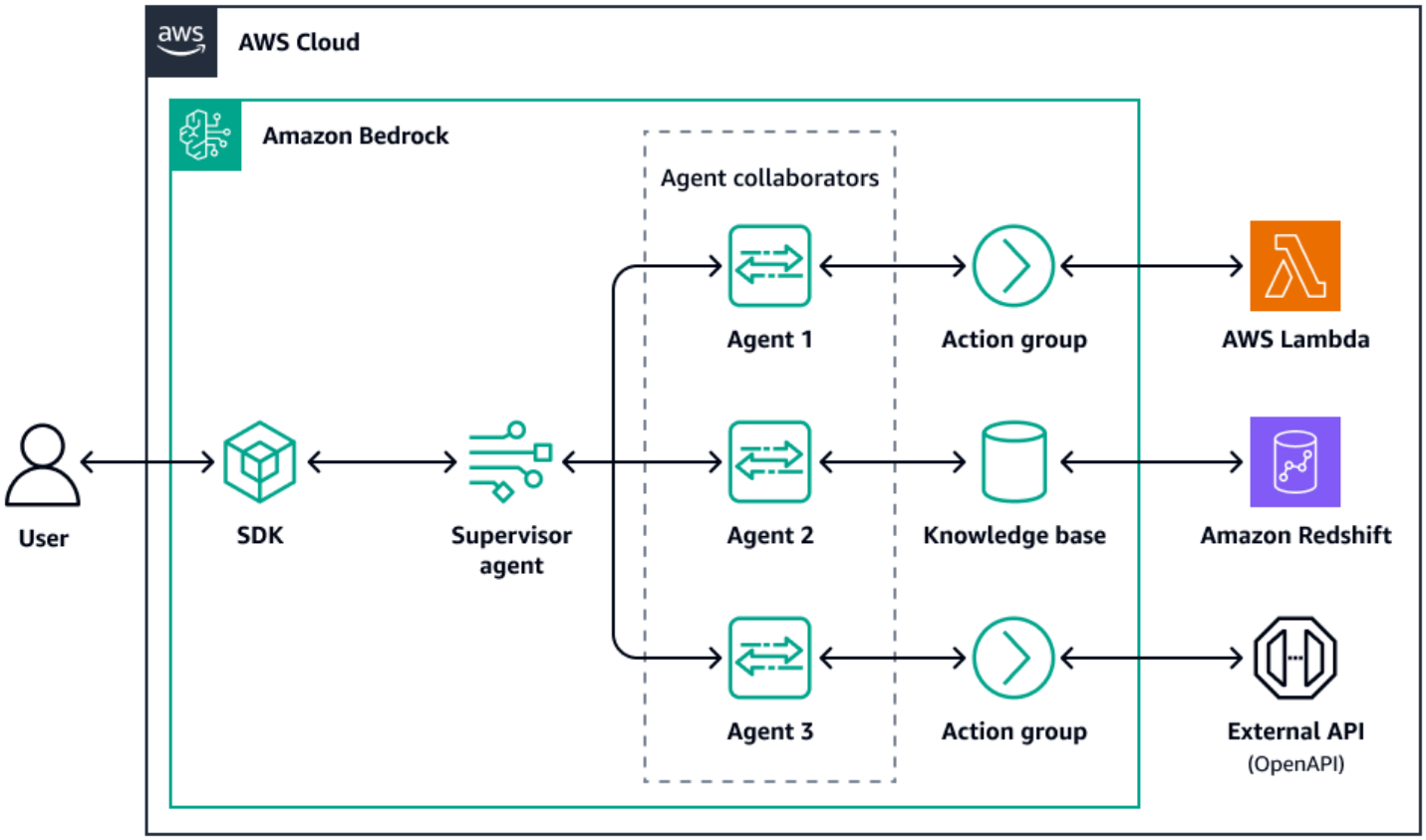
Supervisor

1. A user submits the query "Create a project brief and summarize the top 5 competitors."
2. The orchestrator agent does the following:

- Assigns a research agent to find competitor data
- Sends the raw findings to a summarization agent
- Passes results to a brief-writer agent
- Compiles the final output for the user

Each agent operates independently, but the orchestrator coordinates the tasks. This is like a Lambda function that handles workflow tasks.

The following diagram an AWS architectural example of a supervisor:



1. A user submits a task to an Amazon Bedrock supervisor agent.
2. The supervisor agent parses the request into subtasks for each agent collaborator.
3. Each subtask is assigned to a collaborator agent with role-specific prompts or toolchains.
4. Worker agents call external APIs or tools through an action group.
5. Each worker agent returns the output in a structured format.
6. When all workers return their results, the supervisor evaluates, synthesizes, and returns the final response.

This structure allows for modularity, adaptability, and introspection across complex multistep agent workflows.

Takeaways

Where event orchestration uses centralized control (for example, AWS Step Functions) to direct service execution, role-based agent systems use an LLM-powered orchestrator agent to reason about the goal, delegate subtasks to worker agents, and synthesize the final output.

In both paradigms, the orchestrator does the following:

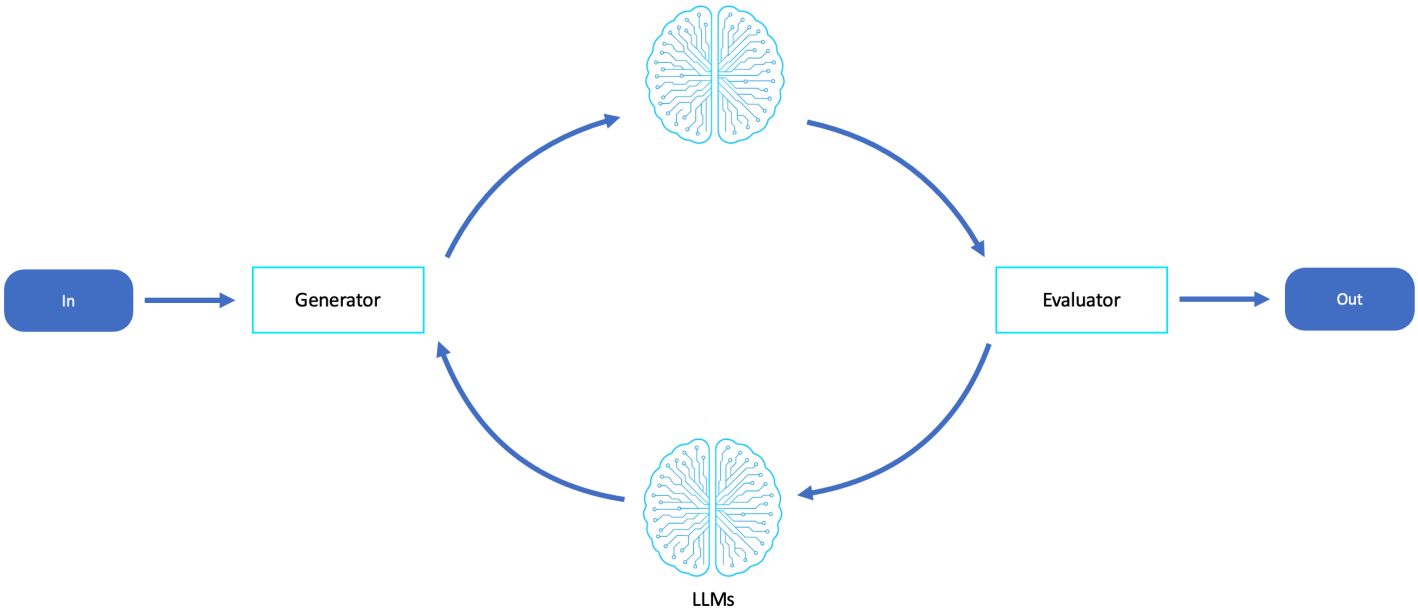
- Maintains context and execution flow
- Handles branching, sequencing, and error handling
- Produces a unified result from distributed components

Agentic orchestration, however, adds reasoning, adaptability, and semantic delegation. This makes it well-suited to open-ended, ambiguous, and evolving tasks.

Evaluator reflect-refine loop patterns

Tasks such as code generation, summarization, or autonomous decision-making benefit greatly from runtime feedback, enabling the system to evolve through observation and refinement. To operationalize this, the reflect–refine cycle can be implemented as an event-driven feedback control loop – a pattern inspired by systems engineering, adapted for autonomous, intelligent workflows.

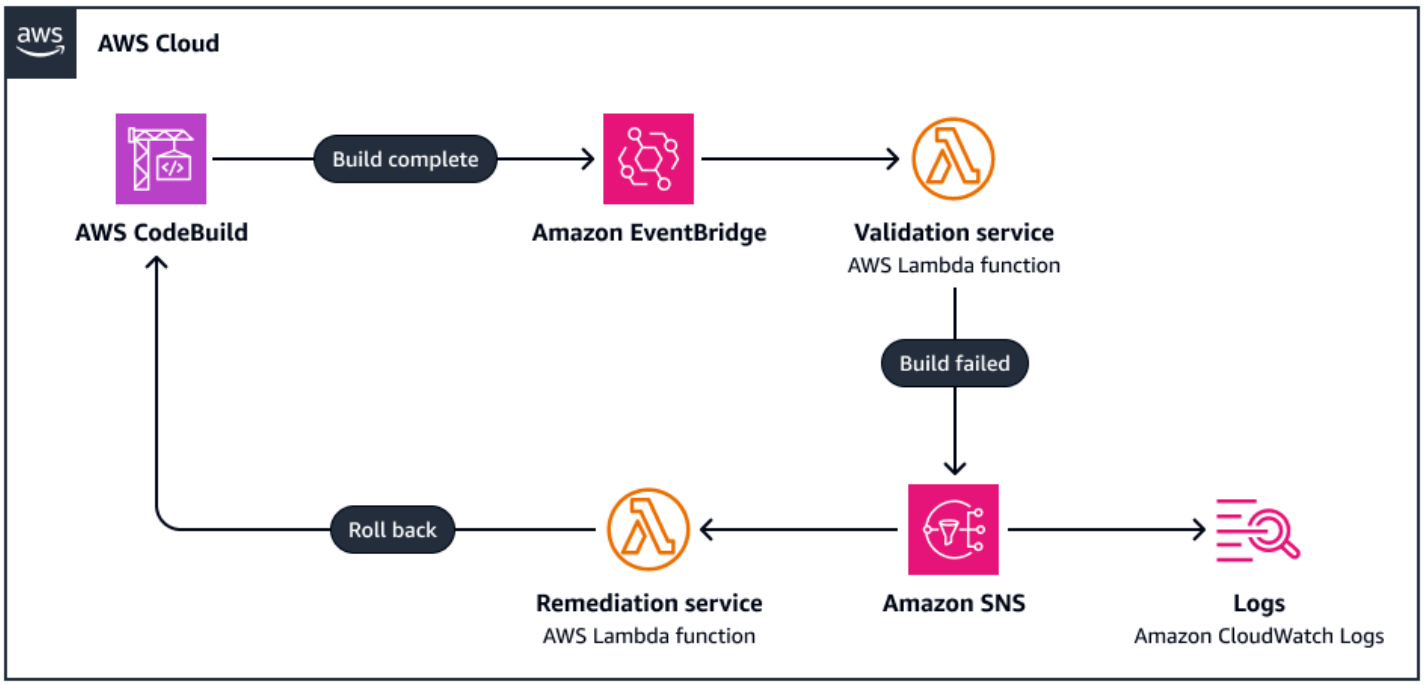
The following diagram is an example of an evaluator reflect-refine feedback loop:



Feedback control loop

A feedback control loop is a pattern that monitors its own outputs and behaviors, evaluates them against defined criteria or a desired state, and then adjusts its actions accordingly. This architecture is inspired by control theory and is foundational in domains such as automation, continuous integration and continuous delivery (CI/CD) pipelines, and machine learning operations.

The following diagram is an AWS architectural example of a feedback control loop:



1. A deployment pipeline emits a buildComplete event.
2. The event triggers an automated test or evaluation job that validates the build.
3. If validation fails (for example, due to failing tests, security issues, or a policy violation), the system:
   - Emits a buildComplete event
   - Logs the issue or sends a notification
   - Triggers a remediation or corrective action, such as rollback, patching, or retry

The loop continues until it produces an acceptable outcome or escalation, or a time out occurs. This pattern is commonly used for the following:

- Amazon EventBridge rules to route events to evaluation or remediation tasks
- AWS Step Functions for iterative retry logic and branching on evaluation outcomes
- Amazon Simple Notification Service (Amazon SNS) or Amazon CloudWatch alarms for feedback triggers and alerts
- AWS Lambda functions or containerized workers to apply corrective actions

Feedback control loop (evaluator)

An evaluator workflow is a cognitive feedback loop that's powered by LLMs or reasoning agents. The process consists of the following:

1. A generator agent or LLM produces an output (for example, a plan, answer, or draft).
2. An evaluator agent reviews the result using a critique prompt or evaluation rubric.
3. Based on the feedback, the original agent or a new optimizer agent revises the output.

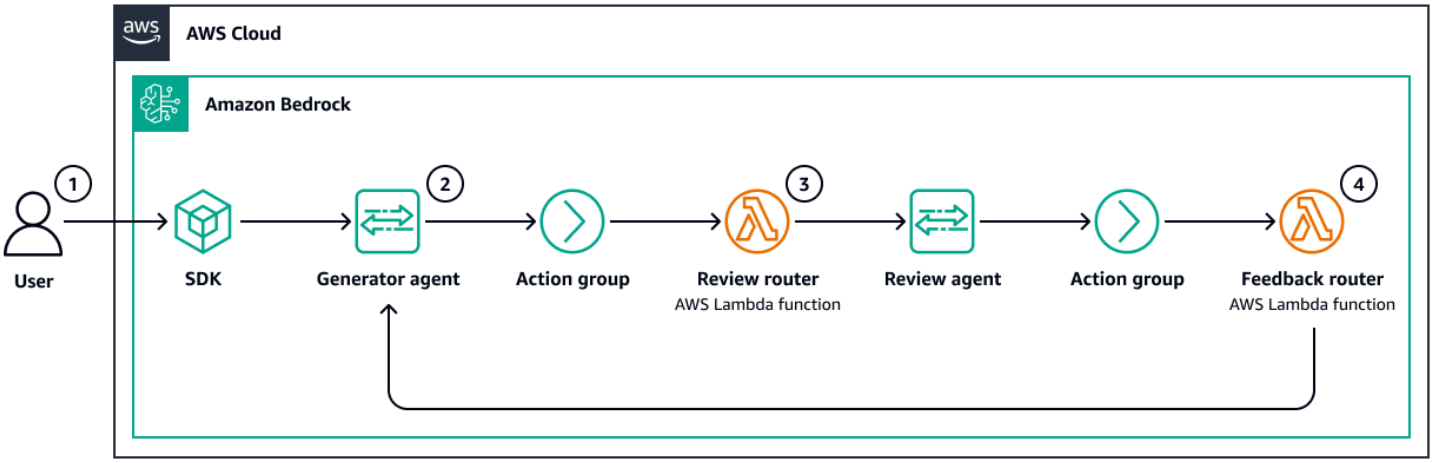
The loop repeats until the result meets a set of criteria, is approved, or reaches a retry limit.

Evaluator

1. A user asks an agent to write a policy summary.
2. The generator agent drafts it.
3. An evaluator agent checks coverage, tone, and legal correctness.
4. If the response is inadequate, it's refined and resubmitted until the feedback loop converges.

This enables self-assessment, iterative refinement, and adaptive output control—all without human input.

The following diagram is an AWS architectural example of a feedback control loop (evaluator):



1. A user issues a task (for example, draft a business strategy).
2. An Amazon Bedrock agent generates an initial draft using an LLM.
3. A second agent (or a follow-up prompt) performs a structured evaluation (for example, "rate this output by clarity, completeness, and tone").
4. If the rating falls below a threshold, the response is revised by:
   - Reinvoking the generator with an embedded critique
   - Sending the feedback to a specialized refiner agent
   - Iterating until an acceptable response is reached

Optional components like AWS Lambda controllers or AWS Step Functions can manage feedback thresholds, retries, and fallback strategies.

Takeaways

Where traditional feedback control loops use events, metrics, and remediation logic to validate and adjust system behavior, agentic evaluator loops use reasoning agents to evaluate, reflect, and revise output dynamically.

In both paradigms:

- Output is evaluated after it's generated
- Corrective or refining actions are triggered based on feedback

- System continuously adapts toward a target quality or goal

The agentic version transforms static validation into semantic reflection, enabling self-improving agents that evaluate their own effectiveness.

Designing agentic workflows on AWS

Each pattern in this guide can be built using AWS services. Amazon Bedrock agents provide orchestration, data access, and interaction channels.

Component	AWS service	Purpose
LLM reasoning	Amazon Bedrock	Agent logic, planning, tool use
Tool execution	AWS Lambda, Amazon ECS, Amazon SageMaker	Host external tools for agents
Memory and RAG	Amazon Bedrock knowledge base, Amazon S3, OpenSearch	Persistent and semantic memory
Orchestration	AWS Step Functions	Multistep task and agent coordination
Event routing	Amazon EventBridge, Amazon SQS	Decoupled interagent messaging
User interface	Amazon API Gateway, AWS AppSync, SDK	Entry points for applications or systems
Monitoring	Amazon CloudWatch, AWS X-Ray, AWS Distro for OpenTelemetry	Observability and agent introspection

Conclusion

Agentic workflow patterns are the next evolutionary stage of event-driven architectures, wherein business logic is not statically defined but dynamically reasoned through using large language

model (LLM)–enhanced cognition. By combining traditional cloud-native primitives with LLM workflows and agent design patterns, organizations can build adaptive, intelligent, and modular systems that respond with purpose and learn from experience.

In these patterns, Amazon Bedrock is the gateway to agentic cognition, allowing LLM-based agents to access event workflows, interact with tools and memory, and deliver structured, traceable, and aligned results.

As you design and deploy agentic systems, these workflow patterns provide blueprints for building autonomous, composable AI architectures. These systems are grounded in serverless best practices and augmented with intelligent foundation models.

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	July 14, 2025