Agentic AI patterns and workflows on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Agentic AI patterns and workflows on AWS

Table of Contents

Agentic AI patterns and workflows on AWS

Aaron Sempf and Andrew Hooker, Amazon Web Services

July 2025 (document history)

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

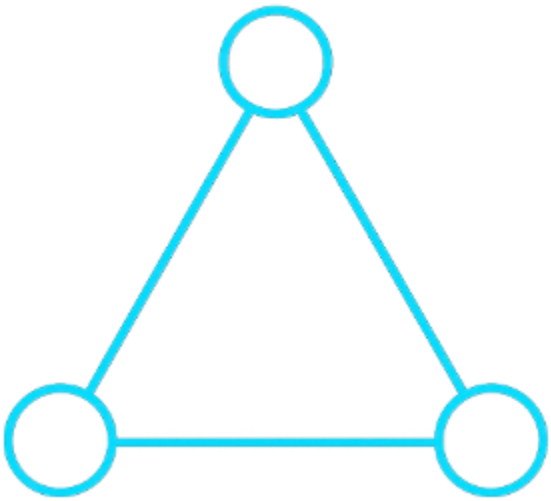
For more information about this content series, see Agentic AI.

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.

In this section

- Computer-use agents
- Coding agents
- Speech and voice agents
- Workflow orchestration agents
- Memory-augmented agents
- Simulation and test-bed agents
- Observer and monitoring agents
- Multi-agent collaboration

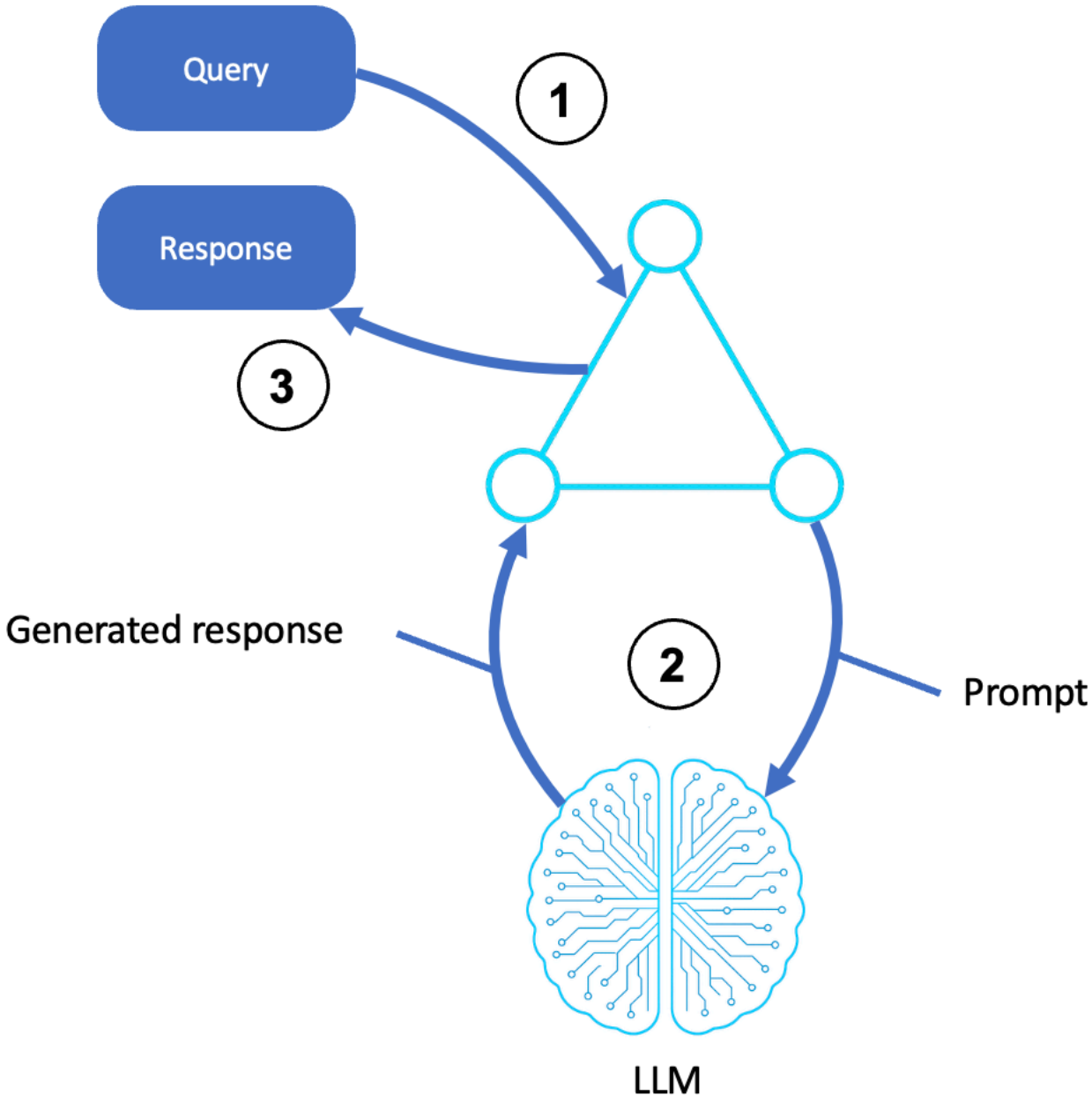
Basic reasoning agents

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

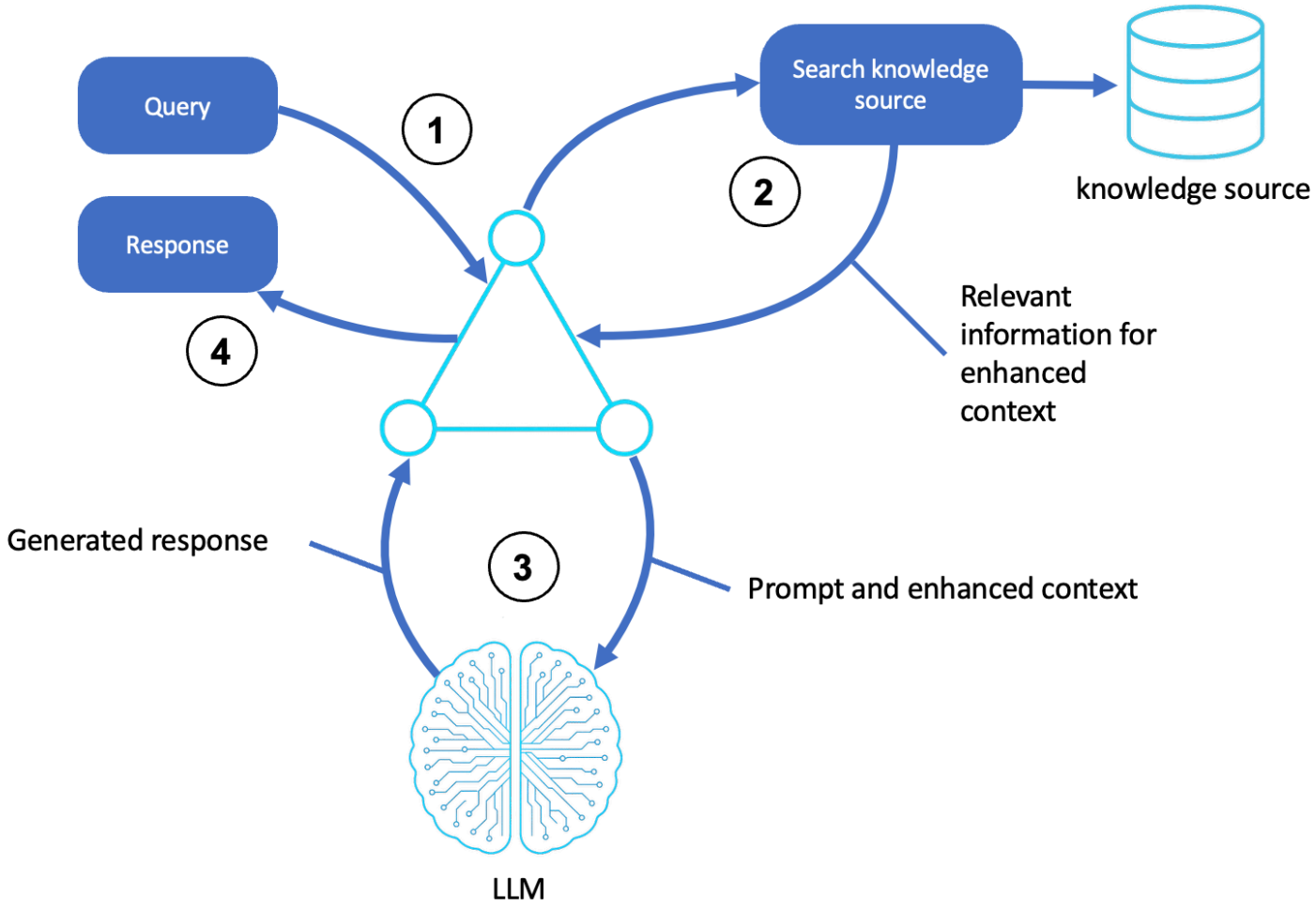
The basic reasoning agent is foundational because of its simple structure. It has core capabilities that turn goals into reasoning paths that lead to intelligent outputs. This pattern is often a starting point for advanced patterns, such as tool-based agents and agents that use retrieval-augmented generation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



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.

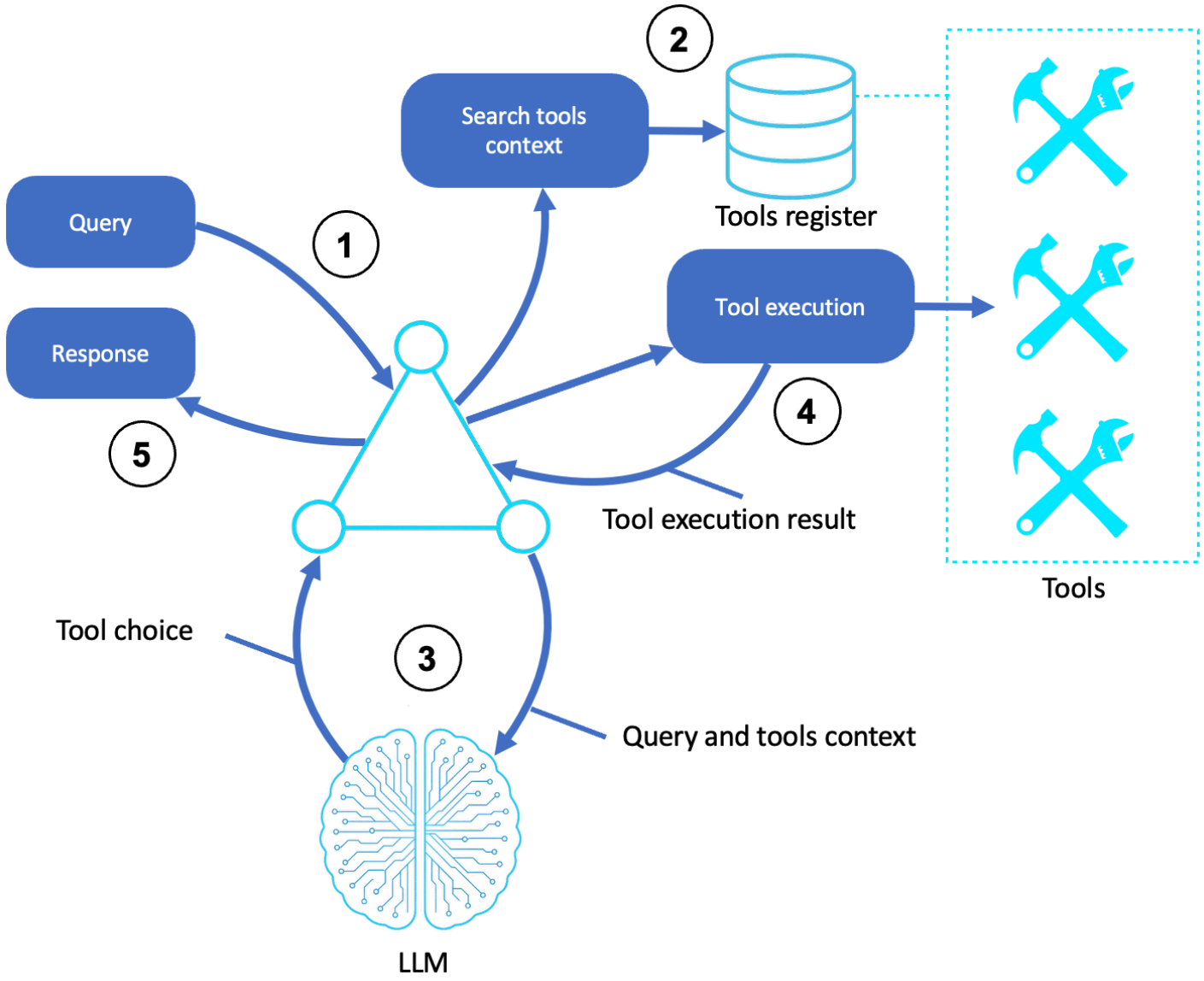
Tool-based agents for calling functions

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

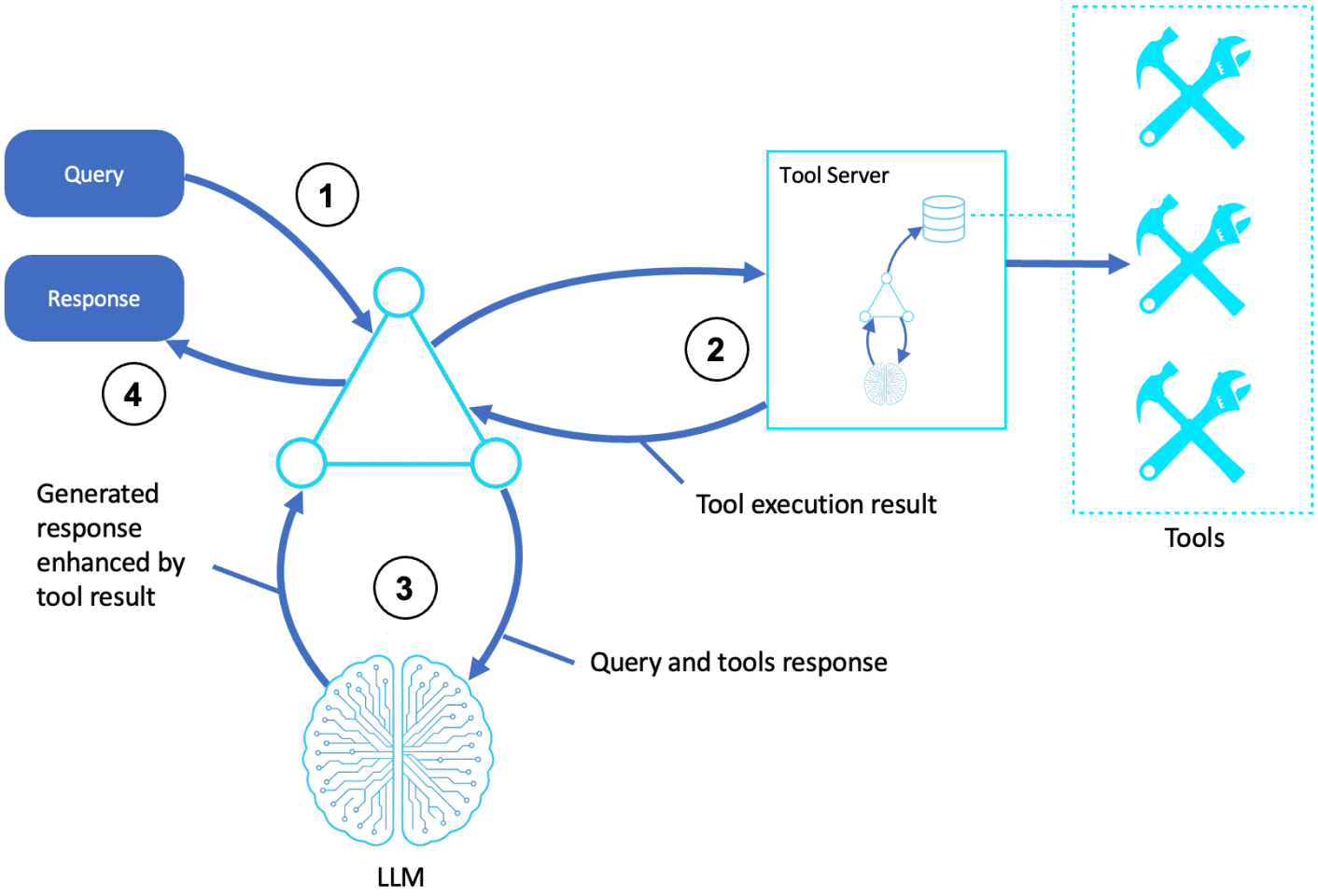
Tool-based function-calling agents represent a shift from understanding language to performing actions. These agents invoke dynamic, context-aware tools while maintaining LLM reasoning, transforming passive assistants into systems that complete tasks, access services, and integrate business operations. 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.

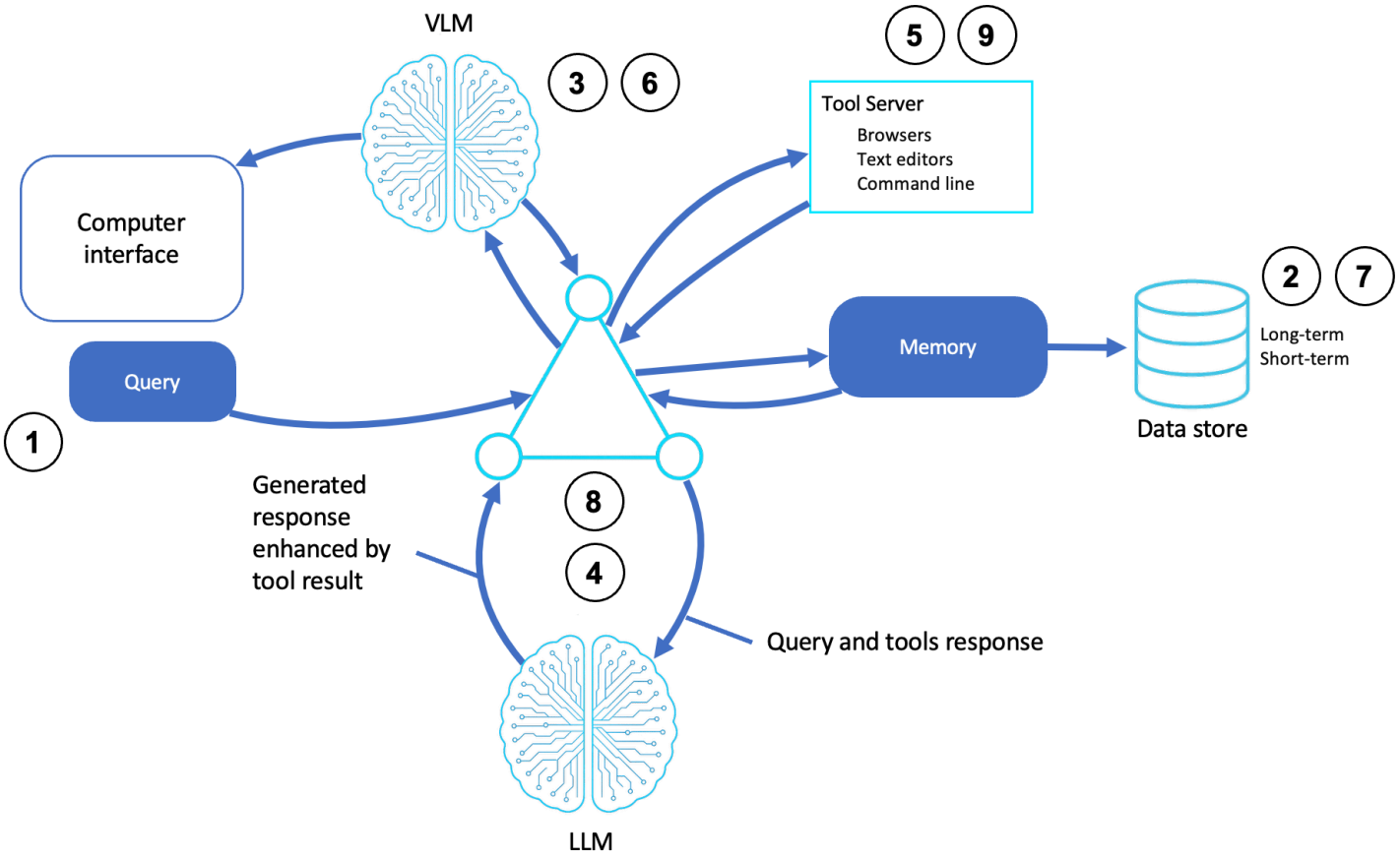
Computer-use agents

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.

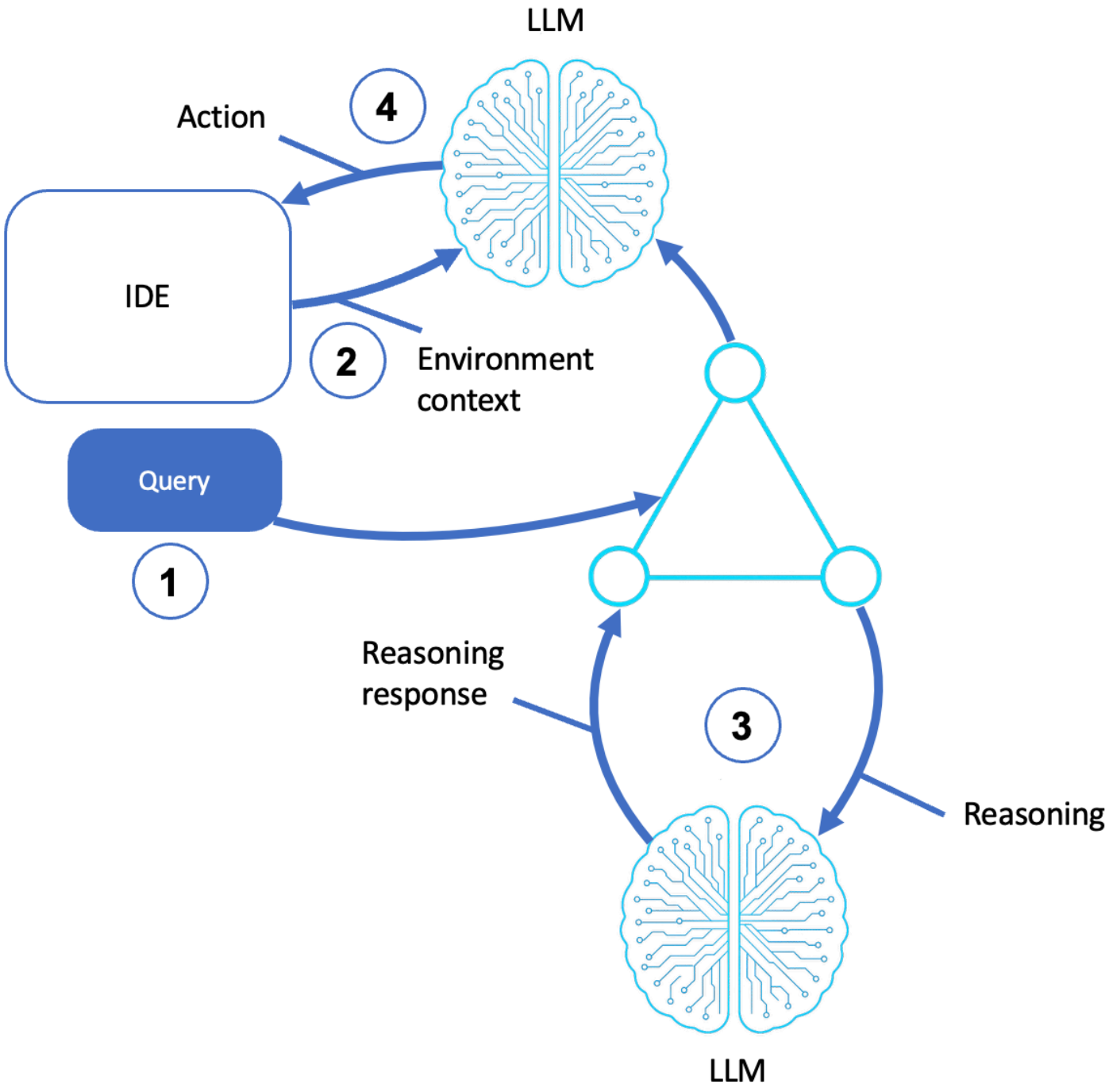
Coding agents

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.

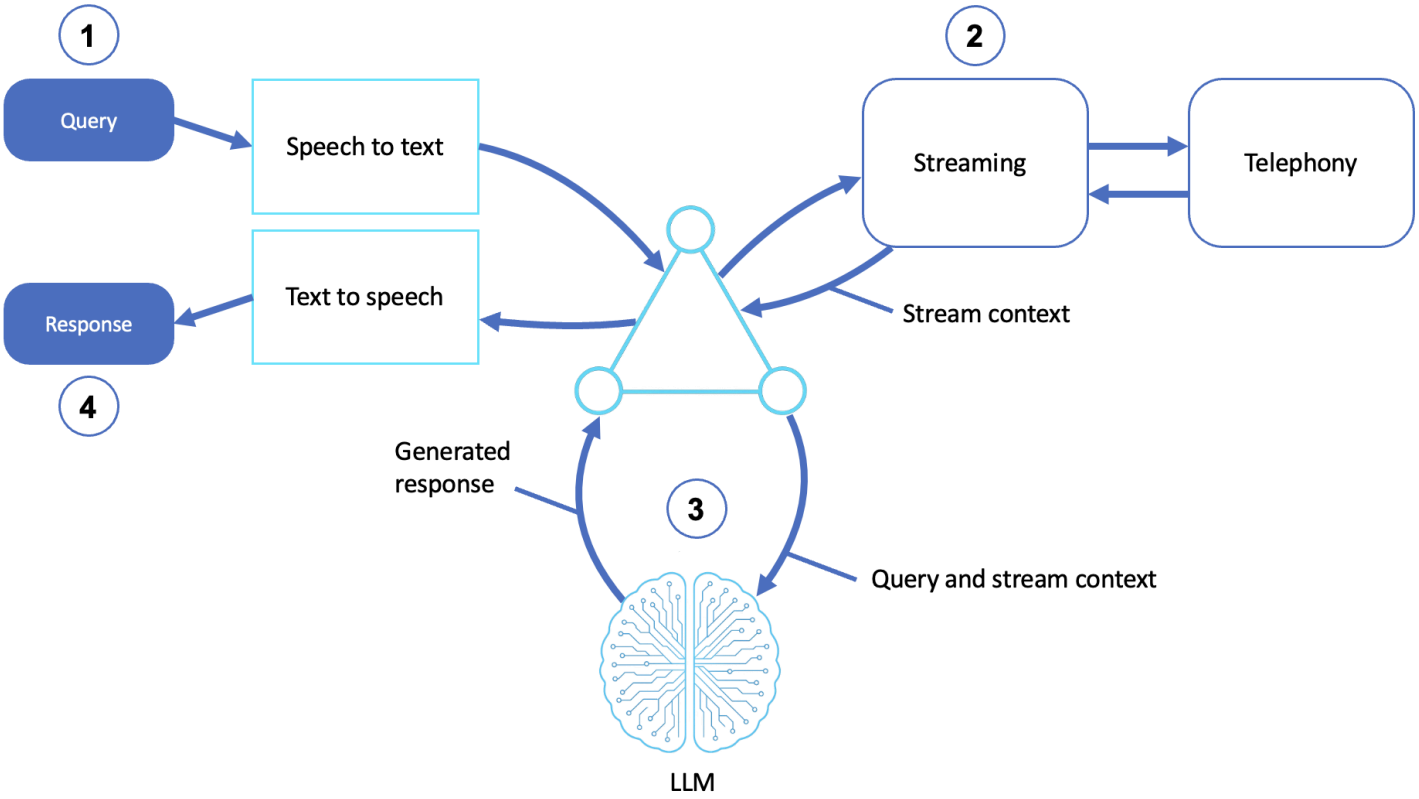
Speech and voice agents

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, 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.

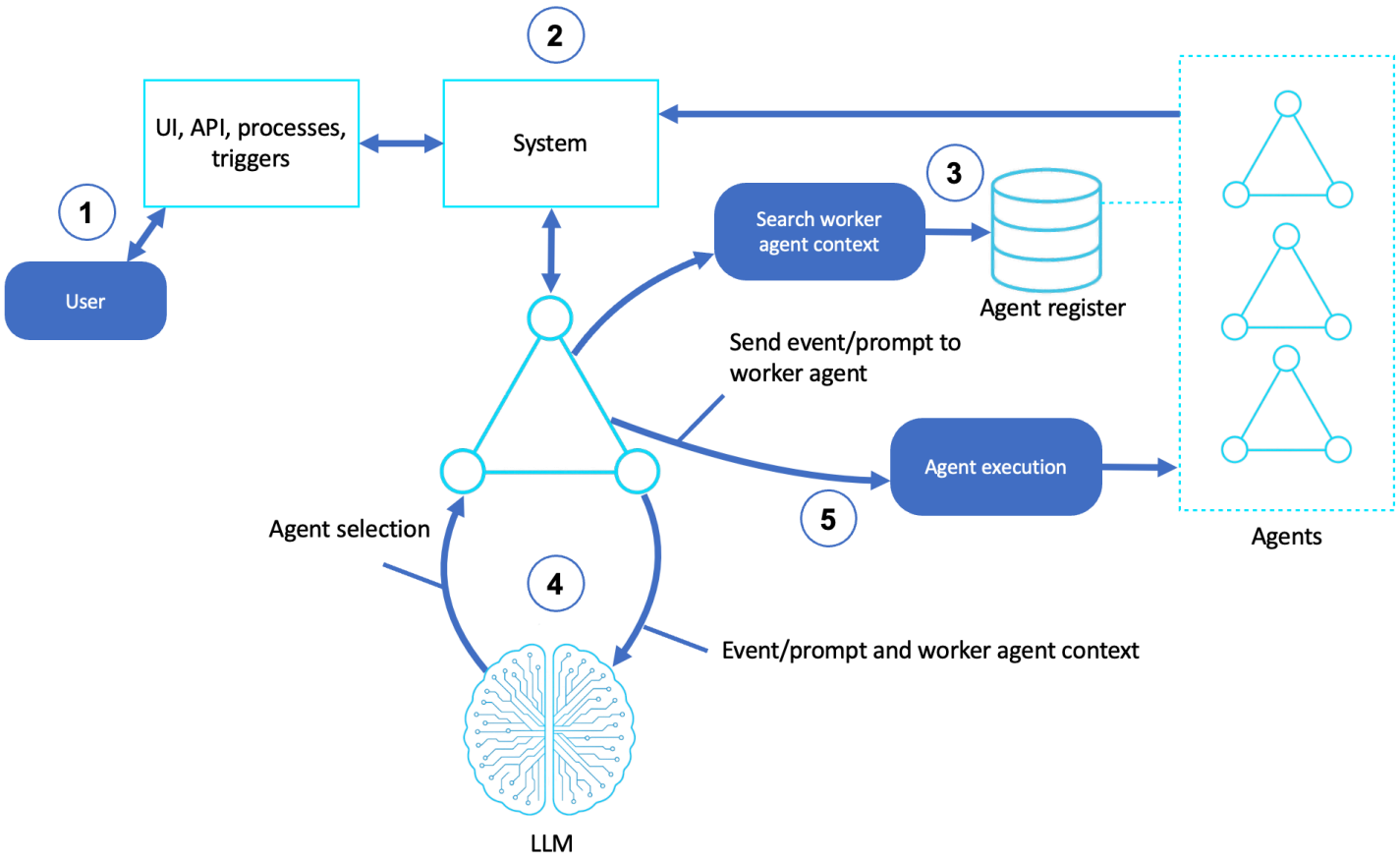
Workflow orchestration agents

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.

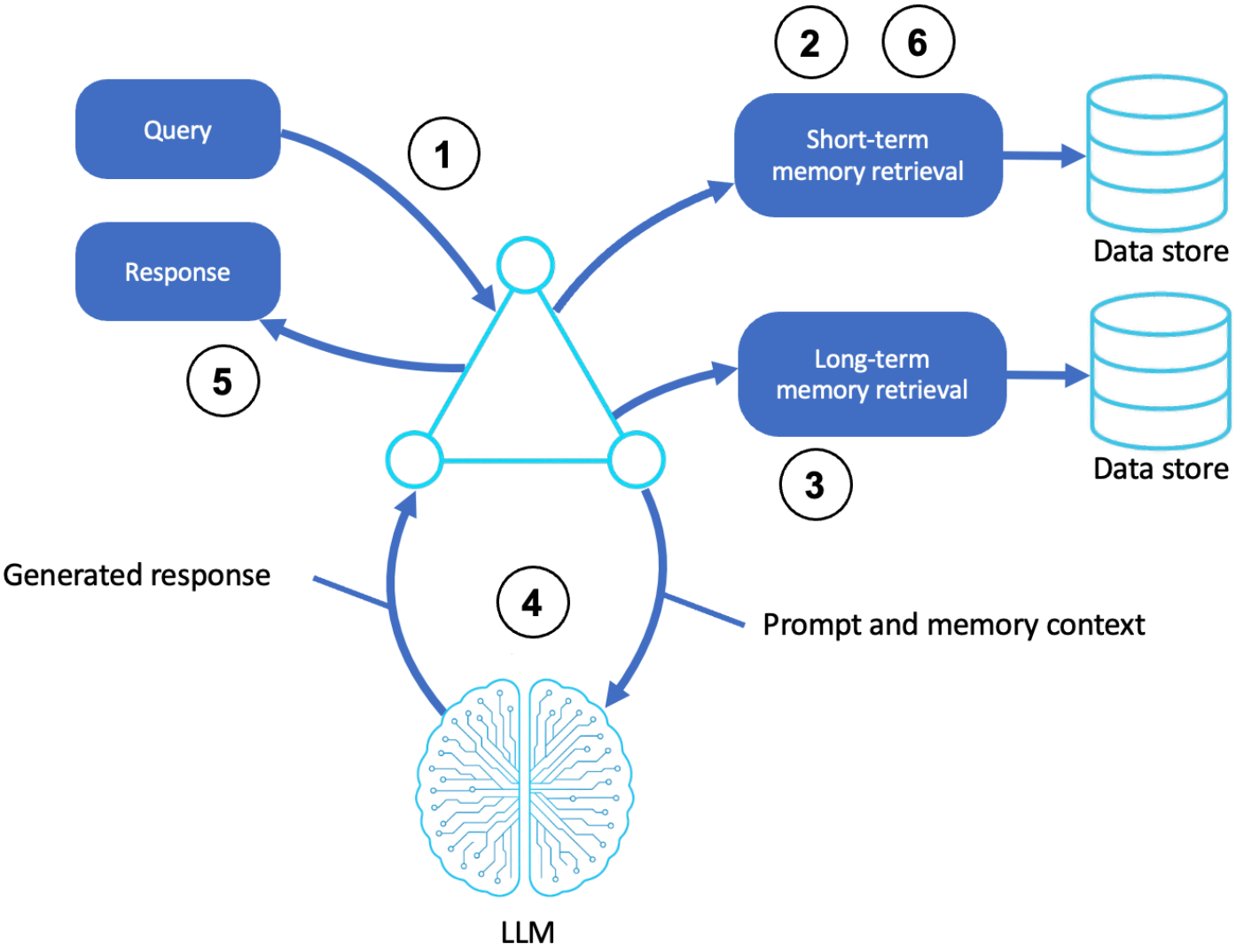
Memory-augmented agents

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, PostgreSQL, 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.

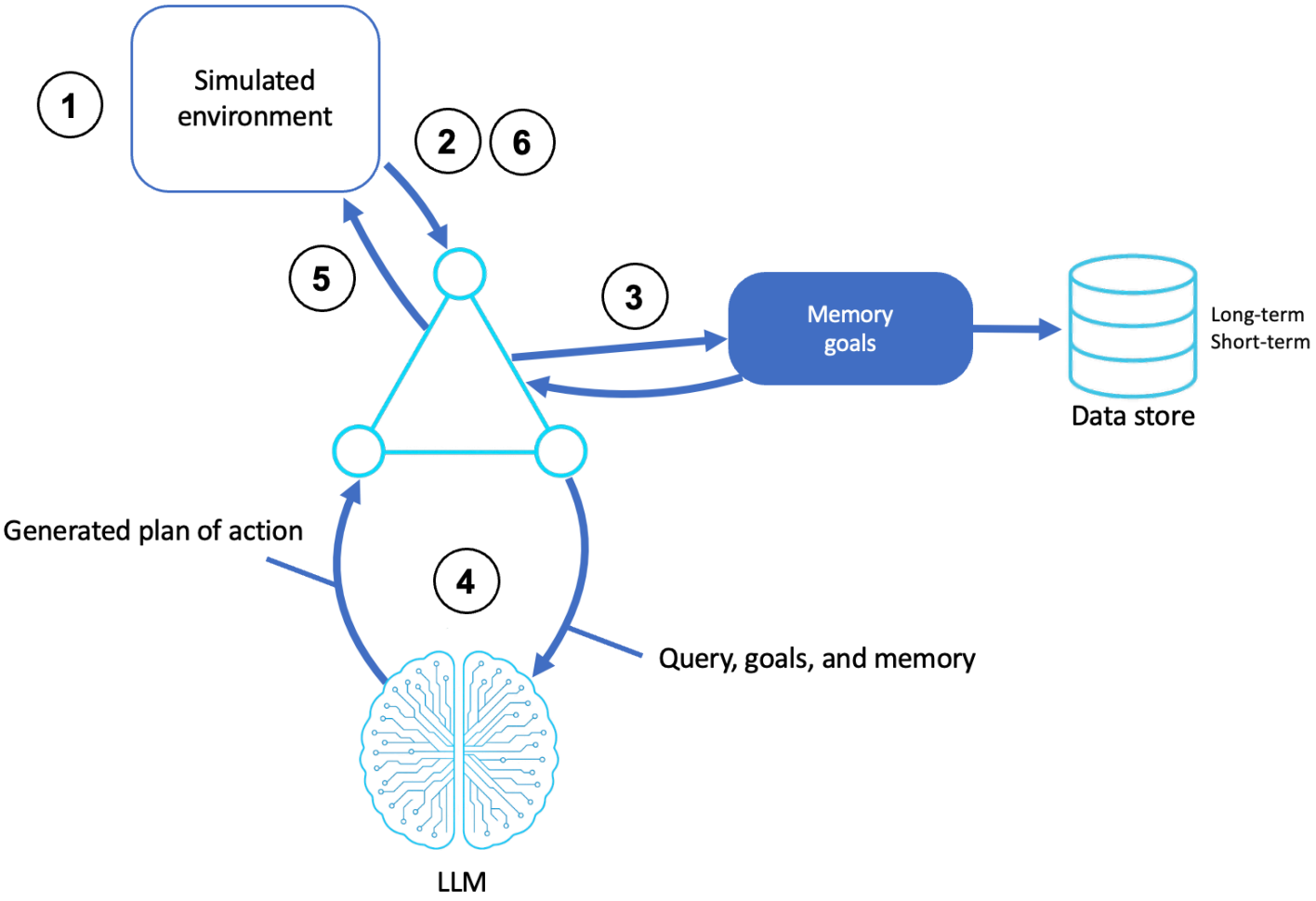
Simulation and test-bed agents

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	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 reinforce ment 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

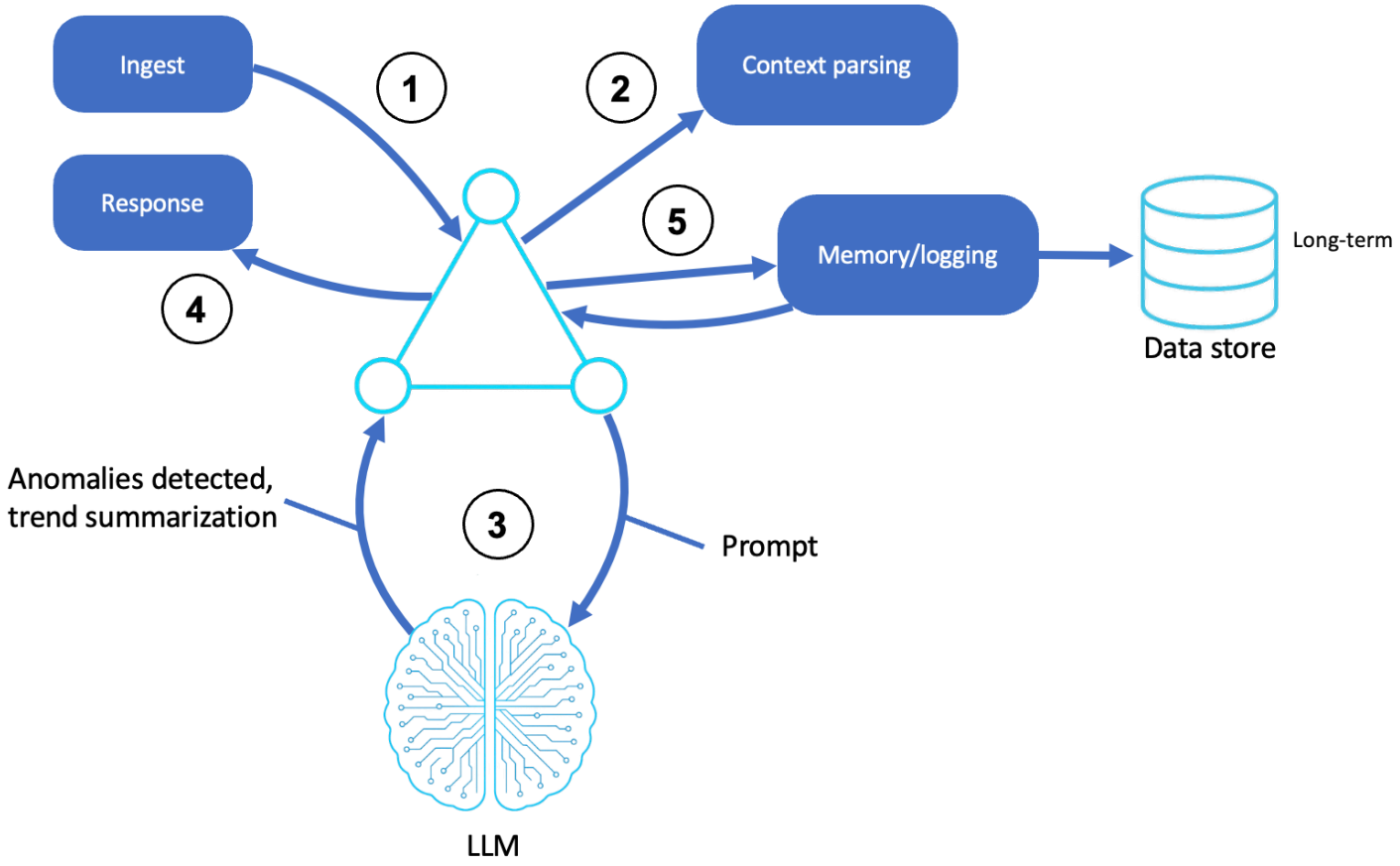
Simulation and test-bed agents are for structured exploration prior to being deployed to production systems. Use these agents to train autonomous navigation policies, test business processes in synthetic environments, and evaluate swarms for coordination patterns.

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



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
- Amazon Quick Suite 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

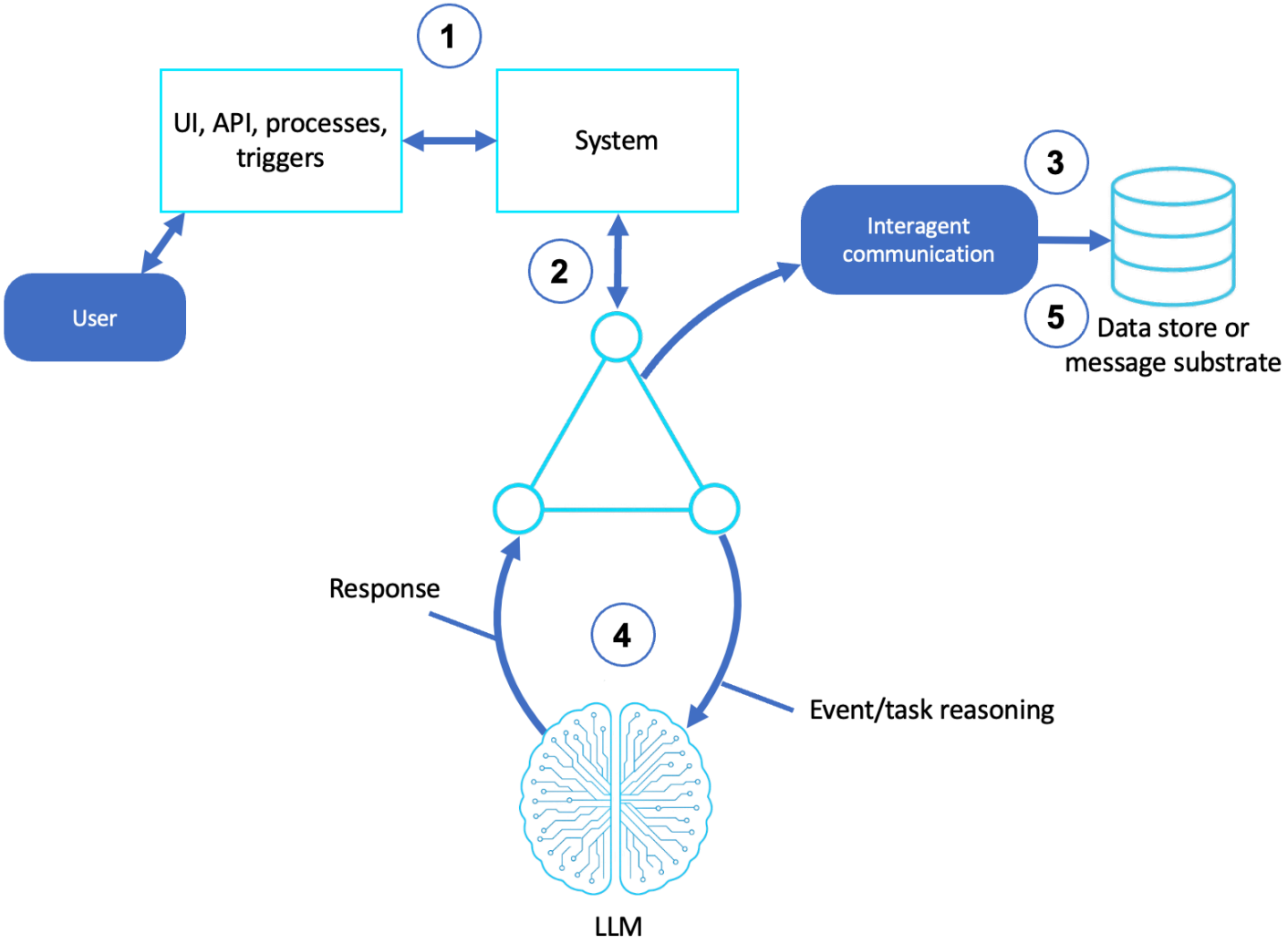
Multi-agent collaboration refers to a pattern in which multiple autonomous agents, each with a distinct role, specialization, or objective, negotiate to solve complex tasks. These agents may operate independently or with other agents by sharing information, dividing responsibilities, and collectively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

In this section

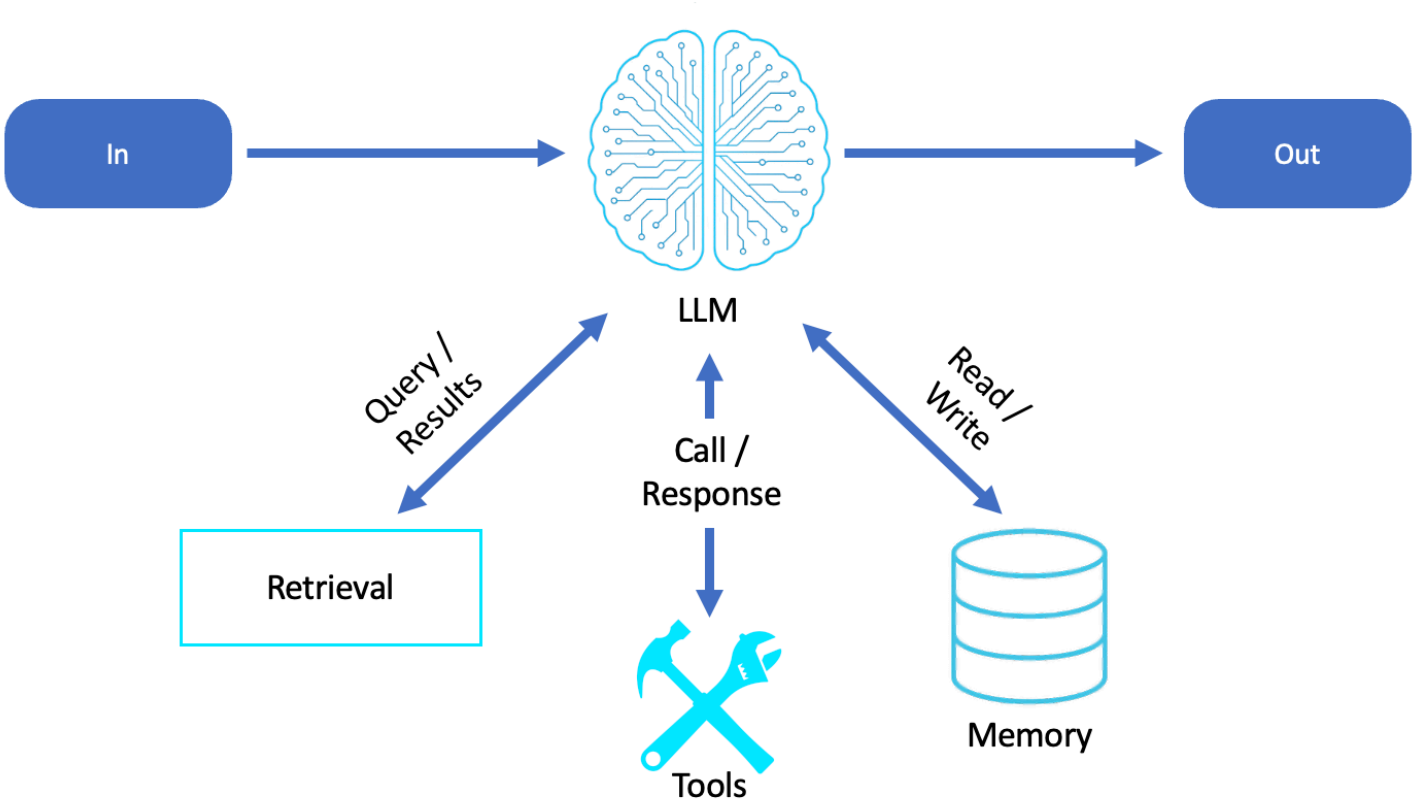
Overview of LLM-augmented cognition

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

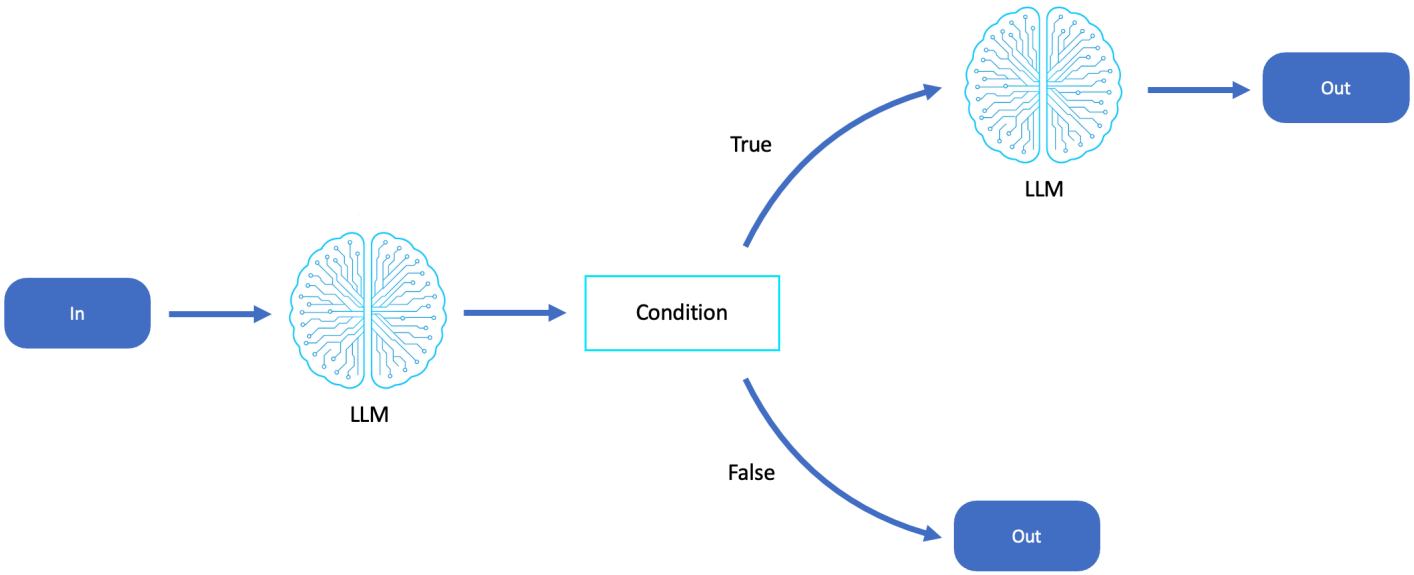
- **Memory** – Incorporating persistent or a session-based state into the reasoning loop, either by using structured databases or contextual summaries

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

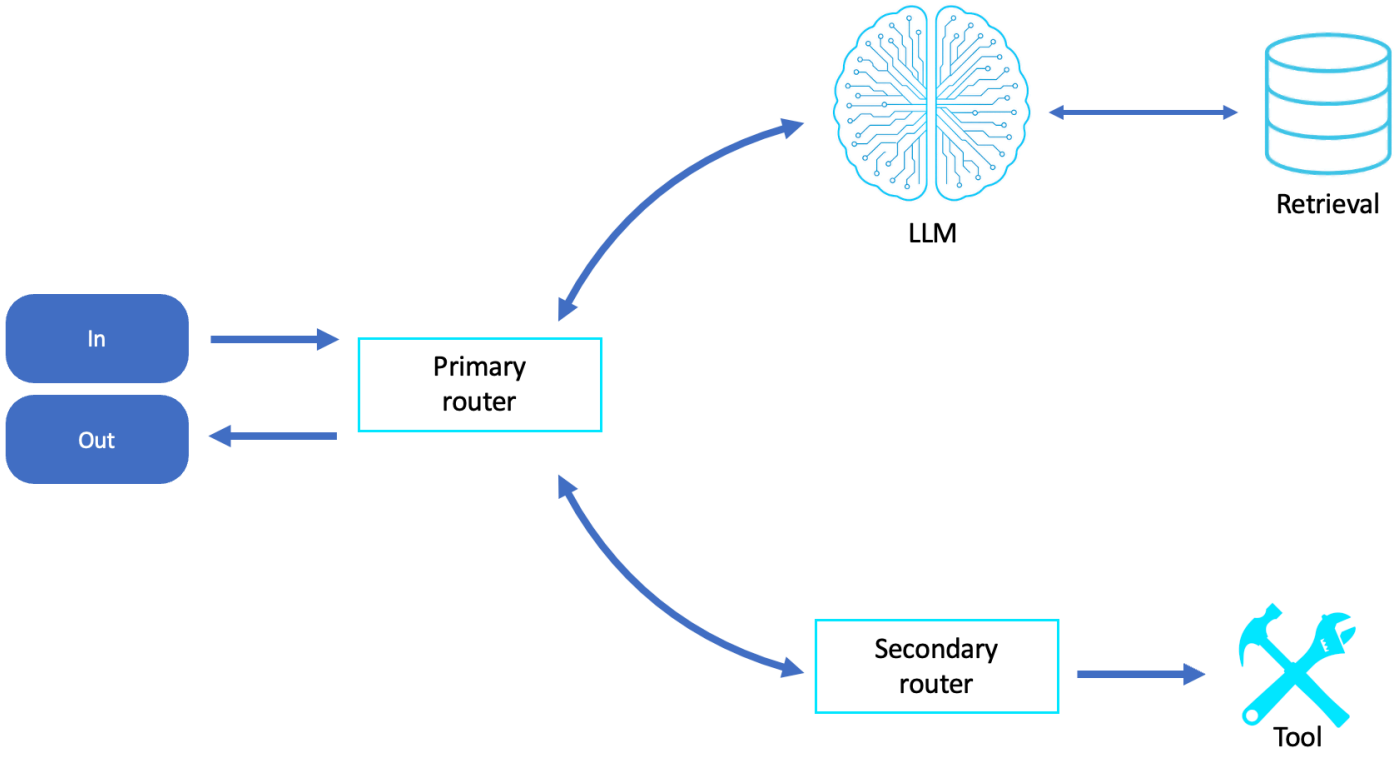
- Linear or branching chains of LLM calls
- Intermediate results passed as structured input or embedded into follow-up prompts
- Can be orchestrated with AWS Step Functions, AWS Lambda, or agent-specific runners

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

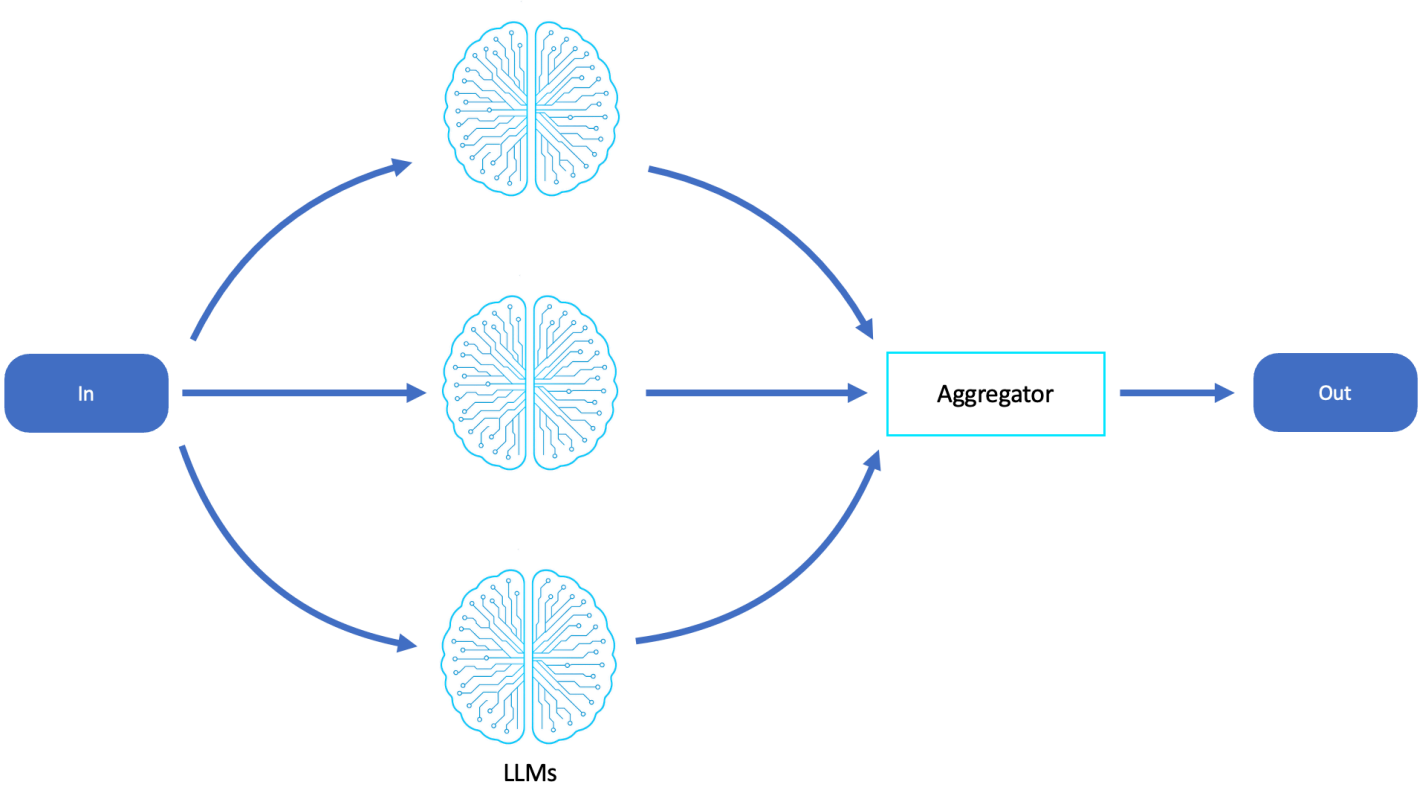
- A first-pass LLM acts as a dispatcher
- Routes can invoke distinct workflows or even other agent patterns
- Supports modular expansion of capabilities

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

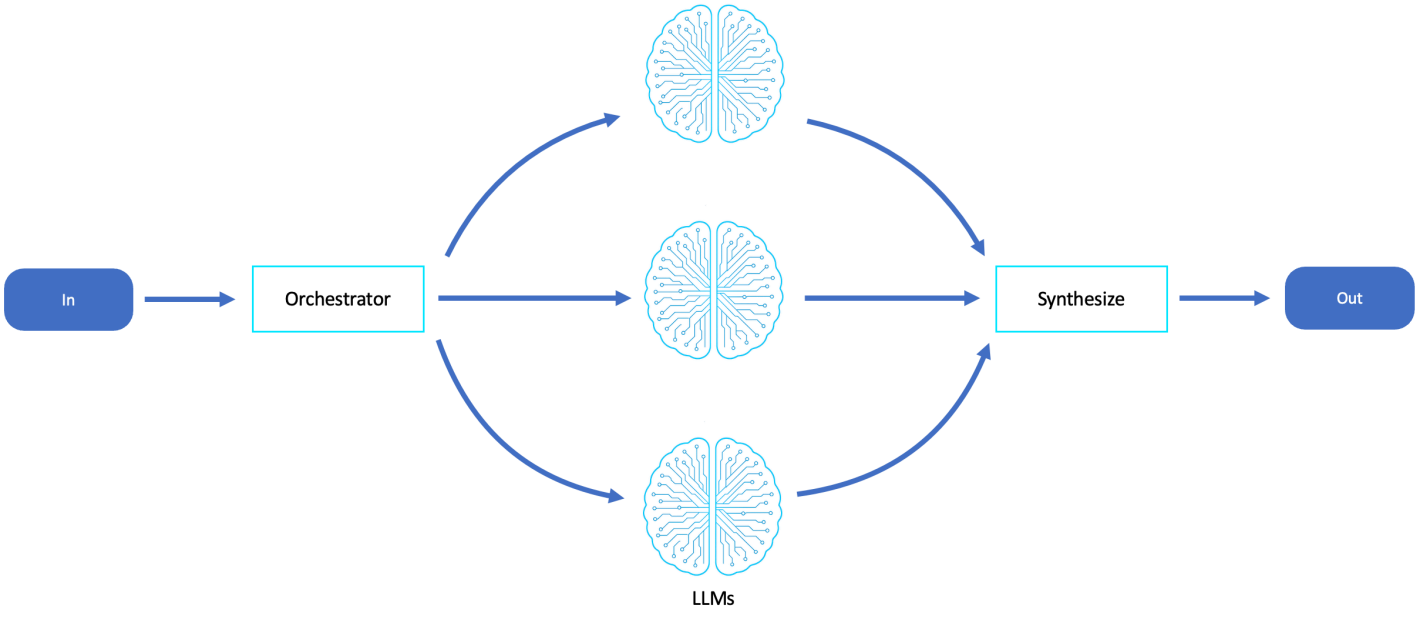
- Parallel execution of LLM tasks (by using AWS Lambda, AWS Fargate, or an AWS Step Functions map state)
- Requires result alignment, validation, or deduplication at the synthesis stage
- Well-suited for stateless agent loops

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

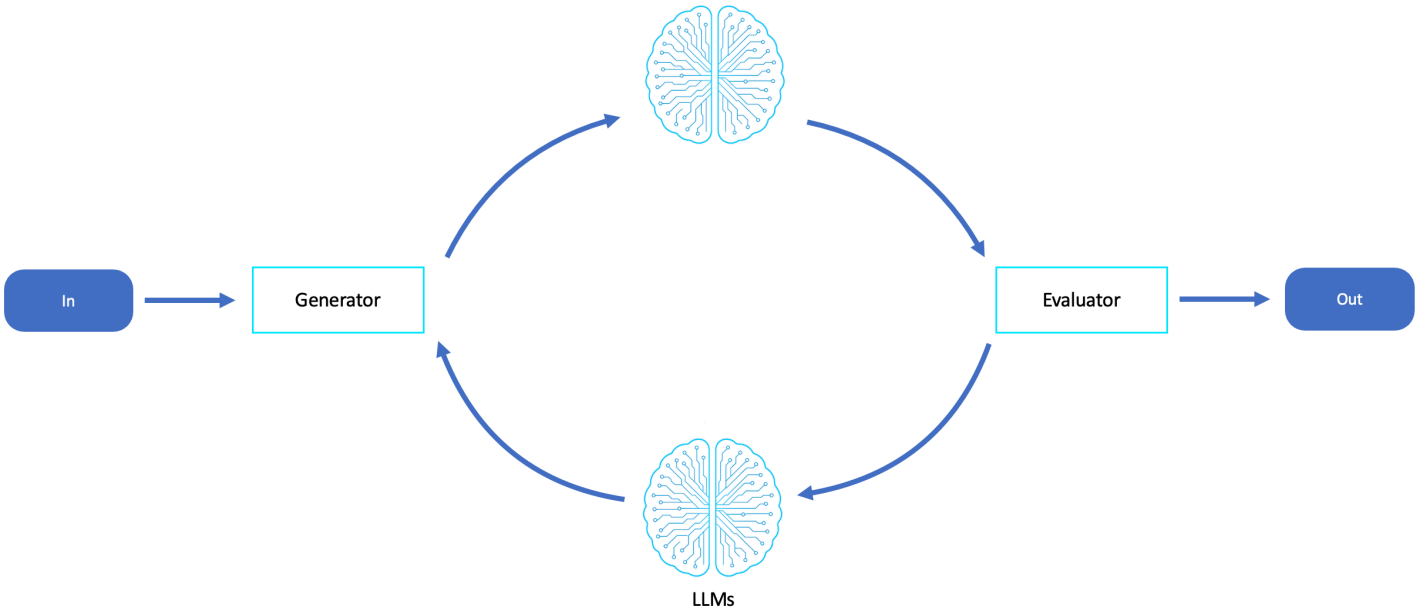
- Orchestrator performs goal meta-reasoning
- Worker agents may include tool access, memory, or domain-specific prompting
- Can be hierarchical (that is, multilevel task delegation)

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.

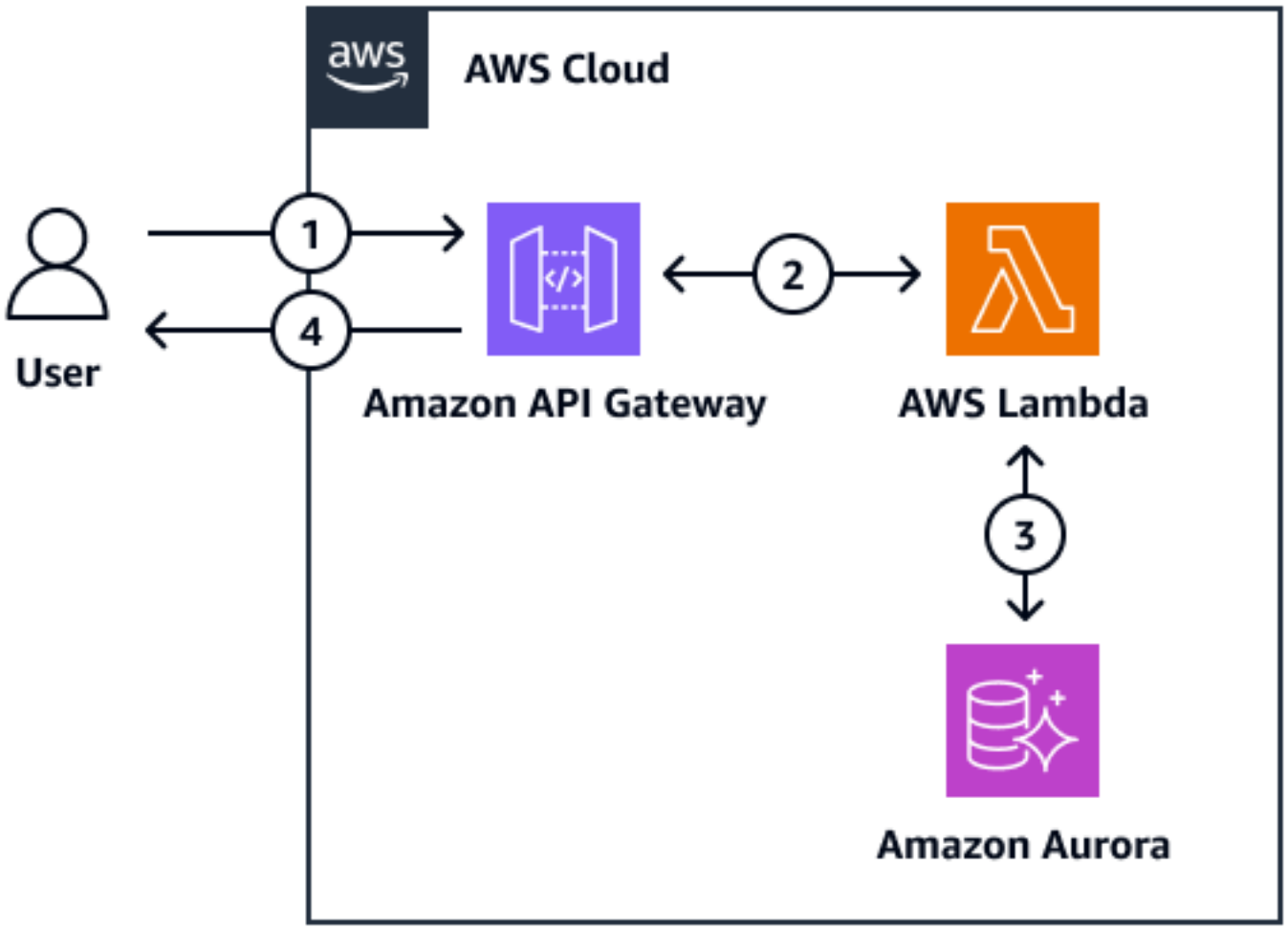
In this section

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:

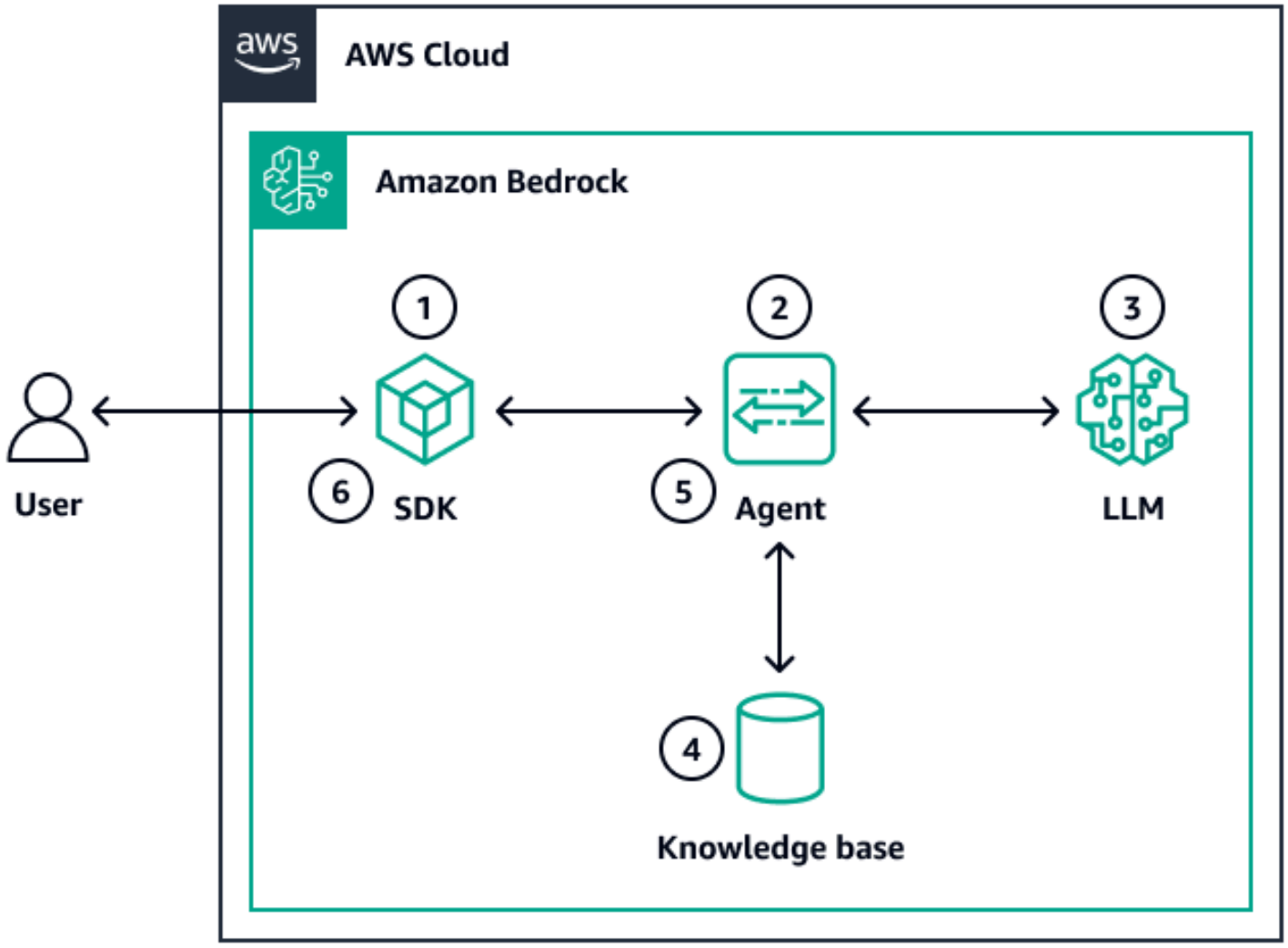


1. A user submits a request to Amazon API Gateway.
2. Amazon API Gateway routes the request to an AWS Lambda function.
3. AWS Lambda performs data enrichment by querying an Amazon Aurora database
4. Amazon API Gateway returns the enriched payload to the caller.

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.

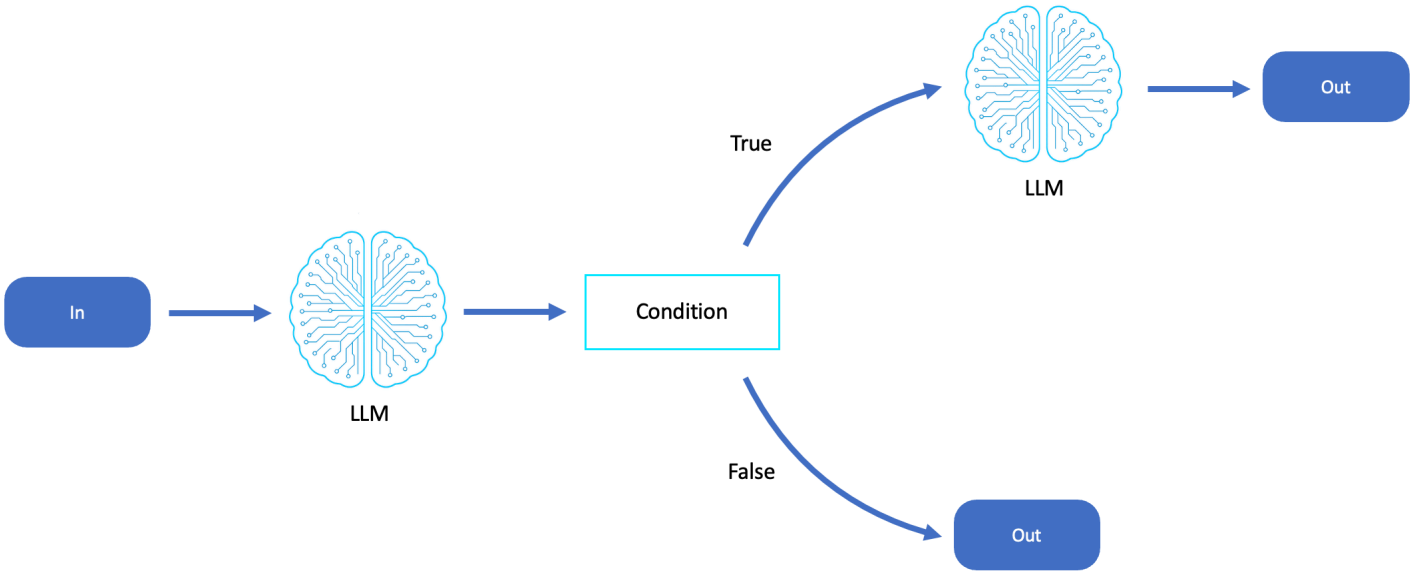
Core insights

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:

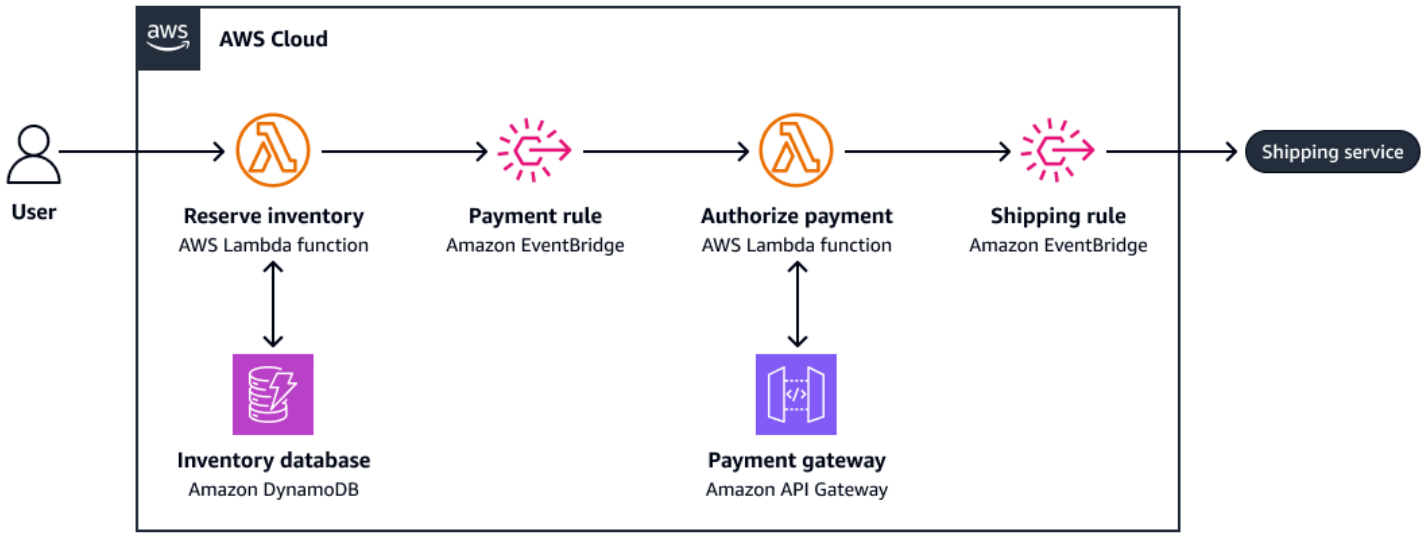


Saga choreography

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 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

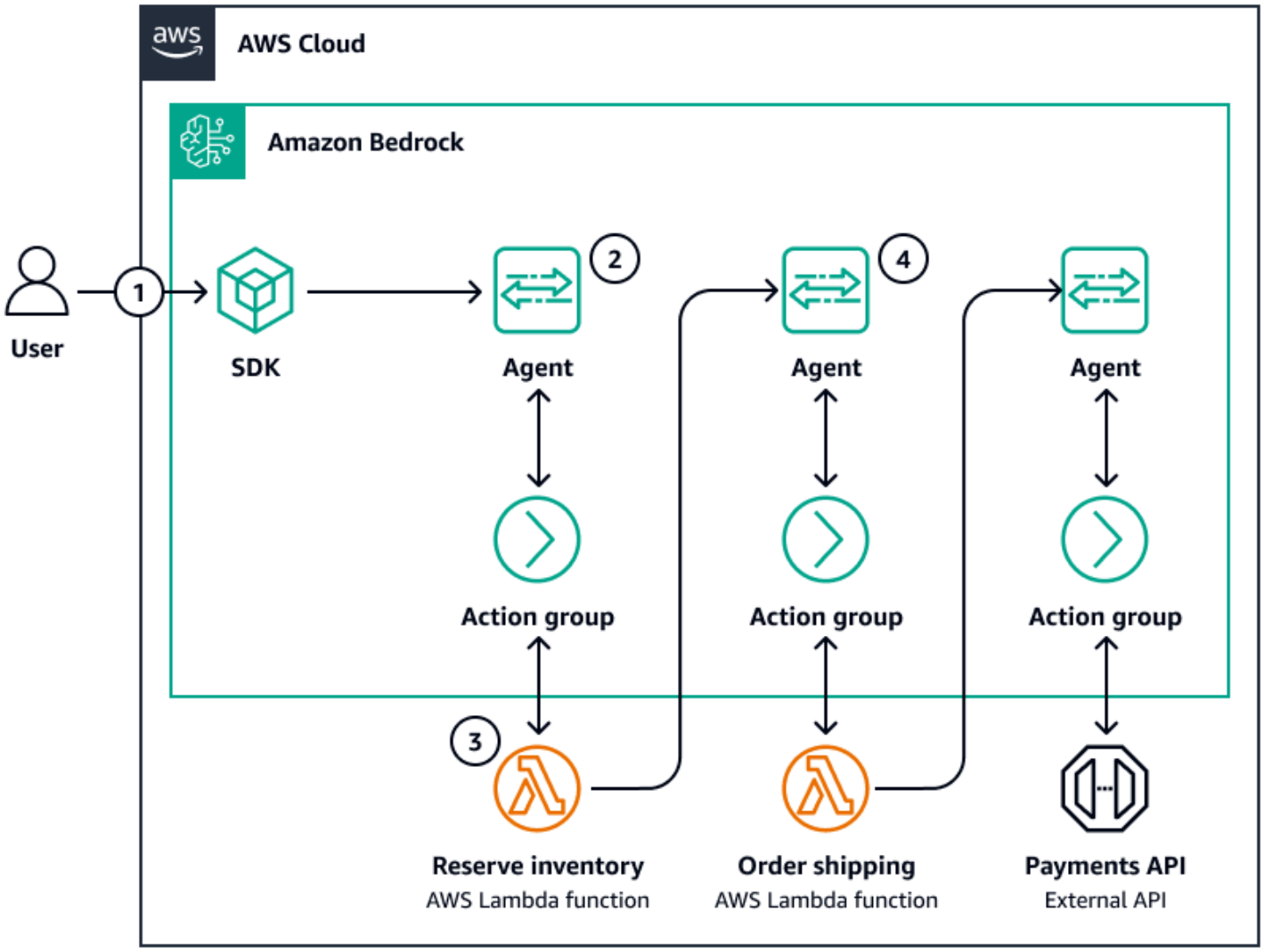
3. LLM executes a subtask (for example, by using a tool call or retrieving knowledge)
4. LLM refines the output or revisits an earlier step if it deems a result unsatisfactory

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 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

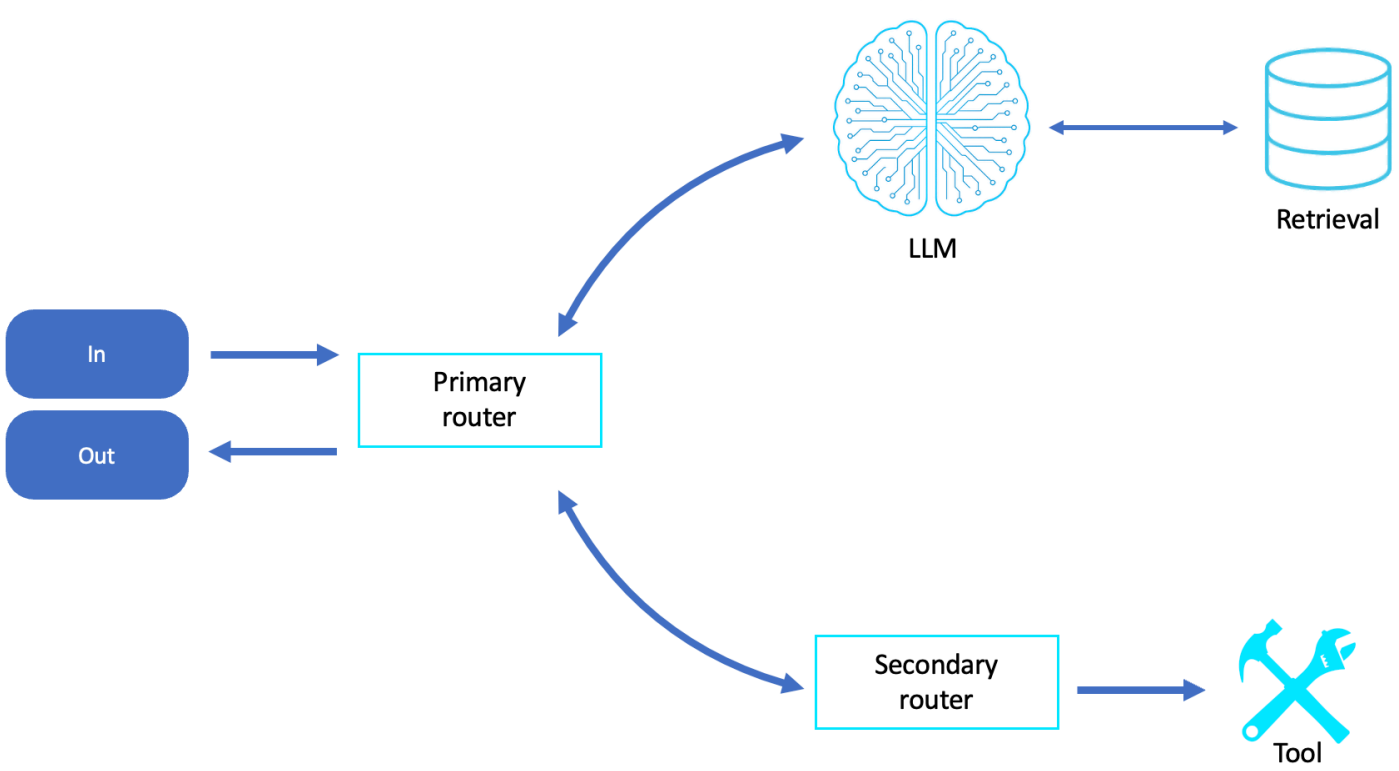
Where the saga pattern manages distributed service calls with compensating logic, prompt chaining manages reasoning tasks with reflective sequencing and adaptive replanning. Both systems allow for incremental progress, decentralized decision points, and failure recovery, and it does all of this through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

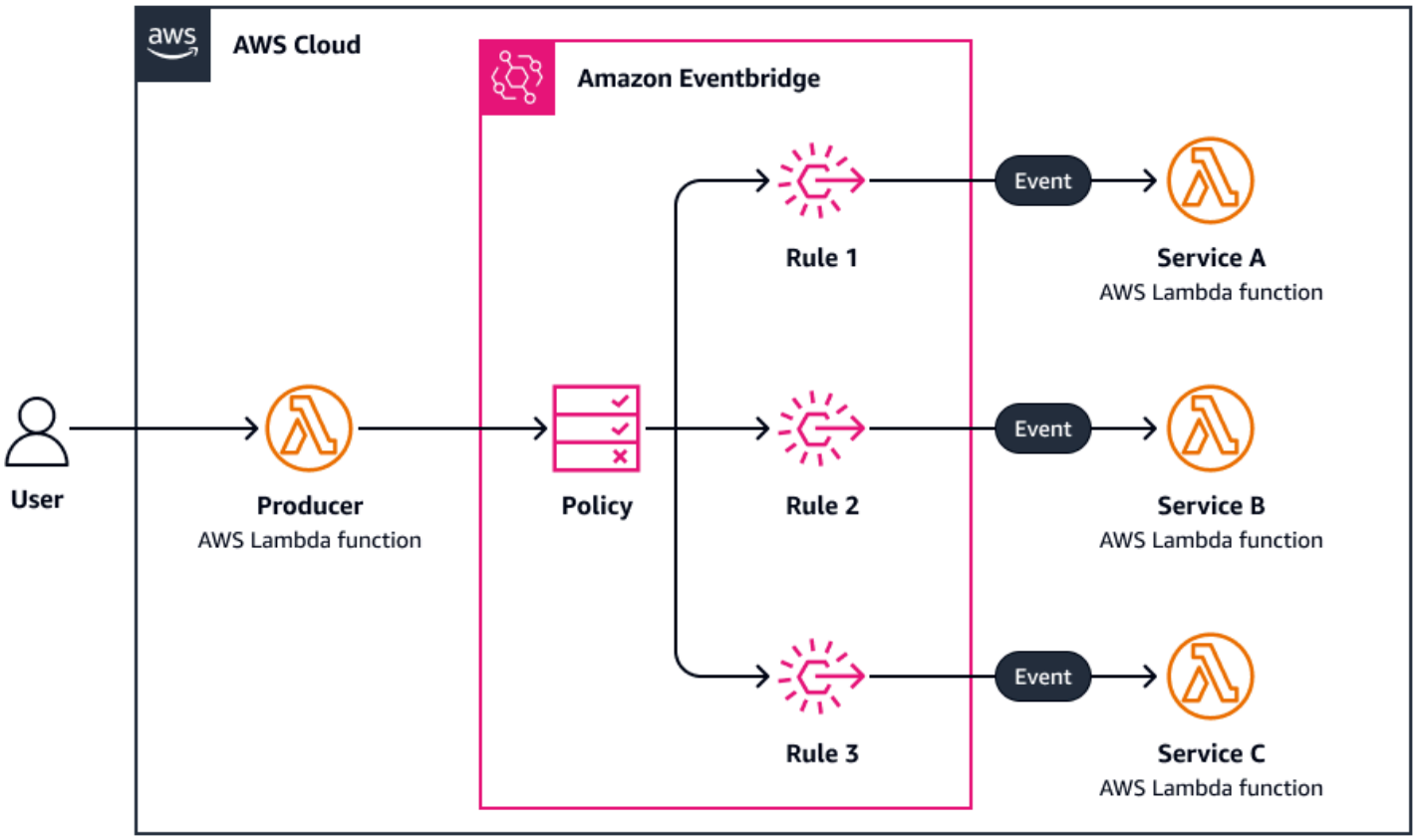
The following diagram is an example of LLM routing:



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 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

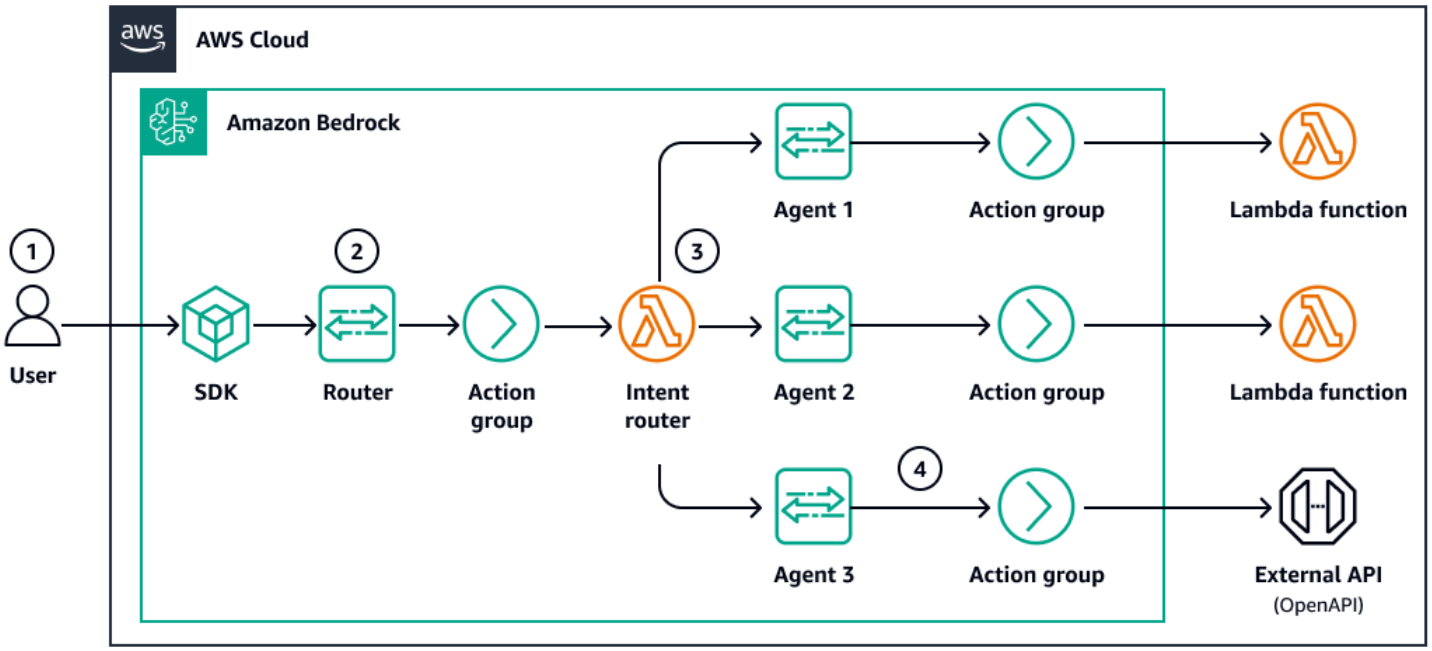
In agentic systems, routing also performs dynamic task delegation – but instead of Amazon EventBridge rules or metadata filters, the LLM classifies and interprets the user's intent through natural language. The result is a flexible, semantic, and adaptive form of dispatching.

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 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 EventBridge 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:

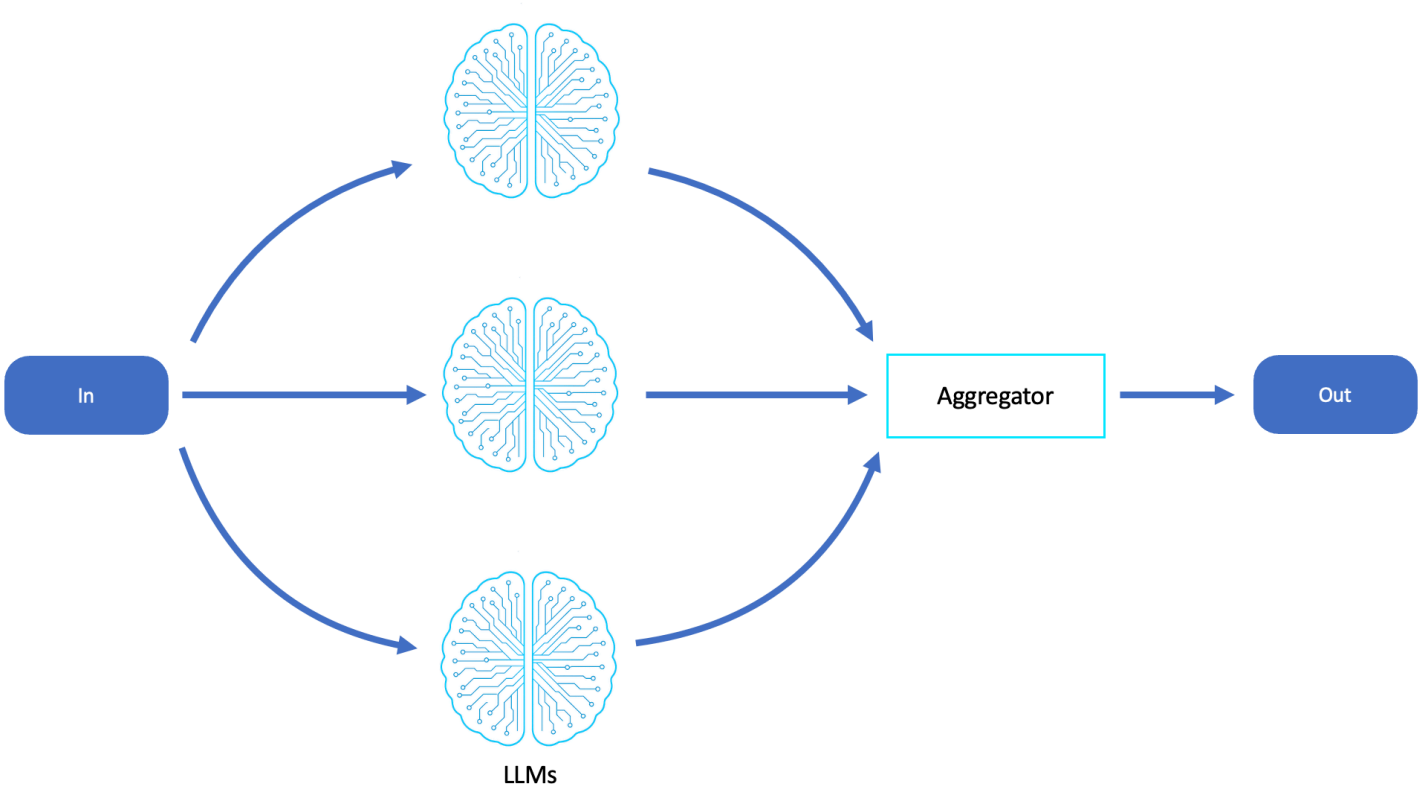
- Broader input understanding
- Intelligent fallback and tool selection
- Natural extensibility through new agent roles or prompt styles

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:



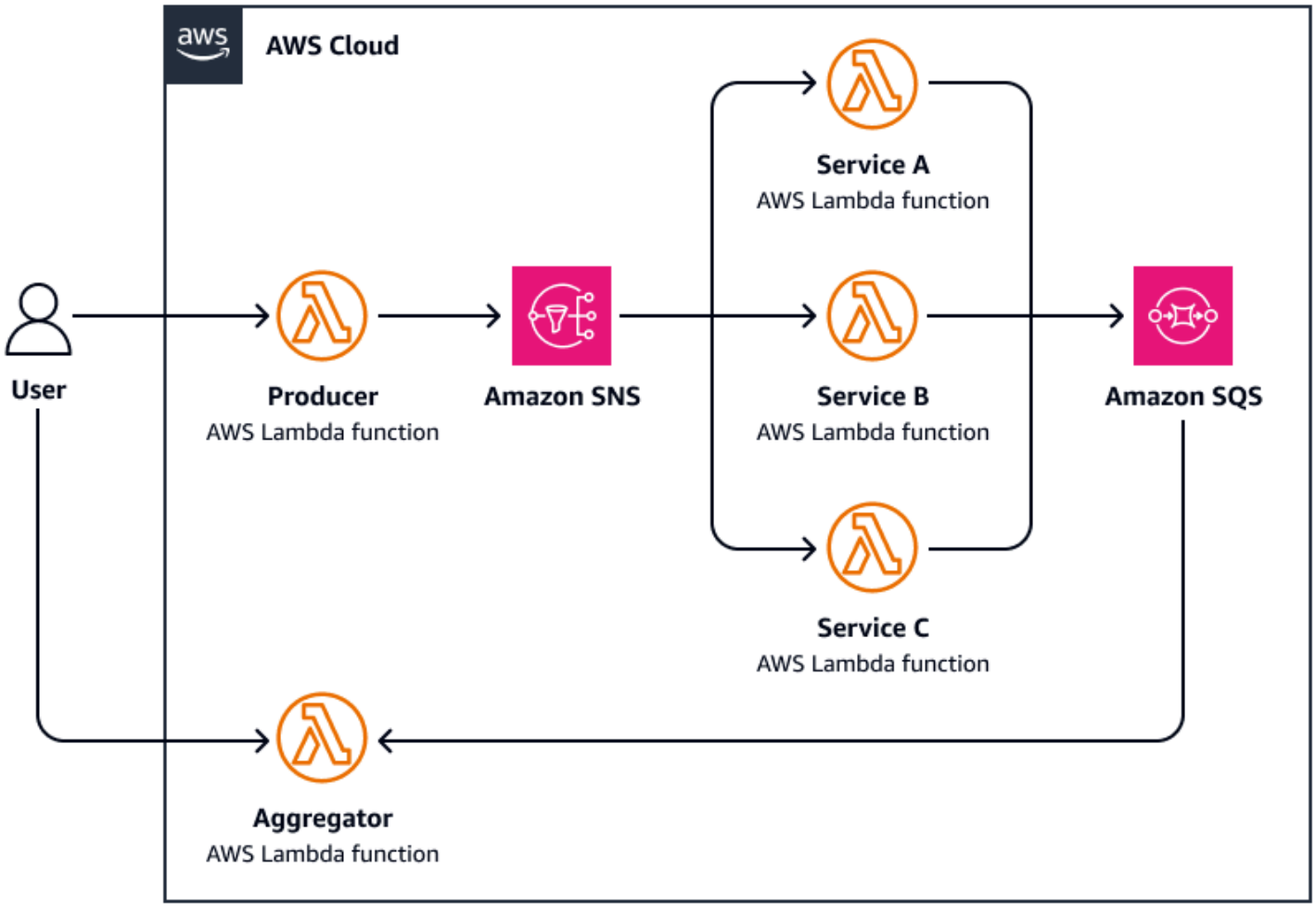
Scatter-gather

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 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)

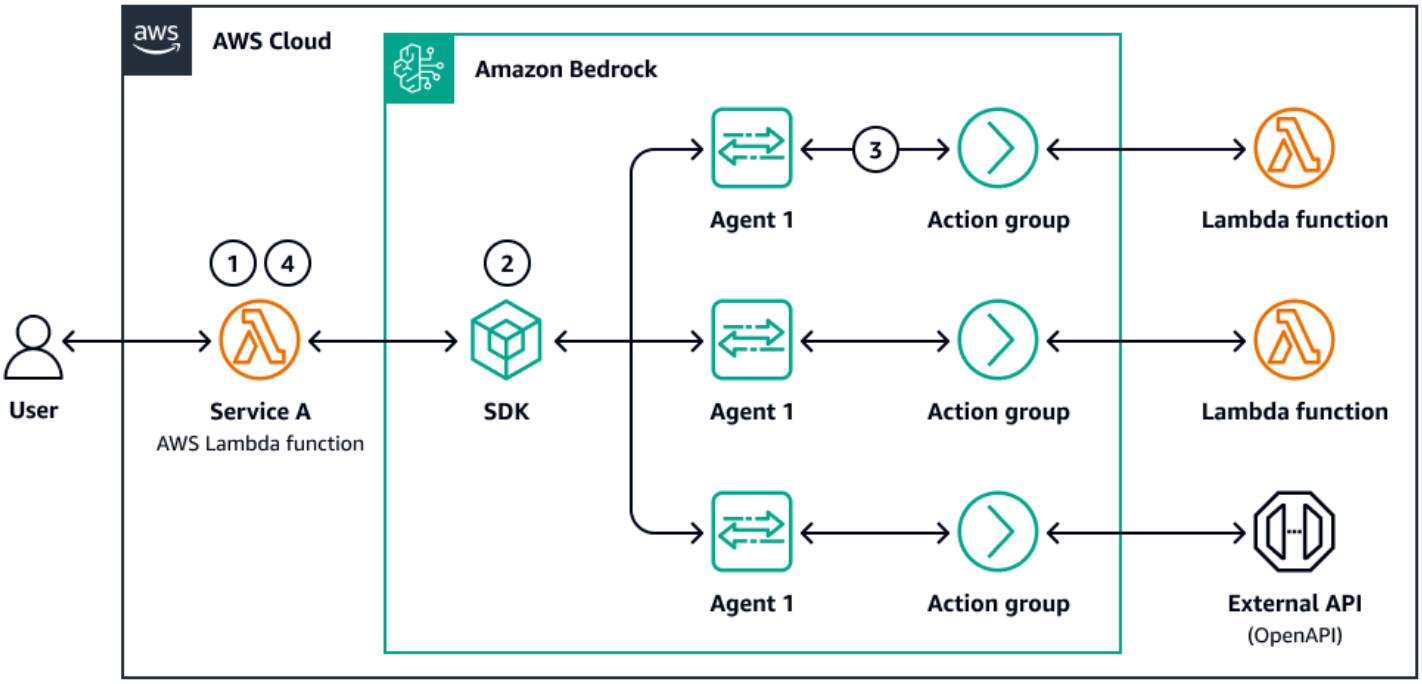
In agentic systems, parallelization closely mirrors scatter-gather by distributing subtasks across multiple LLM calls or agents, each independently reasoning through a portion of the problem. Returned results are gathered and synthesized by an aggregation process, which is often another LLM or controller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

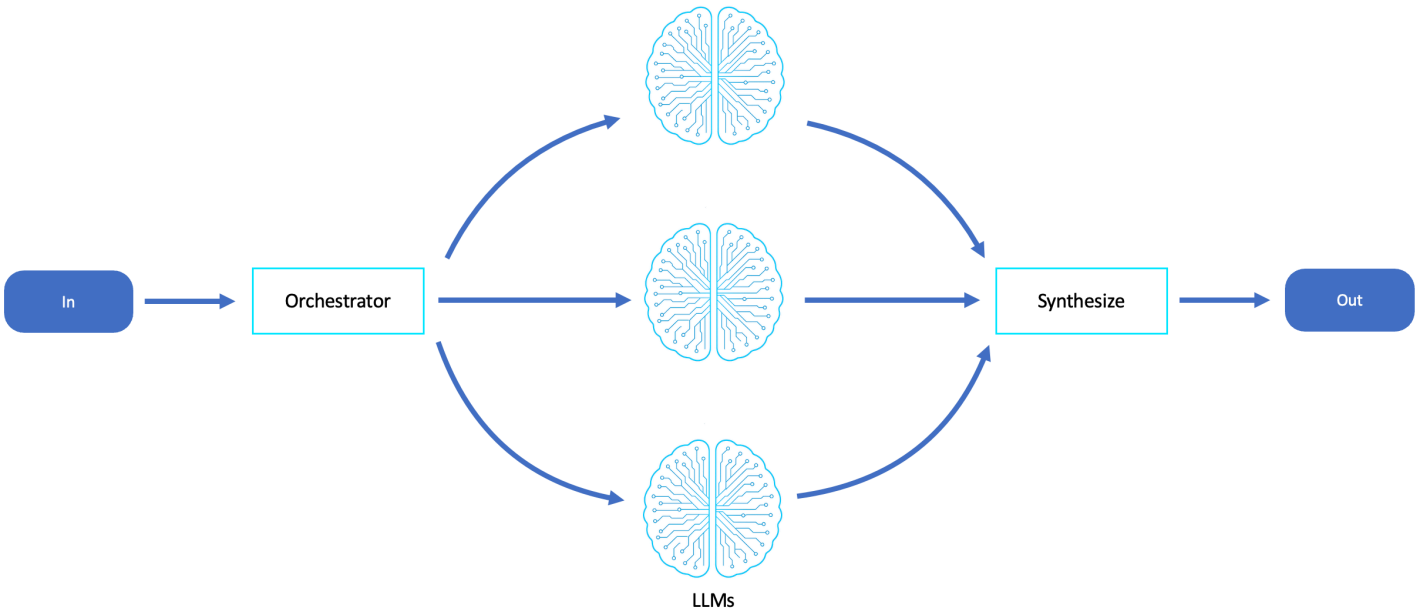
Agentic parallelization uses scatter-gather patterns to distribute LLM tasks, enabling parallel processing and intelligent result synthesis.

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:



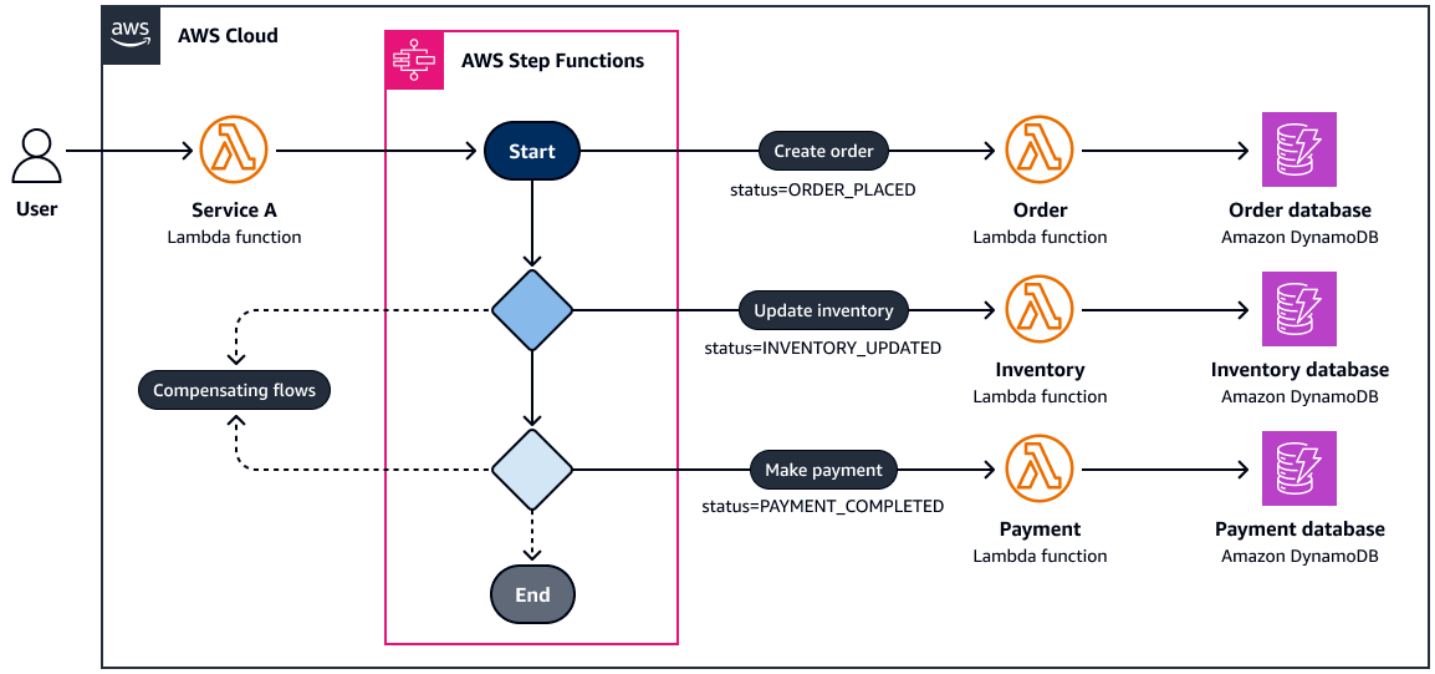
Event orchestration

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 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).

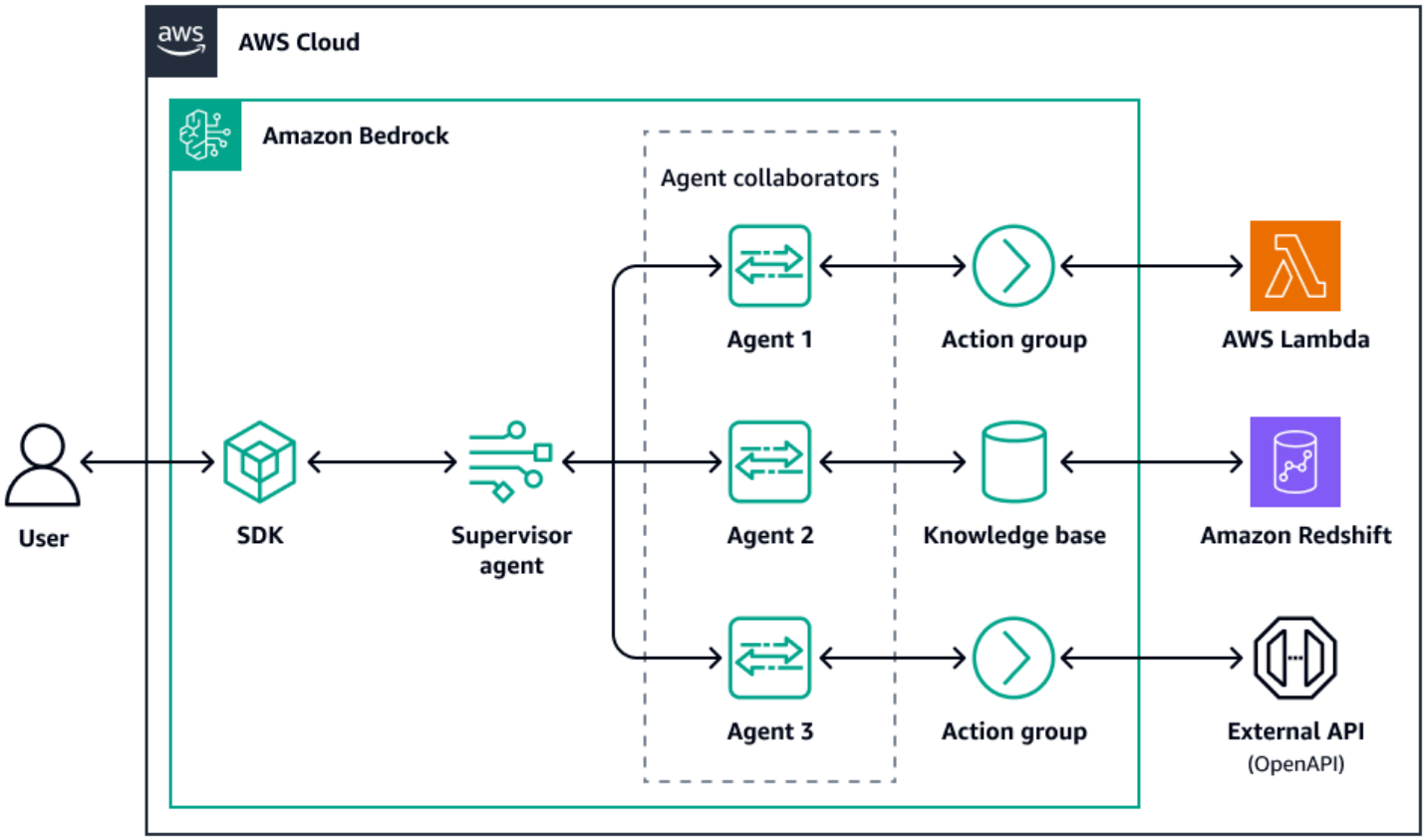
Supervisor

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 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:

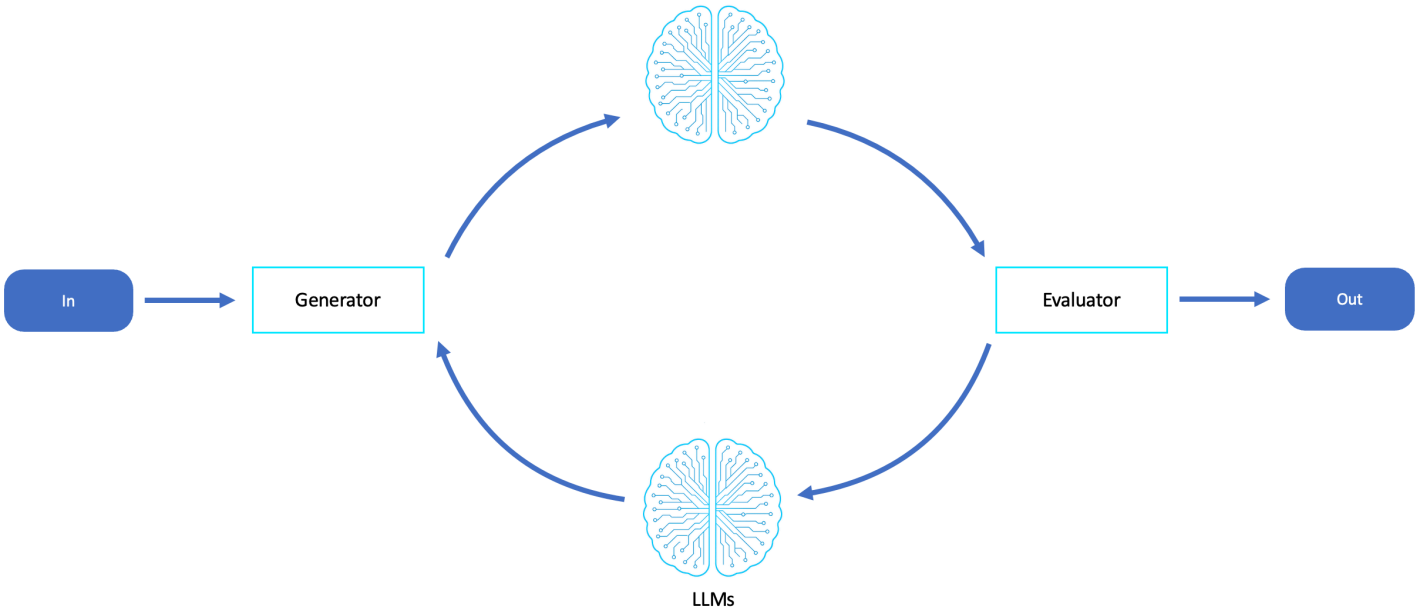
- Maintains context and execution flow
- Handles branching, sequencing, and error handling
- Produces a unified result from distributed components

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.

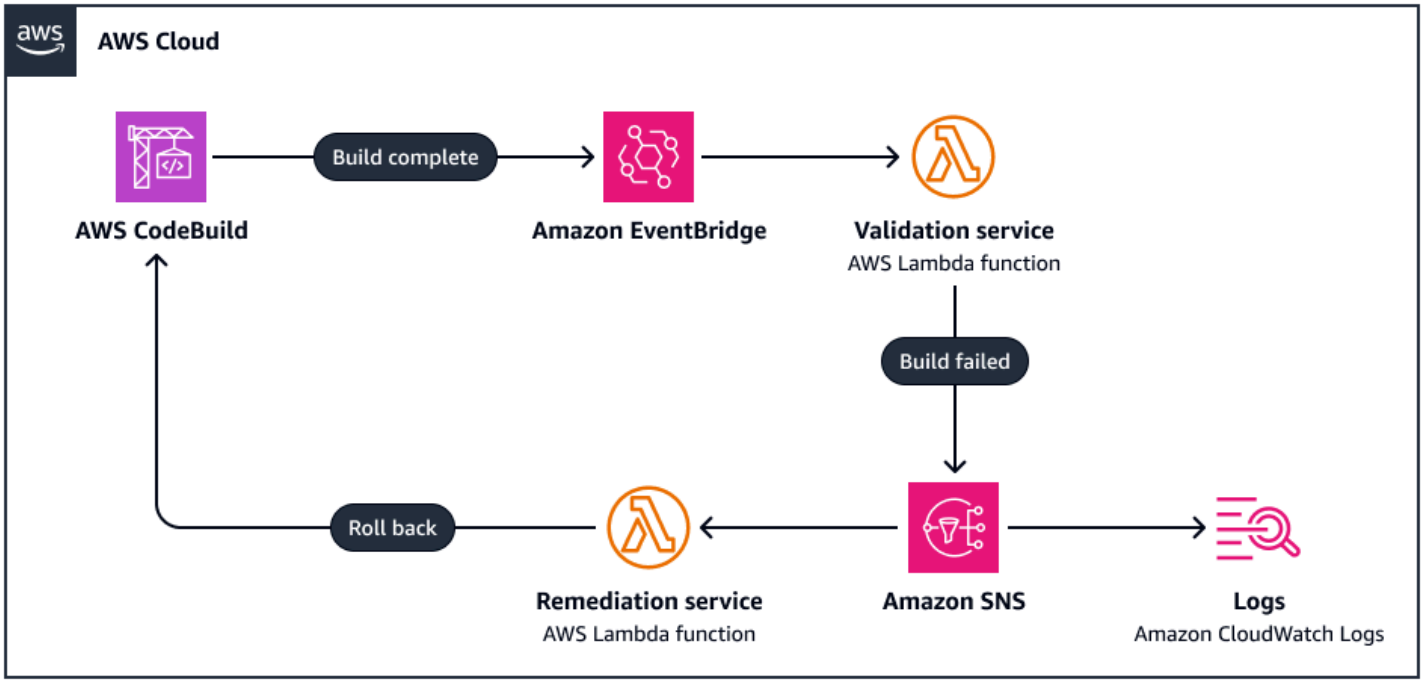
The following diagram is an example of an evaluator reflect-refine feedback loop:



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 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.

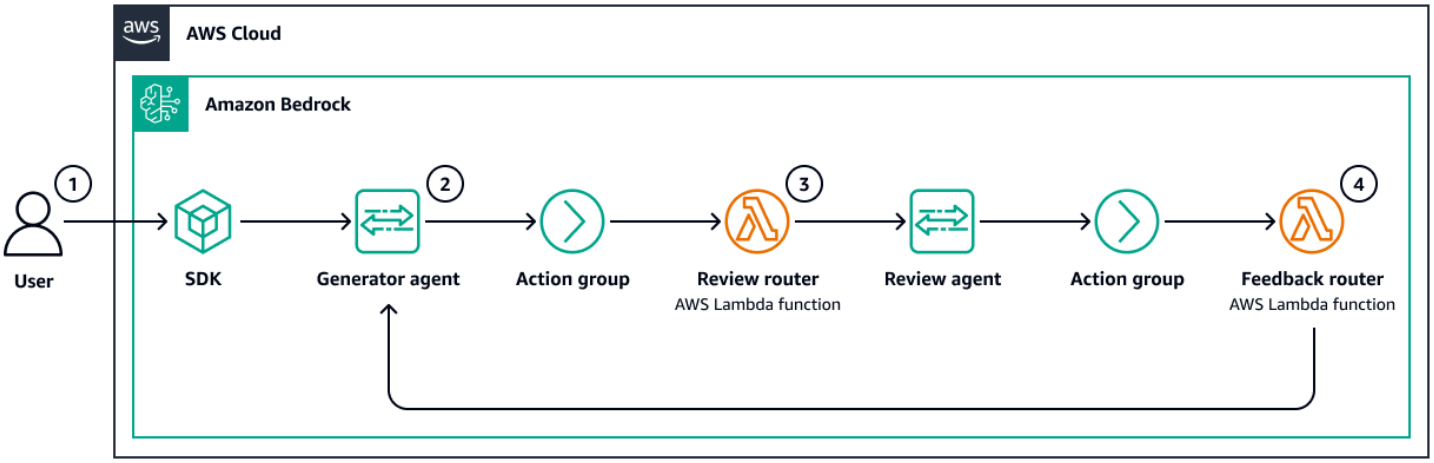
The loop repeats until the result meets a set of criteria, is approved, or reaches a retry limit.

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 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. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	July 14, 2025

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more

information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the “2021-05-27 00:15:37” date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries.

Detective guardrails detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub CSPM, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners,

migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata.

The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your

organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.