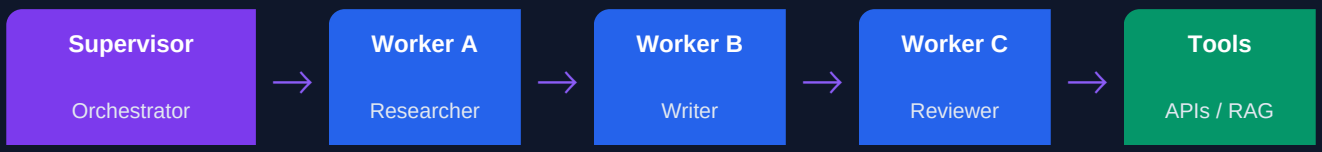
CLOUD AI DEVELOPER GUIDE SERIES

Guide #3

Multi-Agent Orchestration

Bedrock Agents + Step Functions · Azure AI Agent Service · Vertex AI Agent Builder



Senior / Architect ~35 min read Tier 3 of 3

cloudai.dev · 2025 Edition

Table of Contents

01 What Are AI Agents?

Agents vs. Chains vs. Single LLM Calls

As you move from a single LLM call to a full multi-agent system, you gain autonomy and capability at the cost of complexity and cost. Understanding when to use each tier is the first architectural decision.

Tier	Pattern	Control Flow	Use When
Single Call	One prompt, one response	You define everything	Classification, summarisation, extraction
Chain	Multiple LLM calls in fixed sequence	Hardcoded order	Multi-step transforms with known steps
Agent	LLM decides next action dynamically	Model-driven loop	Open-ended tasks, tool use, search
Multi-Agent	Multiple specialised agents collaborate	Supervisor or event-driven	Complex workflows, parallel work, scale

The Agent Loop: Observe, Think, Act

Every AI agent — regardless of framework — runs the same fundamental loop. Understanding it is essential before building multi-agent systems.

Observe	Receive input: user message, tool result, or message from another agent.
Think	Reason about the input using the LLM. Decide: answer directly, call a tool, or delegate to a sub-agent.
Act	Execute the chosen action: return text, invoke a tool (API, function, search, code), or send a message.
Loop	If the task is incomplete, feed the action result back into Observe and repeat until done or max steps reached.

When To Use Multi-Agent Systems

Parallel workstreams	Tasks that can be split and run concurrently — e.g. researching 5 topics simultaneously with 5 worker agents, then merging results.

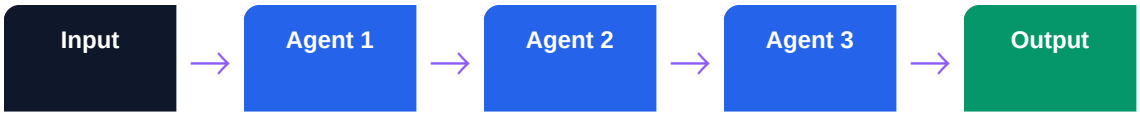
Specialisation	Different agents with different system prompts, models, and tools are better than one agent trying to do everything.
Long-horizon tasks	Tasks that exceed a single context window — a supervisor breaks the task into chunks and routes them across agents.
Quality gates	A reviewer/critic agent checks another agent's output before it reaches the user or the next step.
Cost optimisation	Route cheap subtasks (classification, formatting) to small models (Haiku, GPT-4o mini) and hard reasoning to large models.

02 Agent Patterns

These four patterns cover the vast majority of real-world multi-agent architectures. Most production systems are combinations of two or more.

Pattern 1 — Sequential Pipeline

Each agent receives the output of the previous agent as its input. Control flows in one direction: no loops, no branching. Predictable, debuggable, and easy to reason about.



Pros	Cons
+ Simple to implement	- No parallelism
+ Easy to debug & trace	- Single point of failure
+ Predictable latency	- Error propagates downstream

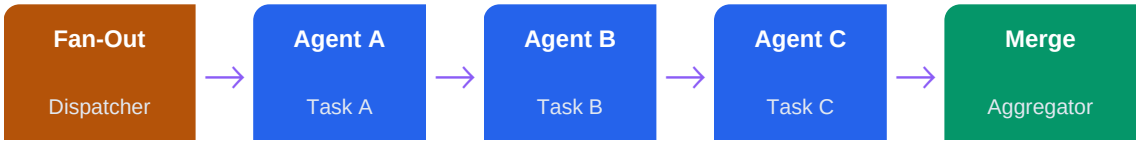
Pattern 2 — Supervisor / Worker

A supervisor (orchestrator) agent receives the task, breaks it down, and routes subtasks to specialised worker agents. Workers return results to the supervisor, which assembles the final output. This is the most common production pattern.

Pros	Cons
+ Specialised agents	- Supervisor is a bottleneck
+ Supervisor handles error recovery	- Harder to trace decisions
+ Easy to add new workers	- Token cost of routing prompts

Pattern 3 — Parallel Fan-Out / Fan-In

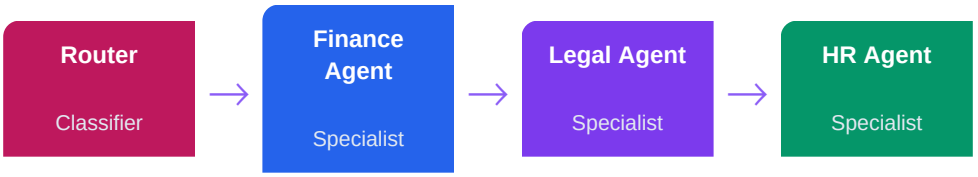
A dispatcher splits a task into N independent subtasks and fires them all simultaneously. Results are collected by an aggregator once all workers finish. Ideal for research, multi-document analysis, or batch processing.



Pros	Cons
+ Massive latency reduction	- Complex error handling
+ Scales to N subtasks	- All-or-nothing by default
+ Workers are independent	- Higher peak token spend

Pattern 4 — Handoff / Routing

A router agent classifies the incoming request and hands it off to the most appropriate specialist agent. The specialist handles the full request independently and returns the answer. Used in customer support bots, enterprise Q&A;, and domain-specific assistants.



Pros	Cons
+ Clean domain separation	- Router is a failure point
+ Cheap routing (small model)	- Ambiguous queries are hard to route
+ Easy to add new domains	- No cross-domain synthesis

03 Platform Agent Services

AWS Bedrock Agents + Step Functions

Bedrock Agents is AWS's native agentic framework — define an agent with a system prompt, attach action groups (Lambda functions or OpenAPI schemas), and optionally connect a Knowledge Base. For multi-agent pipelines, combine with Step Functions Express Workflows.

How it works: Create an agent in the Bedrock console or via SDK. Attach action groups that map to Lambda functions. The agent autonomously decides which actions to call, passes parameters, and iterates until the task is complete. For Supervisor/Worker patterns, invoke multiple Bedrock Agents from a Step Functions state machine or orchestrate via a Lambda supervisor.

- Action groups: Lambda functions or OpenAPI 3.0 schema definitions for external APIs
- Native Knowledge Base integration per agent for RAG-augmented responses
- Bedrock Guardrails applied independently per agent — different safety profiles per role
- Step Functions Express Workflows for parallel fan-out with 5-minute max execution
- Agent aliases + versioning for blue/green deployments and A/B testing
- Session memory: agents maintain short-term context across multi-turn conversations
- Trace feature: full chain-of-thought + action invocation log for debugging

Tool Support	Lambda (direct invocation), OpenAPI schema (REST endpoints), Knowledge Base retrieval, code interpreter
Orchestration	Step Functions Express Workflows (parallel/sequential), Lambda supervisor, EventBridge triggers

Azure AI Agent Service

Azure AI Agent Service (part of Azure AI Foundry) provides a managed runtime for building agents with OpenAI models. It handles the agent loop, tool execution, thread management, and file handling — letting you focus on tools and prompts.

How it works: Create an Assistant with instructions and tools attached. Use Threads to manage conversation state. Start a Run to trigger the agent loop. For multi-agent systems, use AutoGen (Microsoft's open-source framework) or the Semantic Kernel Agent Framework, both deeply integrated with Azure OpenAI.

- Built-in tools: Code Interpreter, File Search (vector store), Bing Search grounding
- Function calling: define custom Python or REST functions as tools
- AutoGen integration: multi-agent conversation patterns with GroupChat and round-robin
- Semantic Kernel: enterprise-grade agent orchestration with C# and Python SDKs
- Thread persistence: conversations stored server-side, resumable across sessions
- Azure AI Foundry: visual agent playground, evals, and prompt flow pipelines
- Managed identity and private networking for enterprise deployments

Tool Support	Code Interpreter, File Search, Bing Search, custom function calling, Azure AI Search
Orchestration	AutoGen GroupChat, Semantic Kernel Agent Framework, Prompt Flow (DAG-based pipelines)

GCP Vertex AI Agent Builder

Vertex AI Agent Builder lets you create agents backed by Gemini models with tools, data stores, and extensions. For multi-agent systems, use the Agent Engine (managed runtime) or Reasoning Engine for complex orchestration logic.

How it works: Define an agent in Agent Builder with instructions, tools (Extensions), and data store connections. Deploy to Agent Engine for a managed, scalable API endpoint. For custom multi-agent logic, use Reasoning Engine with LangChain, LangGraph, or custom Python — all running on Vertex infrastructure.

- Extensions: pre-built integrations (Google Search, Calendar, Gmail, code execution)
- Data store tools: connect Vertex AI Search data stores directly to agents
- Grounding: agents can cite Google Search results in real-time
- Reasoning Engine: deploy LangGraph, LangChain, or custom agent logic as managed APIs
- Agent Engine: fully managed hosting — no infrastructure, auto-scaling, IAM-secured
- Evaluation: built-in agent evaluation with Vertex AI Experiments
- BigQuery tool: agents can query BigQuery tables directly via natural language

Tool Support	Google Search, Code Execution, Calendar, Gmail, BigQuery, Vertex AI Search, custom OpenAPI
Orchestration	Reasoning Engine (LangGraph/LangChain), Agent Engine, Workflows (Cloud Workflows integration)

04 Side-by-Side Comparison

Agent Feature Matrix

Feature	AWS Bedrock Agents	Azure AI Agent Service	GCP Vertex AI Agents
Supported Models	Claude, Nova, Llama, Mistral	GPT-4o, o1, GPT-4 Turbo	Gemini 1.5 Pro/Flash, Llama
Tool Types	Lambda, OpenAPI, KB	Functions, Code, File Search, Bing	Extensions, OpenAPI, BQ, Search
Multi-agent pattern	Step Functions + Bedrock	AutoGen / Semantic Kernel	Reasoning Engine / LangGraph
Built-in memory	Session (short-term)	Thread persistence (long-term)	Session + Reasoning Engine state
RAG integration	Native KB attachment	File Search / AI Search	Data Store tool attachment
Guardrails	Bedrock Guardrails per agent	Azure Content Safety	Vertex safety filters
Agent versioning	Aliases + versions	Assistant versioning	Agent versions in Agent Engine
Visual builder	Bedrock console	Azure AI Foundry	Agent Builder console
Tracing / debug	Trace API (step-by-step)	Run Steps API + Azure Monitor	Cloud Trace + Reasoning logs
Self-hosted option	No — fully managed	Yes — AutoGen open-source	Yes — LangGraph on GKE
Async execution	Step Functions async	Async Run polling	Cloud Tasks + Workflows

Decision Guide

Choose...	When...
AWS Bedrock Agents	You are already on AWS, need native Claude/Nova models, want Guardrails + Knowledge Bases tightly integrated, and prefer Lambda as your tool runtime.
Azure AI Agent Service	Your org uses Microsoft stack, you need thread-persistent multi-turn agents, want AutoGen for complex multi-agent conversations, or need Code Interpreter out of the box.
GCP Vertex AI Agents	You want Gemini models, Google Search grounding, BigQuery as a native tool, or prefer LangGraph/LangChain on a fully managed Vertex runtime.

05 Build a Multi-Agent Pipeline

Config-Driven team.json Pattern

Rather than hardcoding agent roles and models in code, a **config-driven approach** defines your agent team in a JSON file. This makes it trivial to swap models, add workers, or change routing logic without touching application code — critical for production pipelines.

```
{
"team": "content-pipeline",
"agents": [
{
"id": "supervisor",
"agentRole": "supervisor",
"model": "amazon.nova-pro-v1:0",
"systemPrompt": "You are an orchestrator. Break tasks into subtasks and delegate.",
"maxTokens": 2048
},
{
"id": "researcher",
"agentRole": "worker",
"model": "amazon.nova-lite-v1:0",
"systemPrompt": "You are a researcher. Return structured findings as JSON.",
"tools": ["web_search", "knowledge_base"],
"maxTokens": 1024
},
{
"id": "writer",
"agentRole": "worker",
"model": "anthropic.claude-3-haiku-20240307-v1:0",
"systemPrompt": "You are a writer. Produce polished content from research briefs.",
"maxTokens": 4096
},
{
"id": "reviewer",
"agentRole": "worker",
"model": "anthropic.claude-3-haiku-20240307-v1:0",
"systemPrompt": "You are a quality reviewer. Score and suggest improvements.",
"maxTokens": 512
}
]
```

```
}
```

AWS: Bedrock Agents + Step Functions Supervisor (Python)

Prerequisites: Bedrock Agents created for each worker · Step Functions state machine defined · IAM roles with bedrock:InvokeAgent

Install:

```
pip install boto3
```

```
import boto3, json

bedrock_agent = boto3.client("bedrock-agent-runtime", region_name="us-east-1")
sfn_client = boto3.client("stepfunctions")

# Agent IDs from team.json (mapped to Bedrock Agent IDs)
AGENTS = {
"researcher": {"agentId": "AGENT_ID_A", "agentAliasId": "ALIAS_A"},
"writer": {"agentId": "AGENT_ID_B", "agentAliasId": "ALIAS_B"},
"reviewer": {"agentId": "AGENT_ID_C", "agentAliasId": "ALIAS_C"},
}

def invoke_agent(role: str, input_text: str, session_id: str) -> str:
cfg = AGENTS[role]
response = bedrock_agent.invoke_agent(
agentId=cfg["agentId"],
agentAliasId=cfg["agentAliasId"],
sessionId=session_id,
inputText=input_text,
)
result = ""
for event in response["completion"]:
if "chunk" in event:
result += event["chunk"]["bytes"].decode("utf-8")
return result

def run_pipeline(topic: str) -> dict:
session_id = f"session-{topic[:20].replace(' ', '-')}"

# Step 1: Research
research = invoke_agent(
"researcher",
f"Research this topic thoroughly and return a JSON brief: {topic}",
session_id
)

# Step 2: Write
draft = invoke_agent(
```

```
    "writer",
    f"Write a LinkedIn article based on this research brief:\n{research}",
    session_id
    )

    # Step 3: Review
    review = invoke_agent(
    "reviewer",
    f"Review this draft and return a score (1-10) and 3 improvements:\n{draft}",
    session_id
    )

    return {"research": research, "draft": draft, "review": review}

    # Run the pipeline
    result = run_pipeline("The impact of multi-agent AI on software engineering teams")
    print(result["draft"])
    print("Review:", result["review"])
```

Azure: AI Agent Service — Multi-Agent with AutoGen (Python)

Prerequisites: Azure OpenAI resource · azure-ai-projects SDK installed · Project connection string

Install:

```
pip install azure-ai-projects azure-identity
```

```
import os
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential

# Connect to Azure AI Foundry project
client = AIProjectClient.from_connection_string(
conn_str=os.environ["AZURE_AI_PROJECT_CONN_STR"],
credential=DefaultAzureCredential(),
)
agents_client = client.agents

# Create researcher agent
researcher = agents_client.create_agent(
model="gpt-4o",
name="researcher",
instructions="You are a researcher. Return structured findings as JSON.",
tools=[{"type": "bing_grounding"}],
)

# Create writer agent
writer = agents_client.create_agent(
model="gpt-4o",
```

```
    name="writer",
    instructions="You are a writer. Produce polished LinkedIn articles from research
    briefs.",
    )

    # Create reviewer agent
    reviewer = agents_client.create_agent(
    model="gpt-4o-mini",
    name="reviewer",
    instructions="Review content and return a score (1-10) with 3 improvements.",
    )

    # Run pipeline on a shared thread
    thread = agents_client.create_thread()
    topic = "The impact of multi-agent AI on software engineering teams"

    def run_agent(agent, prompt: str) -> str:
    agents_client.create_message(thread_id=thread.id, role="user", content=prompt)
    run = agents_client.create_and_process_run(thread_id=thread.id, agent_id=agent.id)
    messages = agents_client.list_messages(thread_id=thread.id)
    return messages.data[0].content[0].text.value

    research = run_agent(researcher, f"Research: {topic}")
    draft = run_agent(writer, f"Write an article based on:\n{research}")
    review = run_agent(reviewer, f"Review this draft:\n{draft}")

    print(draft)
    print("Review:", review)

    # Cleanup
    for a in [researcher, writer, reviewer]:
    agents_client.delete_agent(a.id)
```

GCP: Vertex AI Agent Engine + LangGraph (Python)

Prerequisites: GCP project · Vertex AI API enabled · Reasoning Engine API enabled · ADC configured

Install:

```
    pip install google-cloud-aiplatform langgraph langchain-google-vertexai
```

```
    import vertexai
    from vertexai.preview import reasoning_engines
    from langchain_google_vertexai import ChatVertexAI
    from langgraph.graph import StateGraph, END
    from typing import TypedDict, Annotated
    import operator

    vertexai.init(project="your-gcp-project-id", location="us-central1")

    # Shared state definition
```

```
class PipelineState(TypedDict):
topic: str
research: str
draft: str
review: str

# Initialise models (different sizes per role)
supervisor_llm = ChatVertexAI(model_name="gemini-1.5-pro")
worker_llm = ChatVertexAI(model_name="gemini-1.5-flash")

# Node functions
def researcher_node(state: PipelineState) -> PipelineState:
result = worker_llm.invoke(
f"Research this topic and return a structured JSON brief: {state['topic']}"
)
return {**state, "research": result.content}

def writer_node(state: PipelineState) -> PipelineState:
result = worker_llm.invoke(
f"Write a LinkedIn article based on this research:\n{state['research']}"
)
return {**state, "draft": result.content}

def reviewer_node(state: PipelineState) -> PipelineState:
result = worker_llm.invoke(
f"Review this draft, score 1-10, give 3 improvements:\n{state['draft']}"
)
return {**state, "review": result.content}

# Build the LangGraph pipeline
graph = StateGraph(PipelineState)
graph.add_node("researcher", researcher_node)
graph.add_node("writer", writer_node)
graph.add_node("reviewer", reviewer_node)
graph.set_entry_point("researcher")
graph.add_edge("researcher", "writer")
graph.add_edge("writer", "reviewer")
graph.add_edge("reviewer", END)
pipeline = graph.compile()

# Run locally (or deploy to Reasoning Engine for managed API)
result = pipeline.invoke({
"topic": "The impact of multi-agent AI on software engineering teams",
"research": "", "draft": "", "review": ""
})
print(result["draft"])
```

```
print("Review:", result["review"])
```

■■ Max iteration limits are critical

Always set a maximum iteration/step limit on every agent loop. An unguarded agent can enter an infinite tool-call loop, consuming thousands of tokens and dollars. For Bedrock: set maxLength on the action group. For Azure: set max_completion_tokens. For Vertex/LangGraph: use recursion_limit in the graph config.

06 Production Patterns

Error Handling & Retries

Classify errors	Distinguish transient errors (throttling, timeout) from permanent errors (invalid input, model refusal). Only retry transient errors.
Exponential backoff	For throttling errors, retry with exponential backoff + jitter. Start at 1s, double each attempt, cap at 60s, max 5 retries.
Dead-letter queues	For async pipelines (Step Functions, SQS), route failed tasks to a DLQ for inspection and manual replay instead of losing them silently.
Partial failure	In fan-out patterns, if one worker fails, decide: fail the whole pipeline, continue with partial results, or substitute a fallback response.
Circuit breakers	If a downstream tool (API, database) is consistently failing, fail fast with a fallback rather than retrying and blocking the agent loop.

Observability & Tracing

What to Log	Why	Tool
Agent input + output per step	Full trace of decisions	CloudWatch / Azure Monitor / Cloud Logging
Tool calls + parameters	Understand what the agent did	Bedrock Trace API / Run Steps / Cloud Trace
Token counts per step	Cost attribution and budgeting	CloudWatch Metrics / Azure Metrics
Latency per agent	Find the bottleneck in the pipeline	X-Ray / App Insights / Cloud Profiler
Guardrail interventions	Detect prompt injection, policy violations	Bedrock Guardrails logs / Content Safety logs
Model refusals	Identify misconfigured prompts	Custom structured logging

Guardrails Per Agent

In a multi-agent system, different agents have different risk profiles. A researcher agent may need broad web access; a customer-facing writer agent needs strict content filtering. Apply guardrails at the agent level, not just the pipeline level.

Agent Role	Recommended Guardrails
Supervisor / Orchestrator	Prompt attack detection, deny list for disallowed topics, PII redaction on inputs
Researcher / RAG worker	URL/domain allowlist for tools, output length limits, source citation enforcement
Writer / Content agent	Toxicity filter, brand tone enforcement, PII redaction on outputs
Customer-facing agent	Strict topic deny list, profanity filter, grounding check (no hallucination)
Code execution agent	Code sandbox isolation, blocked OS calls, output size limits

Cost & Latency Controls

- Use the **smallest model that works** for each agent role. Routing, classification, and formatting tasks rarely need GPT-4o or Claude Sonnet — Haiku or GPT-4o mini are 10-20x cheaper.
- Set **max_tokens per agent**. A reviewer agent returning a 5-point JSON score does not need 4096 tokens. Capping tokens is the fastest way to cut cost and reduce latency.
- Cache repeated tool call results within a session. If two agents call the same API with the same parameters in one pipeline run, serve the cached response.
- Use **prompt caching** (available on Claude via Bedrock) to cache long system prompts. A 2000-token system prompt cached across 100 calls saves 200,000 input tokens.
- For non-latency-sensitive pipelines (batch content generation, nightly reports), use **Bedrock Batch Inference** or Azure batch endpoints — up to 50% cost reduction.
- Monitor P95 latency per agent. If one agent is consistently the bottleneck, run it in parallel with the preceding agent or pre-warm it with a keep-alive Lambda.

■ TeamWeave Pattern: Config-Driven Agent Roles

The config-driven team.json pattern shown in Section 05 maps directly to production supervisor/worker orchestration. Define agentRole: supervisor to route all traffic through a single orchestrator, and agentRole: worker for specialised models. Assign models per role (Nova Pro for reasoning, Haiku for writing tasks) to balance cost and quality at scale.

■ Up Next: Guide #4 — Guardrails & Security

Learn how to implement content filtering, PII protection, prompt injection defence, and compliance controls across AWS Bedrock Guardrails, Azure Content Safety, and GCP Vertex AI safety filters. Built for regulated industries.