AWS re:Invent

DECEMBER 1 – 5, 2025 | LAS VEGAS, NV

AIM3300-R

Architecting multi-agent systems with Amazon Bedrock AgentCore

Laith Al-Saadoon

(he/him)

Principal AI Engineer

AWS

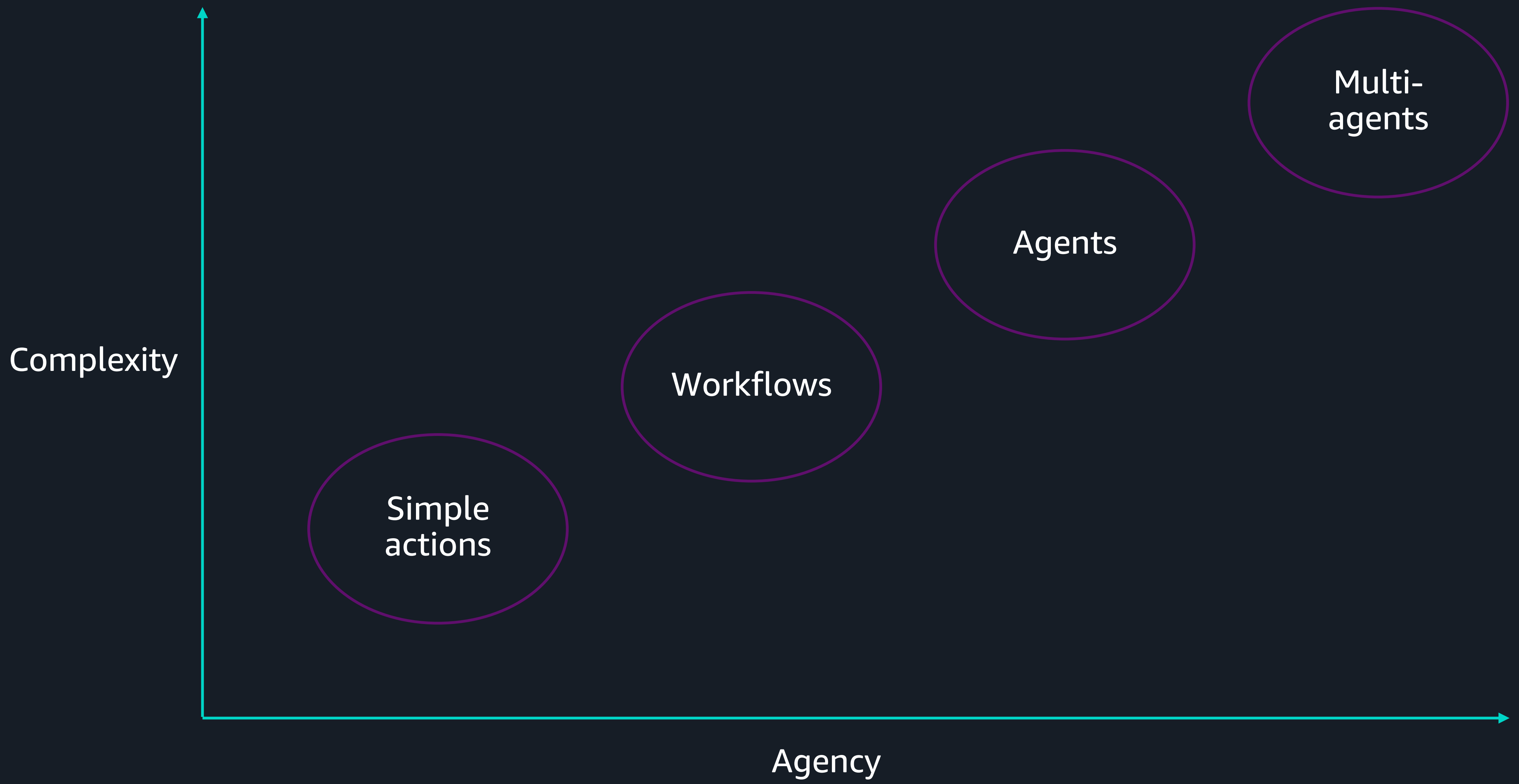
Alain Krok

(he/him)

Senior AI Engineer

AWS

What are multi-agent systems?


Multi-
agents
Agents
Complexity
Workflows
Simple
actions
Agency

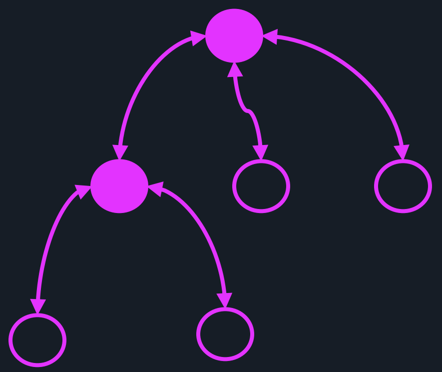
KEY BENEFITS

Why choose multi-agent architecture?

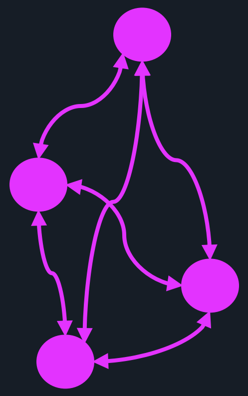
- Specialized agents outperform generalists
- Parallel processing scales performance
- Separation of concerns improves maintainability
- Independent prompts, context, and tools per agent
- Better conceptual model for complex tasks
- Can scale beyond single context window limits

Four core patterns

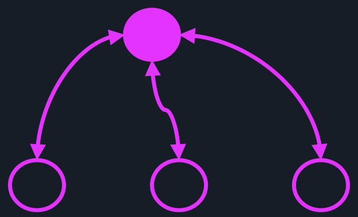
Hierarchical



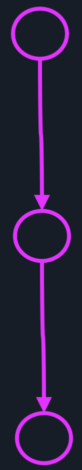
SWARM
(multi-agent collaboration)



Competitive



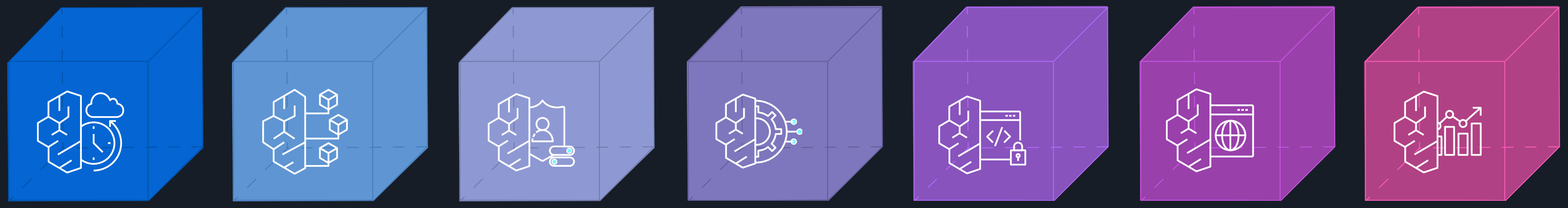
Multi-agent DAG



GENERALLY AVAILABLE

Amazon Bedrock AgentCore

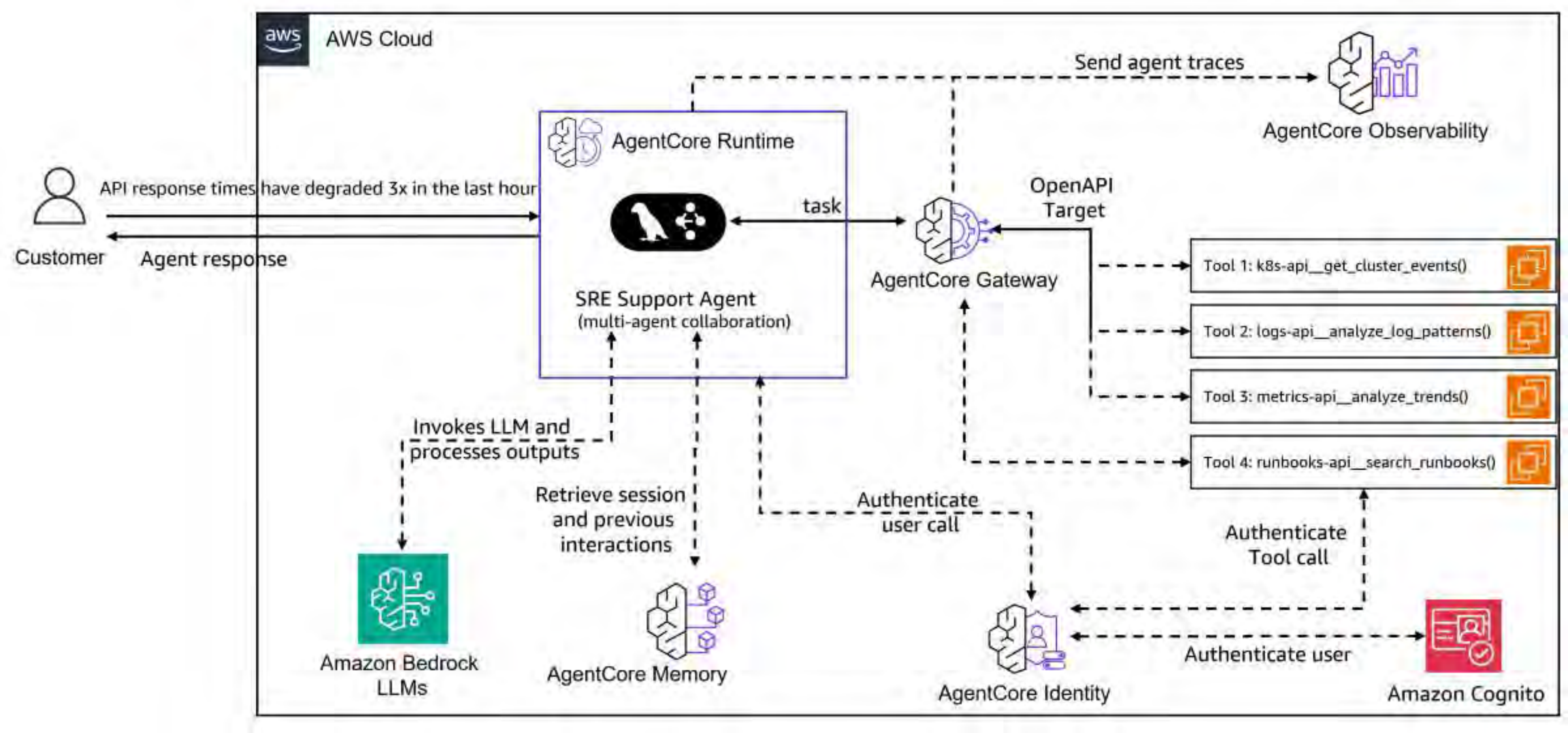
Comprehensive agentic platform: Everything you need for getting agents into production



Amazon Bedrock AgentCore

- AgentCore Runtime
  - Isolated sessions up to 8 hrs
  - Any framework, any LLM. Simple entrypoint contract
- AgentCore Memory
  - Automatic memory extraction (short-term + long-term)
  - Vector storage for semantic retrieval
- AgentCore Gateway
  - MCP client for tool connections
  - Semantic search over tools: 300 tools → 4 relevant ones

Whiteboarding


AWS Cloud
Customer
API response times have degraded 3x in the last hour
Agent response
AgentCore Runtime
task
SRE Support Agent
(multi-agent collaboration)
Send agent traces
AgentCore Observability
OpenAPI
Target
AgentCore Gateway
Tool 1: k8s-api__get_cluster_events()
Tool 2: logs-api__analyze_log_patterns()
Tool 3: metrics-api__analyze_trends()
Tool 4: runbooks-api__search_runbooks()
Invokes LLM and
processes outputs
Retrieve session
and previous
interactions
Authenticate
user call
Authenticate
Tool call
Authenticate user
Amazon Bedrock
LLMs
AgentCore Memory
AgentCore Identity
Amazon Cognito

	SWARM (multi-agent collaboration)	Hierarchical	Competitive	Multi-agent DAG
Handoff	Dynamic and peer-to-peer, based on agent discoveries and needs	Role-based; tasks are passed from the orchestrator to the appropriate worker	Task is passed from the orchestrator to all workers	Pre-determined
Entry point	Default or last active	Leader	Leader	Workflow input
Decision making	Distributed; agents make local decisions about task planning and handoffs	Centralized; the orchestrator makes the overall decisions and routes tasks	Centralized; the orchestrator makes the overall decisions and routes tasks	Predefined paths
Interaction	Any to any agent	Agent <-> leader	Agent <-> leader	State machine
Goal	Different; Each agent has a different goal	Different; Each agent/team has a different goal	Shared; Each agent/team has the same goal	Different; Each agent/team has a different goal
Predictability	Low	Medium	Medium	High
Use case	Exploration, collective adaptation	Centralized decisions	Compare different solutions for one problem	Multistep automation

Questions?

Please complete the session survey in the mobile app

Example: research system architecture

Lead agent responsibilities

- Analyze user query
- Develop research strategy
- Create specialized subagents
- Synthesize results
- Decide if more research needed

Subagent responsibilities

- Execute specific search tasks
- Iterate on findings
- Use interleaved thinking
- Filter and compress information
- Return focused results

Context engineering tips

Think like our agents

Agents can see 50+ subagents for simple queries without clear guidance

Scale effort to complexity

Simple fact-finding: 1 agent with 3-10 tool calls

Complex research: 10+ agents with divided responsibilities

Start wide, then narrow

Start with short, broad queries. Evaluate results. Then progressively narrow focus based on findings

Key principles

Teach delegation

Give objectives, output format, tool guidance, and clear boundaries. Vague instructions cause duplicated work

Tool design critical

Examine all tools first. Match tool to intent. Bad tool descriptions send agents down wrong paths

Let agents self-improve

Frontier models can diagnose failures and suggest improvements.

Production engineering challenges

Stateful Execution

Agents maintain state across many tool calls. Need durable execution, error recovery, and checkpoints. Can't restart from beginning - too expensive. Solution: Resume from error points, let agents adapt to tool failures gracefully

Debugging Complexity

Non-deterministic decisions make debugging hard. Need full tracing to see search queries, source choices, and tool failures. Monitor decision patterns while maintaining privacy. High-level observability reveals root causes

Deployment Strategy

Stateful agents can be anywhere in their process during updates. Use rainbow deployments to gradually shift traffic. Can't update all agents simultaneously without breaking running processes

Evaluation strategies for multi-agent systems

Start Small

- Begin with 20 real-world test cases
- Large effect sizes visible quickly
- Iterate rapidly with small samples
- Scale evaluation as system matures

LLM-as-Judge + Human

- LLM judges for scalability (factual accuracy, citations, completeness)
- Human testing for edge cases and subtle issues
- Focus on end-state vs step-by-step process
- Expect emergent behaviors from interactions

Performance and token economics

- Token usage explains 80% of performance variance
- Multi-agent systems use 15x more tokens than chat
- Parallel tool calling cuts wall time
- Best for high-value tasks that justify token cost
- Not suitable for all domains - coding has fewer parallelizable tasks

When to Use Multi-Agent Patterns

Good Fit

- Open-ended research tasks
- Breadth-first exploration
- Tasks exceeding context windows
- Heavy parallelization possible
- Multiple specialized tool sets
- High-value complex problems

Poor Fit

- All agents need same context
- Many dependencies between agents
- Simple single-path tasks
- Low-value routine operations
- Real-time agent coordination needed
- Cost exceeds value delivered

Key takeaways

Architecture patterns

Choose pattern based on coordination needs: shared scratchpad, supervisor delegation, or hierarchical teams.

Engineering excellence

Invest in prompt engineering, tool design, evaluation, and observability. These are your primary levers for improving agent behavior.

Production readiness

Last mile often becomes most of the journey. Production requires careful state management, error handling, and deployment strategies

ORCHESTRATOR-WORKER

Pattern 1: Hierarchical

Lead agent coordinates specialized workers

Each worker has independent context/tools

Workers return only final results to supervisor

Supervisor acts as intelligent router

Can delegate, monitor, and synthesize results

SHARED SCRATCHPAD

Pattern 2: Multi-agent collaboration

Agents share a common workspace/memory

All actions visible to all agents

Simple router controls state transitions

Best for: Tasks requiring full context sharing

Trade-off: Verbose information passing, but complete transparency across agents

SWARMS

Pattern 3: Competitive

- Variant of hierarchical orchestrator-worker
- Sub-agents spawned in swarms, either collaborative or competitive
- Collaborative agents attempt to fill in gaps

Competitive agents try different solutions

Orchestrator synthesizes and scores

A2A

Pattern 4: Peer-to-peer

Clean handoffs from one agent to another

Use A2A for RPC agents, usually organizational or SDLC boundaries

Multi-Agent Systems Defined

Multiple independent actors powered by LLMs

Each agent has specialized prompts, tools, and capabilities

Agents are connected in a specific architecture

Coordinate through shared state or message passing

Autonomous decision-making within their scope

Graph-based representation: nodes = agents, edges = connections

Thank you

Please complete the session survey in the mobile app