Engineering Production-Grade Multi-Agent Systems in the Cloud

Conf42 Cloud Native 2026

Sandeep Bharadwaj Mannapur · AI & ML Engineer

The Real Conversation Happening Right Now

Four questions every team hits when they move agents to production — and they're not model questions. They're cloud engineering questions.

Orchestration Complexity

How do you manage coordination logic that spans multiple services and agent layers?

Latency Compounding

Each agent hop adds a full round-trip cost. How do you keep the system responsive?

Failure Propagation

A partial error in one agent corrupts reasoning state in another. How do you isolate that?

Non-Determinism at Scale

Autonomous outputs vary. How do you govern a system you cannot fully predict?

⚠ These problems show up in staging. They break you in production. And the answer is never a better prompt — it's always a better architecture.

What We'll Cover

01

The Monolith Problem

Why single-agent architectures fail under production load

02

Orchestration Architecture

When centralized control outperforms decentralized agent coordination

03

Failure Domain Isolation

Engineering boundaries that prevent cascading errors

04

Observability for Reasoning Chains

Capturing decision artifacts across distributed multi-hop paths

05

From Local to Cloud

What actually changes when you move agents into production infrastructure

06

Governance & Trade-offs

Balancing autonomy and determinism when agents influence production

The Monolith Problem

Why Single-Agent Architectures Break in Production

What Demos Show	What Production Exposes
Clean, linear request-response flows	Context windows overflow on complex workflows
A single model handles all reasoning	A single model cannot specialize across domains
Happy-path completions	One failure collapses the entire pipeline
No concurrent load, no partial state	No isolation means no independent scalability

As ticket volume and workflow complexity grew, the cracks became critical failures:

Context Overflow

Reasoning chains grew too long. Context windows overflowed mid-conversation.

Tool Reliability Collapse

A model asked to do too many things started making errors on all of them.

Zero Fault Isolation

A single prompt failure derailed an entire workflow. No isolation. No recovery.

> The insight: the problem was not that we needed a smarter model. We needed a better architecture.

CHAPTER 2

Orchestration Architecture

Centralized Orchestration vs. Decentralized Agent Swarms

Centralized Orchestration

A single orchestrator directs task delegation, manages state, and enforces execution order.

- Deterministic control flow
- Explicit retry logic
- Clear accountability for workflow outcomes

Decentralized Agent Swarms

Agents self-coordinate using shared state or message passing without a central authority.

- Maximizes parallelism
- Emergent behavior becomes difficult to predict
- Contradictory outputs from peer agents are harder to resolve

The Production Verdict: For operational workflows where outputs influence real business actions — centralized orchestration provides the predictability and traceability that swarms cannot.

Anatomy of the Orchestrated Architecture

Orchestrator Agent

Receives requests, decomposes tasks, routes work — never executes domain logic itself

Specialist Agents

Domain reasoning with bounded scope, bounded context window, and owned tools

Tool Layer via MCP

Standardized access to APIs and databases — build once, connect to any agent

A2A Communication

Cross-service, framework-agnostic agent delegation

Response Aggregator

Compiles, validates, finalizes output — the last gate before production systems

Each agent operates on a well-defined context window — reducing hallucination risk and simplifying failure attribution.

CHAPTER 3

Four Patterns for Failure Isolation

Engineering Boundaries That Prevent Cascading Failures

1

Circuit Breakers

Each agent call is wrapped in a circuit breaker. After a failure threshold, the breaker opens and the orchestrator routes to a fallback path — preventing retry storms from amplifying downstream pressure.

2

Agent Sandboxing

Agents operate in isolated execution contexts. State is never shared in-memory — all inter-agent communication passes through a durable message bus, enforcing clear boundaries.

3

Timeout Budgets

Every agent invocation carries an explicit timeout budget propagated from the orchestrator. Agents that exceed their budget are terminated deterministically — graceful degradation, not indefinite hangs.

4

Fallback Reasoning Paths

Critical workflows maintain a simplified fallback agent — lower-capability, minimal tooling — that activates when primary agents fail, ensuring service continuity under degraded conditions.

⚠ We learned timeout budgets the hard way. An upstream API degraded from 200ms to 45 seconds. Without explicit budgets, the orchestrator waited — 45 seconds per request, hundreds of concurrent workflows, all hanging.

CHAPTER 4

Observability

Observability Must Capture Decisions, Not Just Metrics

What standard infrastructure observability gives you:

CPU, memory, request latency, error rates, pod health — sufficient for stateless microservices.

Why it's not enough for agents:

The path of reasoning — not just the outcome — determines correctness. A multi-agent system can return a response with correct formatting and completely wrong reasoning. Metrics cannot catch that.

What multi-agent observability must capture:

- Which agent was invoked and why
- What context was passed — and what was omitted
- Which tools were called and what they returned
- How the orchestrator resolved conflicting agent outputs
- Full reasoning chain across all hops

How to get there:

Instrument each agent with a structured trace schema correlated through a shared identifier across all hops. When something goes wrong, you reconstruct the complete reasoning chain — not just the entry and exit, but every decision in between.

> Standard distributed tracing records service calls. It does not record what the model decided. Multi-agent systems need both.

CHAPTER 5

Local vs. Cloud — What Changes

From Local to Cloud

Concern	Local Development	Cloud Production
State persistence	SQLite on disk	PostgreSQL / DynamoDB — distributed, survives container restarts
Tool access	In-process function calls	Containerized MCP microservices with independent scaling
Agent communication	In-process or localhost A2A	A2A services as cloud functions or K8s pods with HPA
Orchestrator	Single-process	Stateless containers — state lives externally
Inference	Local model (Ollama)	Managed inference behind LiteLLM gateway
Deployment	Docker Compose	Kubernetes with per-agent resource envelopes
Observability	Local Langfuse	Managed distributed tracing across services
Cost	Zero	Token cost compounding — requires per-agent budgets

ⓘ Running a multi-agent system locally is a solved problem. Running it reliably in cloud infrastructure is where engineering discipline separates working systems from brittle demos.

Three Cloud-Specific Failure Modes Nobody Talks About

1 — Token Cost Compounding

A 4-agent system with 5 tool calls per agent is 20+ LLM calls per request. At cloud inference pricing, that's a cost engineering problem before it's an architecture problem. You need per-agent token budgets enforced at the orchestrator level — not as guidelines, as hard limits.

2 — Checkpoint State & Rolling Deploys

When you deploy a new version of Agent B, there are in-flight workflows checkpointed against the old state schema. Without migration-safe state design, those workflows fail in production during deployment. Design your checkpoint schema for backward compatibility from day one.

3 — Cold Start Latency on Agent Containers

Agent containers with model dependencies can take 30–60 seconds to cold start. At production request rates, cold starts become a service reliability issue. Either keep containers warm (cost trade-off) or design fallback paths that tolerate cold start delays gracefully.

⊗ Each of these appeared in production. None of them appear in tutorials.

The Technology Stack That Makes This Real

Not prescriptive — but specific. These are the tools validated under production load.

Layer	Technology	Purpose
Orchestration	LangGraph	Stateful agent graph — checkpointing, conditional routing, HITL
Tool Integration	MCP (Model Context Protocol)	Standardized agent-to-tool contracts
Agent Coordination	A2A Protocol	Cross-service agent delegation — framework-agnostic
Inference Gateway	LiteLLM	Unified model access — rate limiting, fallback routing, cost tracking
Observability	Langfuse	Distributed tracing across agent hops — decisions, not just metrics
Evaluation	DeepEval	LLM-as-judge quality gates before production deployment

The patterns are framework-independent. The tools are the current best-in-class implementations of those patterns.

CHAPTER 6

Governance

An Engineering Requirement, Not a Compliance Checkbox

When autonomous agents influence production workflows — triggering updates, initiating escalations, executing actions — governance transitions from a compliance concern to an engineering requirement.

Output Validation Gates

Structured outputs pass through schema validation and policy rules before being applied to production systems. Invalid or out-of-policy outputs are rejected and logged — never silently applied.

Human-in-the-Loop Thresholds

Actions above a defined confidence threshold or risk level require human approval. The orchestrator routes these to a review queue — the system continues for low-risk decisions, escalates for high-risk ones.

Immutable Audit Logs

Every agent decision resulting in a production action is persisted with full context — enabling forensic review, rollback analysis, and regulatory compliance.

Key Takeaways

Engineering Principles for Production Multi-Agent Systems

→ The problem is engineering, not prompting

Reliability at scale requires architectural discipline — orchestration design, failure isolation, observability. Not better system prompts.

→ Centralized orchestration earns its overhead

For operational workflows, the predictability and traceability of a central orchestrator justify the coordination cost over swarm approaches.

→ Observability must be decision-aware

Capture reasoning artifacts at every agent hop. Infrastructure metrics alone cannot reconstruct what went wrong in a multi-agent reasoning chain.

→ Cloud deployment is a different engineering problem

Token cost compounding, checkpoint migration, cold start latency — design for them from the start.

→ Governance is a first-class concern

Once agents influence production systems, output validation, human-in-the-loop escalation, and audit logging are non-negotiable.

Latency vs Accuracy	Autonomy vs Auditability	Generality vs Reliability

Three tensions you will navigate: Latency vs Accuracy · Autonomy vs Auditability · Generality vs Reliability

Thank You

Sandeep Bharadwaj Mannapur

AI & ML Engineer · 15+ years in enterprise-scale AI/ML systems

Conf42 Cloud Native 2026

Questions welcome.

The architectural patterns discussed in this talk are available as open reference implementations.