Galileo

v1.01

Mastering

Multi-Agent Systems

Real-World Strategies for Multi-Agent Development

Mastering GenAI Series

→ galileo.ai

Preface

So far, the industry has assumed that a single, sufficiently powerful model could handle complex reasoning, coordination, and execution. In practice, people quickly discovered that more data, longer context windows, and better prompts weren't enough to reliably manage complex, interconnected tasks.

Multi-agent systems take a fundamentally different approach. Instead of relying on one all-purpose model, they distribute work across specialized agents. Tasks that once overwhelmed a single model become manageable through parallel processing, specialized expertise, and built-in validation.

However, the same collaboration that makes these systems powerful also makes them more difficult to manage. Each agent needs context, memory, and clear communication rules—challenges that multiply as systems scale.

This book will prepare you to face these challenges head-on. You'll learn when multi-agent systems add real value, how to design them efficiently, choose the right architecture, and build reliable systems that work in production.

The book covers five chapters:

Chapter 1: outlines seven primary advantages, and when these benefits justify the added complexity

Chapter 2: examines how coordination costs can erode those benefits and provides a framework to determine if your project truly needs multiple agents

Preface

Chapter 3: describes four key architectures (centralized, decentralized, hierarchical, and hybrid) and helps you choose the right structure

Chapter 4: focuses on context management, explains how it differs from memory, and outlines four common failure points with practical solutions

Chapter 5: provides a step-by-step LangGraph example showing how to build, test, and monitor a production-ready system

If you lead technology or innovation efforts, this book will help you evaluate when multi-agent systems are worth the investment and provide you with a roadmap to make them reliable.

Pratik Bhavsar

(@ptkbhv / GitHub / LinkedIn)

Contents

01

Benefits of Multi-Agent Systems

02

Why Multi-Agent Systems Fail

Contents

03

Architectures for Multi-Agent Systems

04

Context Engineering for Multi-Agent Systems

Contents

05

How to Continuously Improve Your LangGraph Multi-Agent System

Galileo

Chapter

01

Benefits of Multi-Agent Systems

01 Benefits of Multi-Agent Systems

A passenger writes:

My flight is delayed 3 hours and I'll miss my connection to Tokyo. Can you rebook for me?

Your AI agent takes 20 seconds and suggests a route with a 14-hour layover—even though there's a better option with just 2 hours. It confirms the rebooking but doesn't mention their upgrade won't transfer to the new flight. This same system answers simple questions instantly. *Is my flight delayed?* Two seconds. *What's my gate number?* Two seconds. But the moment someone needs help with a real problem that touches multiple systems, it breaks down.

This isn't a prompt engineering problem. You can't fix it with better examples, temperature adjustments, or a longer system prompt. The problem is architectural: one model trying to coordinate multiple data sources, and perform distinct operations simultaneously. Meanwhile, the companies shipping reliable AI products are doing something different. They're using teams of specialized AI agents that work together, each handling what they do best, instead of forcing one model to do everything.

This chapter will show you why coordinated AI agents solve problems that single models struggle with. You'll learn which scenarios actually benefit from multi-agent systems, and understand the tradeoffs involved.

When Specialization Beats Generalization

When building AI systems, you constantly face this choice: use one model that handles everything or break tasks down into specialized components. The difference becomes most apparent when you compare how each approach handles complex, multipart requests.

A good analogy for this would be that of a busy restaurant. You could have one cook trying to prep ingredients, work the grill, make desserts, and handle orders all at once. They'd do a mediocre job at everything, and customers would have to keep waiting. Alternatively, you could have a prep cook, a line cook, and a pastry chef, each focused on what they do best. Orders move faster, and the food tastes better because each person can focus on their specialty.

Let's walk through a real example to see why this matters for your applications.

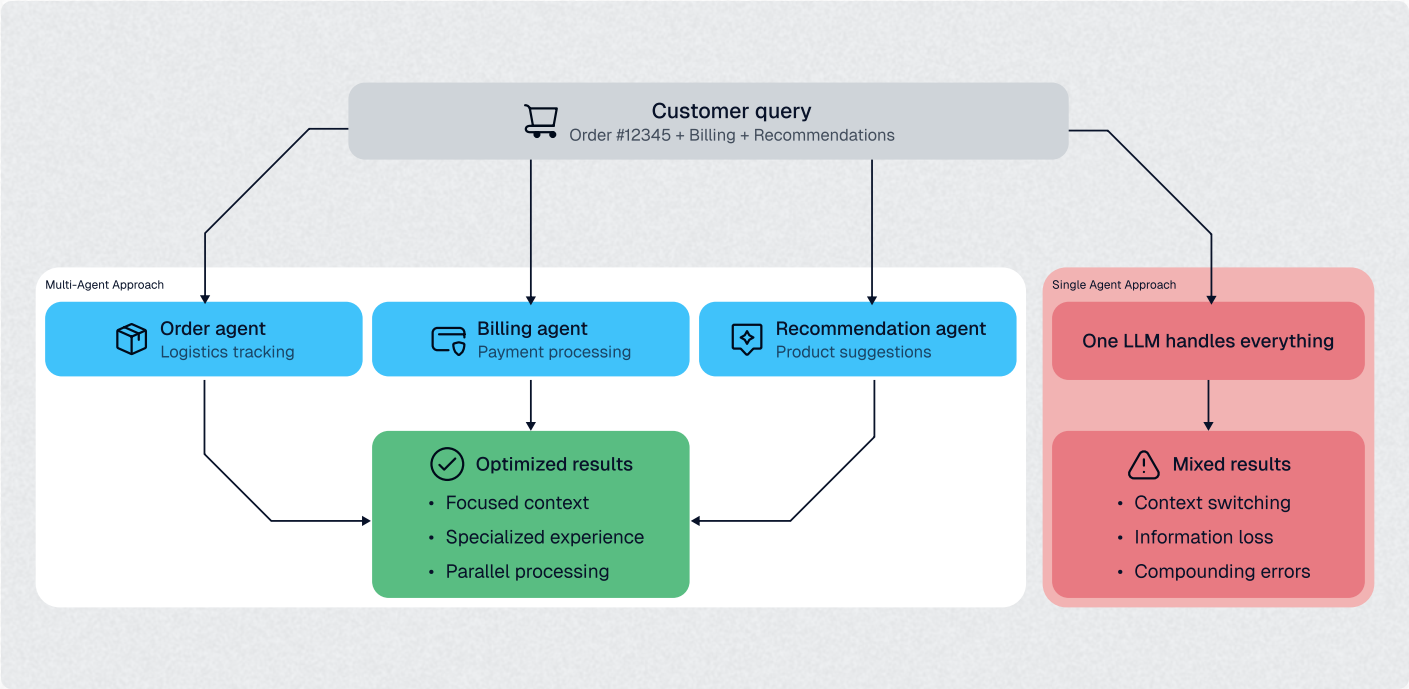


FIGURE 1.1 Single-agent vs multi-agent architecture comparison for handling complex e-commerce customer queries

Single-Agent Approach

When you ask one LLM to handle an entire e-commerce customer inquiry `(FIGURE 1.1)` like *My order #12345 hasn't arrived, and I noticed you charged me twice. Also, can you recommend similar products to what I ordered?* you're asking a single model to manage order tracking, payment processing, and product recommendations all at once. These three are entirely different jobs that require distinct skills.

This approach produces mediocre results across all three tasks. Your model loses context as it switches between database queries, financial calculations, and recommendation algorithms. As the AI loses context, it drops information, and new errors build up throughout the process. You may have seen this happen in your own systems when handling complex requests.

Multi-Agent Approach

Now consider what happens when you split this same inquiry across specialized agents:

Each agent works well at its specific task while maintaining a focused context that reduces hallucinations. Your Order Agent uses logistics-specific prompts and maintains shipping context. Your Billing Agent focuses solely on transaction records with financial calculation optimizations. Your Recommendation Agent analyzes purchase patterns without getting drawn into payment disputes.

→ Order Tracking Agent

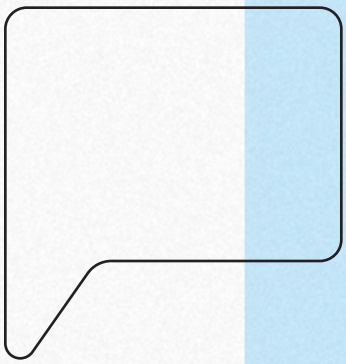
Checks logistics databases, provides precise shipping updates

→ Billing Agent

Reviews payment records, identifies duplicate charges, and initiates refunds

→ Recommendation Agent

Analyzes purchase history, suggests relevant alternatives



Model selection flexibility provides the key advantage. Your Math Agent can use Claude for reliable calculations at a temperature of 0.1. Your Creative Agent uses GPT-4 at temperature 0.9 for marketing copy. Your Summary Agent runs on Gemini to cut costs. You match tools to tasks instead of forcing one model to handle everything.

You might have heard about the router in GPT-5. Here's how the math works: if 70% of your queries are simple FAQs, why send them to an expensive reasoning model? Route them to a lightweight model that costs 1/100th as much. Save your premium models for the 5% of queries that actually need complex reasoning. This is how production systems get 60% cost reductions while keeping quality up. You can use our agent leaderboard to help you pick cost-effective models for agents based on your complexity needs.

So far, we've seen how specialization solves the basic problem of trying to make one model do everything well. Each agent focuses on what it does best, and you can match the right model to each task. But there's another problem that single agents struggle with: they have no way to check their own work. This gets us to our next advantage of multi-agent systems.

Validation Through Orthogonal Checking

Even the best models make mistakes, and you've likely experienced this firsthand. The problem with single agents is that they have no mechanism for self-correction. When they generate wrong information, they often sound just as confident as when they're right. This makes the errors particularly dangerous and here's where having multiple agents check each other's work becomes very valuable.

You've probably done this many times. Say, you write a proposal or report on your own and read through it multiple times, and it looks perfect to you. Still, when you send it to a colleague for review, they may spot a few typos, unclear sentences, and gaps in logic that you completely missed. You'll see this better in FIGURE 1.2.

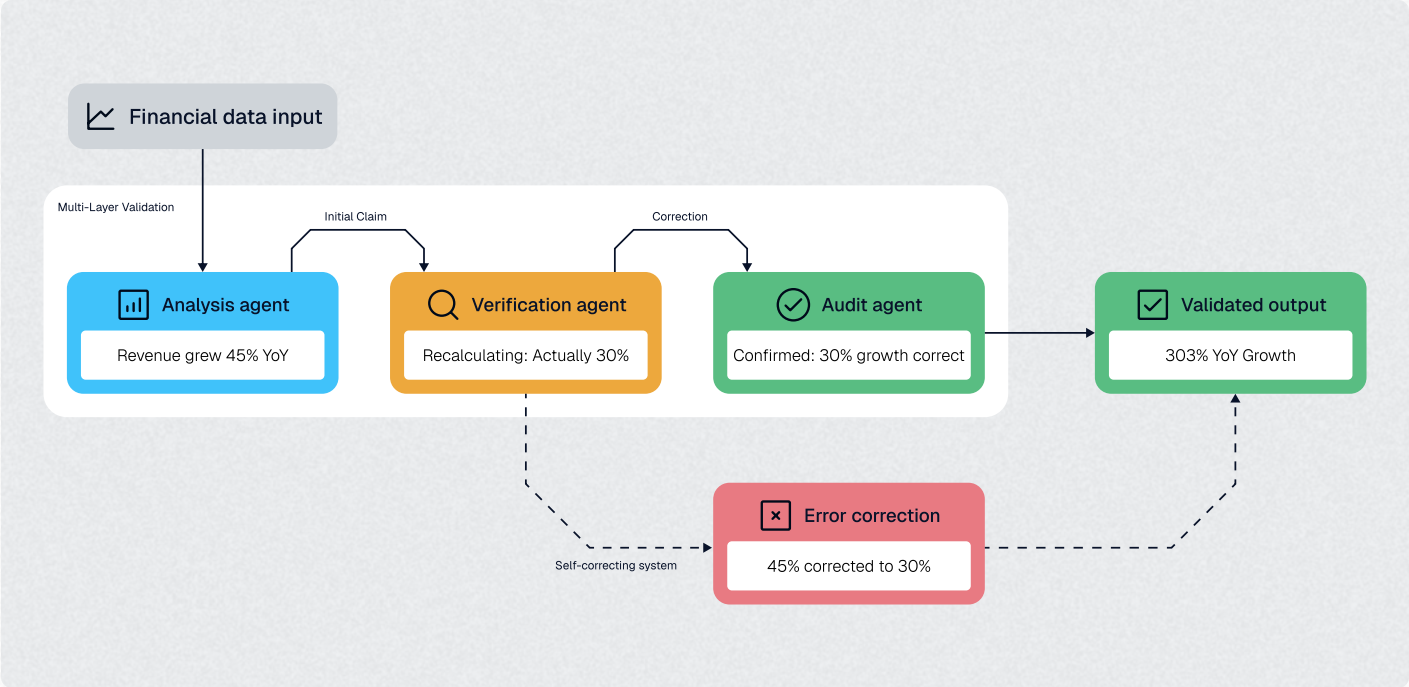


FIGURE 1.2 Multi-layer validation system with peer review between specialized agents

Single-Agent Approach

A single LLM analyzing financial data might confidently state: "Based on the Q3 reports, revenue grew 45% year-over-year." If this calculation is wrong, there's no internal way to catch the error. Your model has no self-doubt, no verification step, and no way to question its own output. It can then end up giving incorrect information with the same confident tone it uses for correct ones, which can mislead the user.

Multi-Agent Approach

But what if you had multiple agents that could double-check each other's work? The same analysis with validation layers works like this:

→ Analysis Agent

"Revenue grew 45% YoY"

→ Verification Agent

"Wait, let me recalculate: Q3 last year was $10M, this year $13M, that's 30% growth."

→ Audit Agent

"Confirmed: 30% growth is correct. The 45% figure was comparing different quarters."

Your system self-corrects through peer review. Each agent validates different aspects:

- → Generation Agent creates the initial response
- → Logic Agent checks reasoning consistency
- → Fact Agent verifies claims against knowledge
- → Safety Agent checks content appropriateness

EXERCISE

Think about a recent AI output from your organization that contained an error. How many validation agents would you need to catch similar errors? List the specific checks each agent would perform.

This peer review approach is effective in real-world systems. In fact, Anthropic's Constitutional AI shows this principle in practice. One model generates responses while another critiques them based on constitutional principles. The critique-revision loop catches outputs that the initial model misses.

Sequential validation gates have changed the way we handle errors in production systems. For simplicity, you can consider the example of claim processing, a popular use case for AI in insurance:

1. **Intake Agent**
   Captures details
   (catches incomplete data)
2. **Validation Agent**
   Checks requirements
   (prevents downstream errors)
3. **Fraud Detection Agent**
   Analyzes patterns
   (flags suspicious claims)
4. **Approval Agent**
   Makes decisions (only sees
   pre-validated claims)

Each gate prevents errors from spreading through your system. A single model trying to handle all these checks at once often misses edge cases that specialized validators catch. The advantage of this approach is that you can add or remove validation steps based on the level of accuracy required for your specific use case.

Validation solves the accuracy problem, but there's still the issue of speed. When you have a large batch of work, say, analyzing hundreds of customer reviews or processing multiple documents, a single agent works through them one by one. Even with accuracy, this sequential approach creates a bottleneck that slows down your entire system.

Parallel Processing for Scale

Processing large volumes of data one by one hits two walls: time limits and context limits. Your model slows down as it works through each item, and it starts forgetting early information as the context window fills up. Multi-agent systems solve this by dividing the work and processing it simultaneously.

Think of it this way: It's like having 500 emails in your inbox. If you read them one by one, it takes hours, and by the time you reach the end, you've forgotten what the first ones were about. But if you had three assistants help you, each taking 125 emails to sort through at the same time, you'd finish in a fraction of the time and could compare patterns across all the emails when they bring you their summaries.

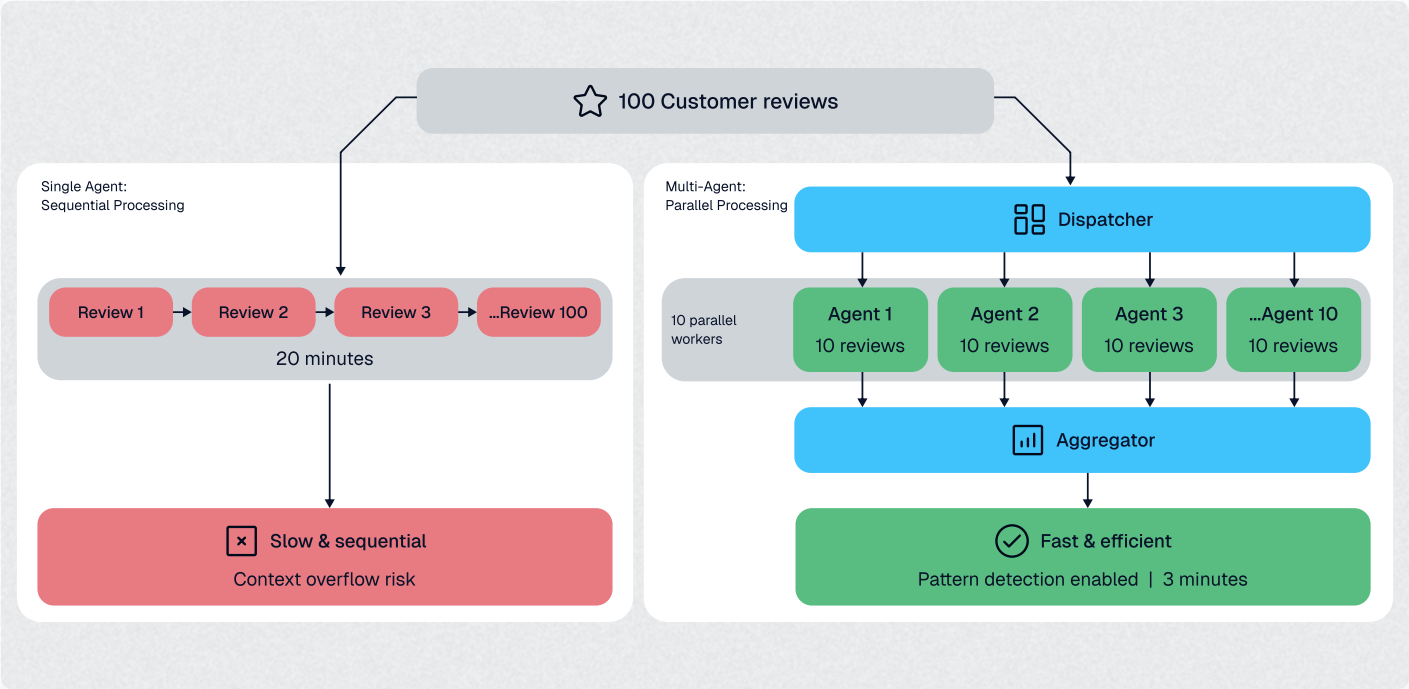


FIGURE 1.3 How multi-agent systems achieve dramatic speedups through parallelization

Single-Agent Approach

A single LLM working through customer reviews works like a lone analyst reading through feedback forms one by one (FIGURE 1.3). It starts with review #1, analyzes the sentiment, writes a summary, and then moves to review #2. By the time it reaches review #50, it's struggling to remember patterns from review #5. The context window fills up with processed reviews, leaving less room for new analysis.

→ Review 1 → Sentiment analysis → Summary →
→ Review 2 → Sentiment analysis → Summary →
→ Review 3 → Sentiment analysis → Summary →
→ ... continues for 20 minutes

Your model maintains a growing context, risks token limit overflow, and processes each review in isolation. It can't identify patterns across the dataset because earlier reviews get pushed out of context. After 20 minutes of processing, you get individual summaries but miss the bigger picture, like discovering that 40% of complaints mention the same shipping issue.

Multi-Agent Approach

You can divide and conquer, instead of processing everything one by one. Here's how the same workload gets handled with parallel agents:

→ Dispatcher Agent

Splits reviews into batches

→ 10 Analysis Agent

Process 10 reviews each at the same time

→ Aggregator Agent

Combines results into a final report

You go from 20 minutes down to 3 minutes. But the speed boost isn't the only advantage. When agents work in parallel, they can spot patterns across all the reviews that a single model would miss because it has forgotten the early ones by the time it reaches the end.

This same parallel processing principle applies not only to data analysis but also to other areas. Copilot and Cursor demonstrate this in practice. They analyze entire codebases at the same time, understand dependencies across files, and suggest multi-file edits in parallel. One agent updates function signatures while another fixes all calling locations. A third updates tests. This parallel approach makes refactoring possible that would be impractical sequentially.

The parallel processing advantage becomes even more pronounced as your data volumes grow. While a single agent's performance degrades with more data, parallel agents maintain consistent speed regardless of scale. But what happens when something breaks? In a single-agent system, one failure can bring down your entire workflow. Multi-agent systems handle this differently, as they're designed to continue running even when individual components fail.

Fault Tolerance

Single points of failure can bring down entire workflows. When your one model crashes, times out, or produces substandard output, everything comes to a halt. Multi-agent systems handle this differently by distributing the work and building in resilience from the start. If you've ever had a critical workflow fail because one component went down, you'll surely appreciate this approach.

For example, if you're doing everything yourself and get sick, the whole project comes to a halt. But if you're working with a team where each person handles different parts, the project keeps moving even if one person can't contribute. The other team members pick up parts of your tasks and help you deliver most of the work on time instead of missing the deadline completely.

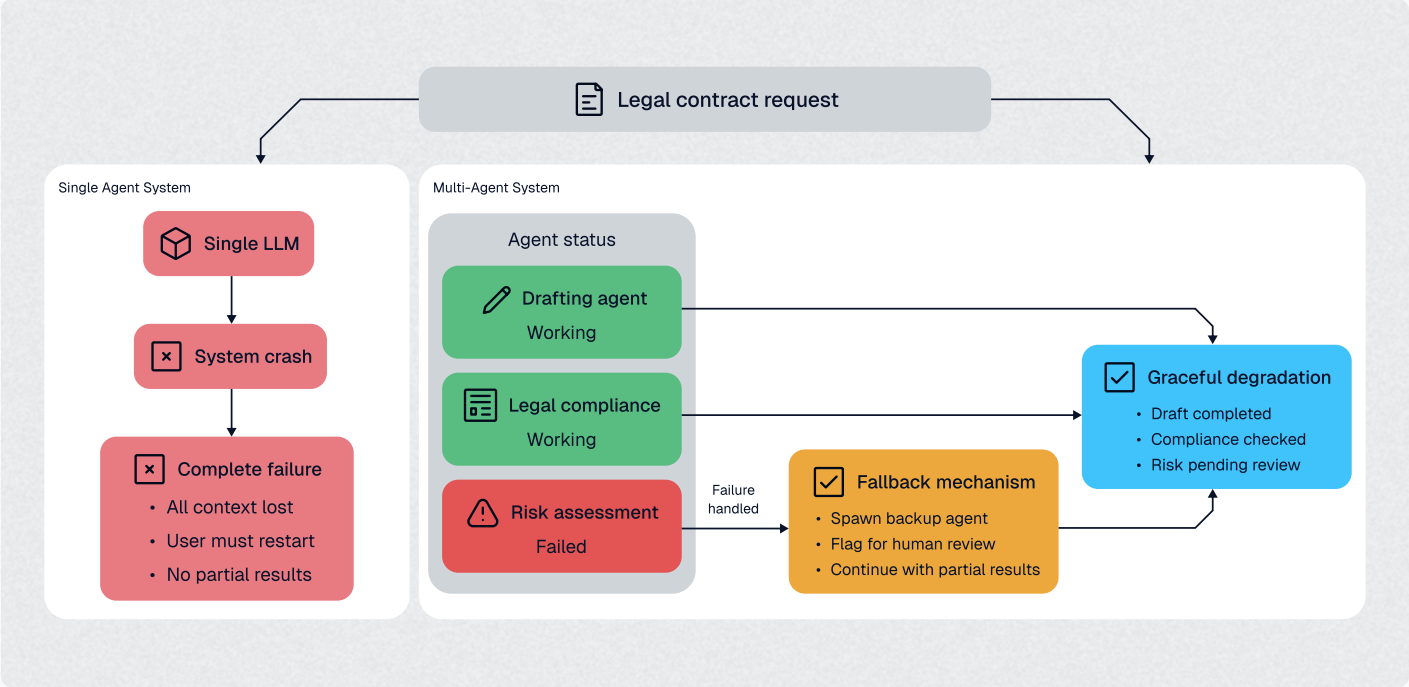


FIGURE 1.4 Graceful degradation vs catastrophic failure in single-agent and multi-agent systems

Single-Agent Approach

A single model handling all tasks creates a single point of failure. When that model crashes or produces incorrect output, your entire workflow stops. The user loses all context and must start over. You'll see this happen in FIGURE 1.4.

Multi-Agent Approach

With multiple agents, failure becomes manageable instead of catastrophic. This risk goes down substantially when you use a multi-agent approach:

- → Drafting Agent
  Creates initial contract
- → Legal Compliance Agent
  Reviews for regulatory issues
- → Risk Assessment Agent
  Identifies potential liabilities

If your Risk Assessment Agent fails, the system can:

- → Continue with other agents' work
- → Spawn a backup risk agent
- → Flag the specific issue for human review

In this scenario, the partial work isn't lost. Your user gets a draft contract with compliance review, plus a note that risk assessment is pending.

Single failures are manageable. But what happens when an agent keeps failing? The system needs fallback strategies:

→ Primary Analysis Agent times out → Switch to Backup Analysis Agent

→ Backup also fails → Return cached results with staleness warning

This way, your user gets a degraded but useful response, not an error.

Take the example of a customer service system where the Payment History Agent fails due to database maintenance (in production). In a single-model system, the entire interaction fails. In a multi-agent system, you still provide shipping updates and product recommendations. The response acknowledges the limitation: "Your order ships tomorrow. I'm unable to access payment history right now, but here are similar products you might like."

This graceful degradation approach ensures that your users always receive some value, even when parts of your system are experiencing issues.

We know that fault tolerance keeps your system running, but you're still treating every request the same way. A simple question about business hours doesn't need the same computational power as a complex technical problem.

Dynamic Routing Based on Confidence

Not all queries are equal. A simple “What are your business hours?” shouldn’t cost the same to process as a complex technical troubleshooting request. Multi-agent systems can analyze incoming queries and route them to the appropriate level of processing power and expertise. This is where you can start seeing real cost savings while improving response quality.

When you call your doctor’s office, the receptionist doesn’t transfer everyone to the head physician. “I need to refill my blood pressure medication,” gets handled by the pharmacy tech in 30 seconds. “I’m having chest pains” gets you transferred immediately to a nurse who can assess urgency and connect you with the doctor. In the same manner, “I’m feeling anxious about my diagnosis” gets you routed to someone trained in patient care who can provide emotional support while also checking if you need medical follow-up.

Each type of call gets exactly the right level of attention in the above scenario. You’ll see something similar happen in FIGURE 1.5.

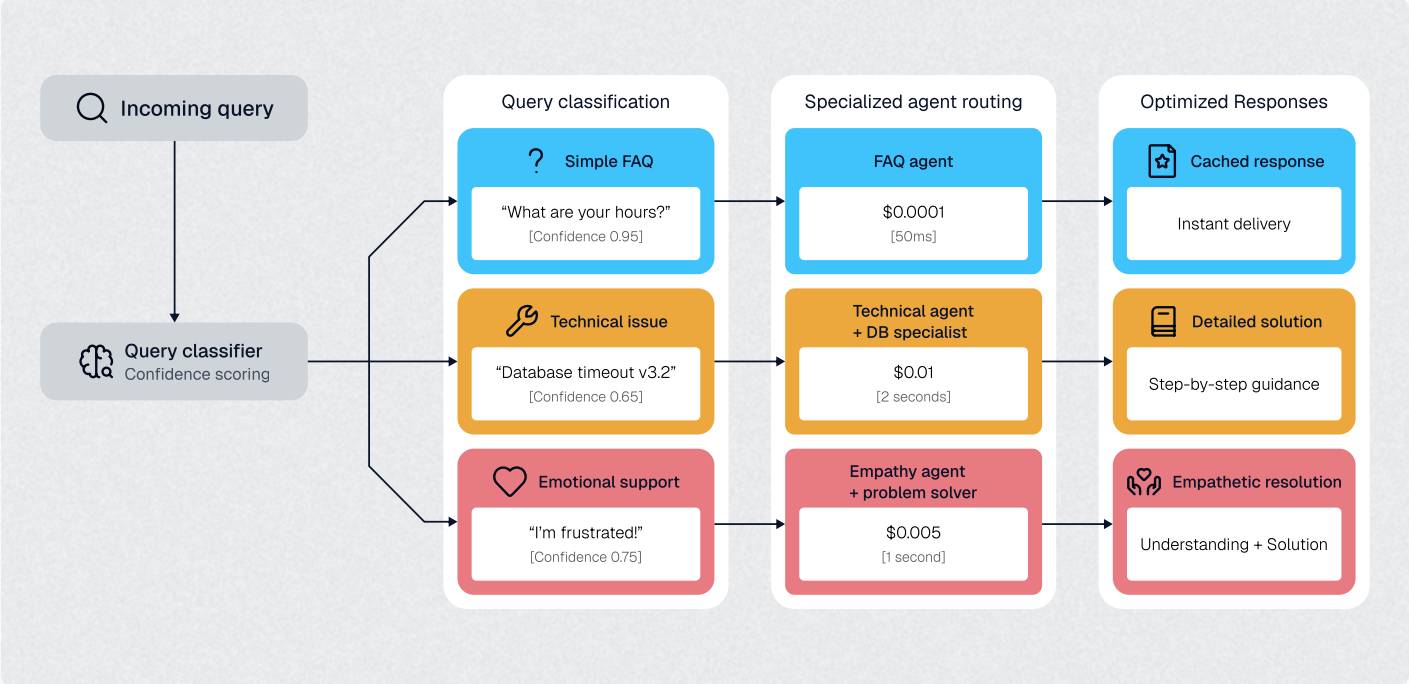


FIGURE 1.5 How multi-agent systems route queries to appropriate expertise levels

Single-Agent Approach

You typically treat every query the same way. Simple FAQs and complex technical issues use the same model and approach. There's no adaptation based on query complexity or model confidence.

Multi-Agent Approach

Smart routing changes this completely. Your system dynamically adjusts based on query classification:

For "What are your business hours?"

- → Routes to FAQ Agent
- → Uses cached response
- → Costs $0.0001, responds in 50ms

For "My database connection keeps timing out after upgrading to v3.2"

- → Routes to Technical Support Agent
- → Spawns Database Specialist Agent for backup
- → Costs $0.01, responds in 2 seconds with detailed troubleshooting

For an angry customer expressing frustration

- → Empathy Agent handles emotional response
- → Problem-Solving Agent works on the actual issue
- → Both collaborate on a unified response

Confidence-based escalation makes progressive automation possible:

- → High confidence (>0.9): Fully automated response
- → Medium confidence (0.7-0.9): Agent response with human review option
- → Low confidence (<0.7): Route to specialized agent or human

You can gradually adjust instead of going all-or-nothing with automation. For example, you start conservative, gradually increasing automation thresholds as you gather data and your system learns which queries it handles well and which need help.

EXERCISE

Map out your current queries by complexity. What percentage could be handled by a lightweight FAQ agent? What percentage need specialized routing? Use this to estimate potential cost savings from dynamic routing.

The key insight here is that routing intelligence becomes a competitive advantage. While most use expensive models for simple questions, you're optimizing costs without sacrificing quality.

But what about conversations that span multiple exchanges? Single models tend to forget early parts of long conversations as their context window fills up. Multi-agent systems solve this by separating memory management from response generation. Let’s see how.

Context Preservation Across Interactions

Long conversations expose a fundamental limitation of single models: they forget. As conversations grow longer, important information from early messages gets pushed out of the context window. Multi-agent systems handle this by separating conversation management from response generation. If you've built around conversational AI, you know how frustrating this limitation can be for users.

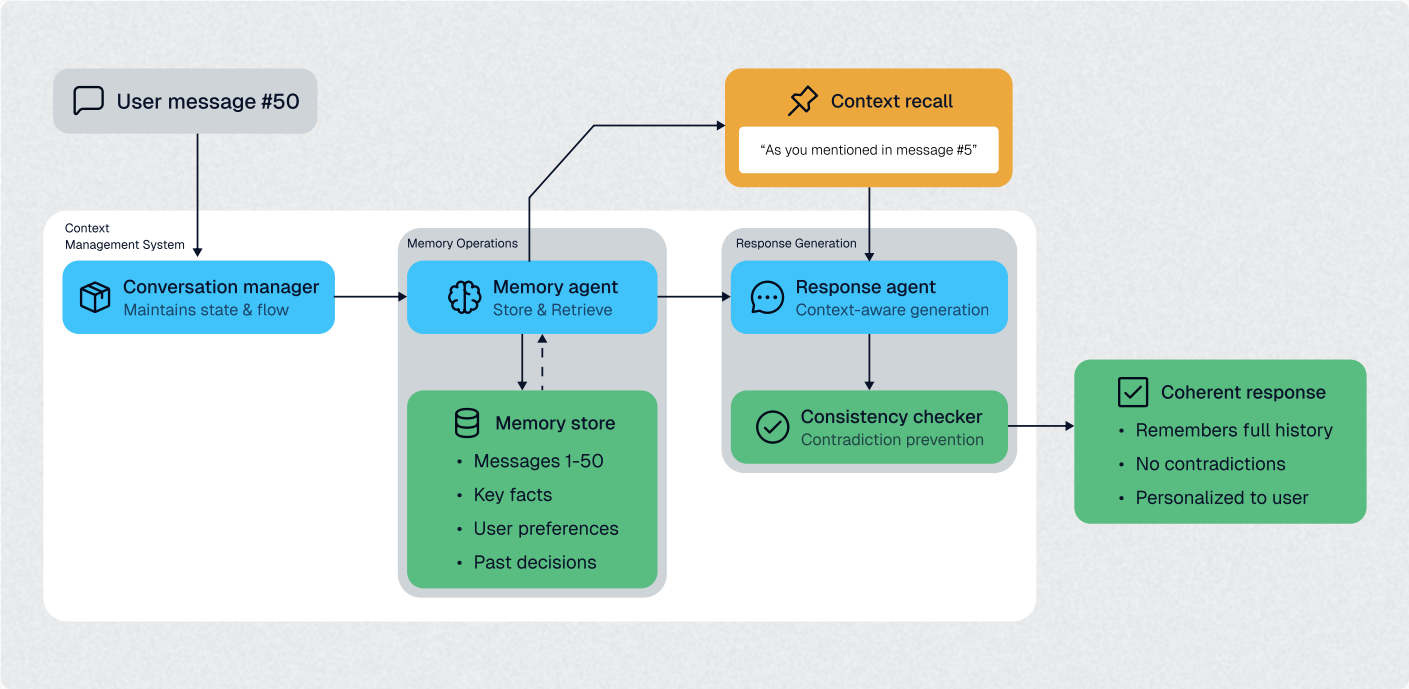


FIGURE 1.6 Context preservation across extended conversations in multi-agent vs single-agent systems

Single-Agent Approach

In a long conversation, your single LLM might forget earlier context or contradict itself as the context window fills up. By message 50, it has no memory of message 5. Critical information gets pushed out of the context window. (FIGURE 1.6)

Multi-Agent Approach

- → **Conversation Manager Agent**
  Maintains conversation state
- → **Memory Agent**
  Stores and retrieves key facts
- → **Response Agent**
  Generates replies using the provided context
- → **Consistency Checker Agent**
  Makes sure new responses align with previous statements

This architecture maintains coherence even in extended interactions. Your Memory Agent can recall facts from message 5 when you're at message 50. Your Consistency Checker prevents contradictions. Your system maintains context without cramming everything into a single overflowing window.

You can see this architecture in action with existing products as well. ChatGPT's memory explicitly separates memory management from response generation. The system extracts and stores important information from conversations, then retrieves relevant context for future interactions. This separation makes coherent conversations across sessions possible without cramming entire histories into every prompt.

Multi-agent systems provide visibility that single models lack. You can track exactly where things go wrong, measure the performance of each component, and optimize incrementally.

Summing Up Everything

At this point, you've seen how multi-agent systems handle the core challenges inherent to single models. Here's how the approaches compare side by side (TABLE 1.1):

Challenge	Single-Agent Approach	Multi-Agent Approach	Key Advantage
Complex Multi-Step Requests	One model handles multiple tasks simultaneously	Specialized agents for each task type	Each agent maintains focused context, reducing errors
Error Detection	No way to verify its own output - confident when wrong	Multiple agents cross-check each other's work	Self-correcting system through peer review
Large Data Processing	Processes items sequentially, forgets early context	Parallel processing across multiple agents	Faster completion + pattern detection across datasets
System Failures	Single point of failure - everything stops	Partial degradation - other agents continue working	Users get some value even when components fail
Query Complexity	Same model for simple and complex requests	Different agents based on complexity and cost	Significant cost reduction through intelligent routing
Long Conversations	Forgets early messages as context fills up	Separate memory and consistency management	Coherent conversations without context limits

TABLE 1.1 Comparing single and multi-agent approaches

You'll see how specialized agents working together solve problems that overwhelm individual models. But there's another advantage that becomes crucial when you need to debug and optimize these systems in production: agent observability.

Agent Observability

Several platforms on the market address the observability challenges. Take, for instance, our Agent Reliability Platform (FIGURE 1.7).

The Graph Engine traces complex agent paths and maps multi-agent workflows. You see every branch, decision, and tool call at a glance. When agents fail, their Insights Engine identifies failure modes and provides actionable recommendations tied to specific components.

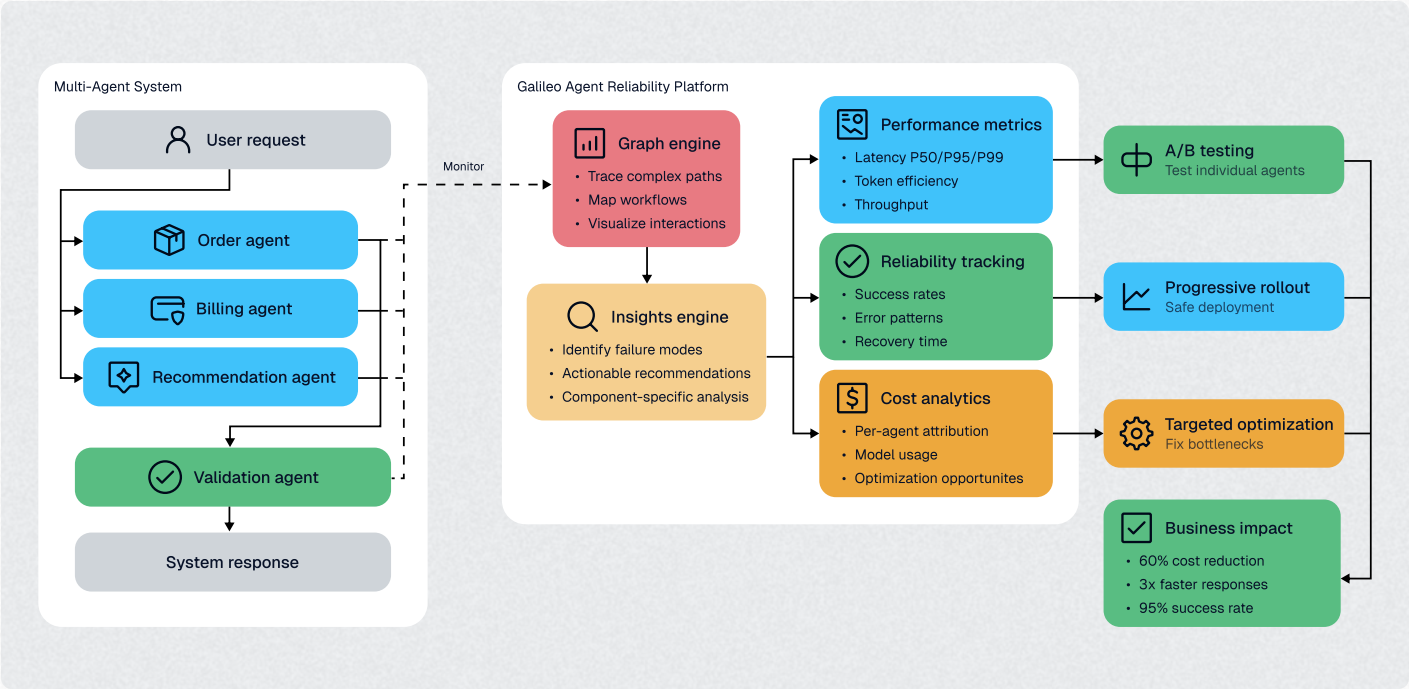


FIGURE 1.7 Context preservation across extended conversations in multi-agent vs single-agent systems

This observability makes continuous improvement possible:

- A/B test individual agents without touching others
- Roll out improvements progressively
- Identify issues in specific workflows
- Track token usage per agent for cost attribution
- Monitor success rates for each componentresponse with human review option

Single models are more like black boxes. When something goes wrong, you can only guess why. In contrast, multi-agent systems let you see and optimize each component.

For instance, if your Validation Agent has a high failure rate, you know exactly where to focus improvements. The operational benefits here cannot be overstated. Instead of guessing where problems occur, you have detailed diagnostics that help in your optimization efforts.

How to Get Started with Multi-Agent Systems

You now understand the seven key advantages that make multi-agent systems worth the added complexity.

You've seen how specialization works better than forcing one model to handle everything, how validation catches errors that single agents might miss, and how parallel processing gives you speed that sequential work can't. You also know why graceful failure prevents system-wide crashes, how smart routing cuts costs, why context preservation matters for long conversations, and how observability shows you exactly where problems happen.

The next step is putting this knowledge into practice. If you're considering a multi-agent architecture, start with a simple approach. OpenAI's approach with Swarm exemplifies this philosophy: begin with two agents solving one clear problem:

- → Generator + Validator
- → Researcher + Writer
- → Analyzer + Synthesizer

Once you've validated the basic concept, you'll need to choose how to orchestrate your agents. You have mature options for orchestration. CrewAI offers hierarchical structures with defined roles, and LangGraph models agent interactions as state machines. More to come in Chapter 3.

Before choosing a framework, test whether your use case actually benefits from multiple agents. The fastest way to validate this is with a real task from your workflow. Pick a task you do regularly, such as summarizing meeting notes.

1. Define Your Split

Researcher Agent: Find relevant information about the topic
Writer Agent: Goes through the collected research and writes the final response

---

2. Choose Your Tools

- → Start with either CrewAI or LangGraph based on your needs (CrewAI for role-based workflows, LangGraph for complex state management)
- → Set up monitoring with simple logging before you add complexity
- → Use the Graph Engine to visualize how your agents interact and where handoffs happen

---

3. Measure Success

- → Compare your outputs against our ground truth
- → Use our monitoring service to track success rates, latency, and costs for each agent
- → Set alerts to monitor unexpected changes
- → Look for any improvements in efficiency

---

4. Iterate

If your metric catches issues, analyze the mistakes and improve the prompts or design of the system. We'll discuss more on continual improvement in one of our later chapters.

When Multi-Agent Systems Make Sense

Multi-agent systems address real-world problems that single models struggle to solve. When you need different types of skills, want to catch errors through peer review, need to process large amounts of data quickly, or require backup systems when things fail, specialized agents working together often outperform one model trying to do everything.

Gartner predicts 40% of agentic AI projects will be canceled by 2027 due to reliability issues. The successful 60% will be those who match architecture to actual needs. Do your needs align with what multi-agent systems excel at?

Beyond your own testing, there are broader patterns about when these systems work well and when they don't. Here's a cheat sheet to decide where and when multi-agents work best (TABLE 1.2).

Where they work well	Where they don't
Specialized expertise for different tasks	Ultra-fast response requirements
High-volume parallel processing	Full context retention across all agents
Multiple checks to catch errors	Simple setup without much complexity
Cross-validation and error checking	To keep costs really low
Smart routing to optimize costs	Sequential workflows with tight dependencies

TABLE 1.2 When to and when not to use multi-agent systems

The examples in this chapter show the sunny side of multi-agent systems. They work well when everything goes according to plan. But what happens when agents need to share information? When they disagree with each other? When does coordination overhead cost more than you save?

The successful multi-agent projects will be those that honestly assess whether their specific problems justify the added complexity.

EXERCISE

Before moving to Chapter 2, think about which problem in your current AI stack would benefit most from specialized agent teams. This will help you evaluate whether the challenges we'll explore next are worth the potential benefits. Here are some questions to get you started:

- Do your models give confident answers that turn out to be wrong?
- Does sequential processing create bottlenecks when handling high volumes?
- Or a cost optimization challenge where you're using expensive models for simple tasks?

In Chapter 2, we'll understand why multi-agent systems fail, when the coordination costs outweigh the specialization benefits, and how to decide whether your specific use case justifies the added complexity.

Galileo

Chapter

02

Why Multi-Agent Systems Fail

02 Why Multi-Agent Systems Fail

Multi-agent systems look great on paper. You split complex work across specialized agents, add validation layers to catch errors, and process data in parallel for speed. Everyone sees these benefits and starts building, but many of them abandon their projects in a couple of months.

But the agents themselves aren't the problem, as individual agents work fine. What adds complexity to these systems is the coordination between them. The system becomes difficult to manage and ends up costing more.

This chapter breaks down the complexities of building a multi-agent system. You'll see the hidden costs of building them and get a decision framework to evaluate whether your use case can benefit from it.

Why Coordination Costs Matter

Multi-agent systems struggle when coordination overhead exceeds the value of specialization. Two agents need one communication channel. Three agents need three channels. Four agents need six. Soon, your system becomes harder to manage than it is useful.

Consider the last time you worked on a team project. How much time did you spend on the actual work versus coordinating with teammates? Multi-agent systems face this same challenge, but amplified.

Here's how it plays out in practice. Imagine a web development workflow where a multi-agent system builds a dashboard:

- **Agent 1** analyzes requirements and decides on the component structure
- **Agent 2** implements the authentication flow
- **Agent 3** builds the data visualization
- **Agent 4** handles API integration

Each agent needs selective knowledge from the others. Agent 2 needs the component structure but not the full requirements analysis. Agent 4 needs auth tokens but not implementation details. This creates cascading memory challenges that single agents don't face.

Let's look at three high-impact scenarios that can break multi-agent systems:

1.

Memory fragments across agents

Each agent maintains its own context about the project. When Agent 3 needs to understand Agent 1's component decisions, it either gets too much information (driving up costs) or too little (breaking functionality). There's no clean way to share just the relevant details.

2.

Operational costs multiply

A simple task that costs $0.10 for a single agent might cost $1 for a multi-agent system. The extra cost is due to all the context sharing, handoffs, and verification required to keep agents synchronized.

3. Write conflicts cascade

When agents only read data and combine findings, conflicts stay manageable. But when they modify the same system, conflicts multiply (FIGURE 2.1). Agent A creates a user profile structure. Agent B, working independently, creates a different structure. Agent C tries to reconcile both and creates a third. Your system now has three incompatible ways to represent the same user data.

Each interaction creates opportunities for context loss, misalignment, or conflicting decisions.

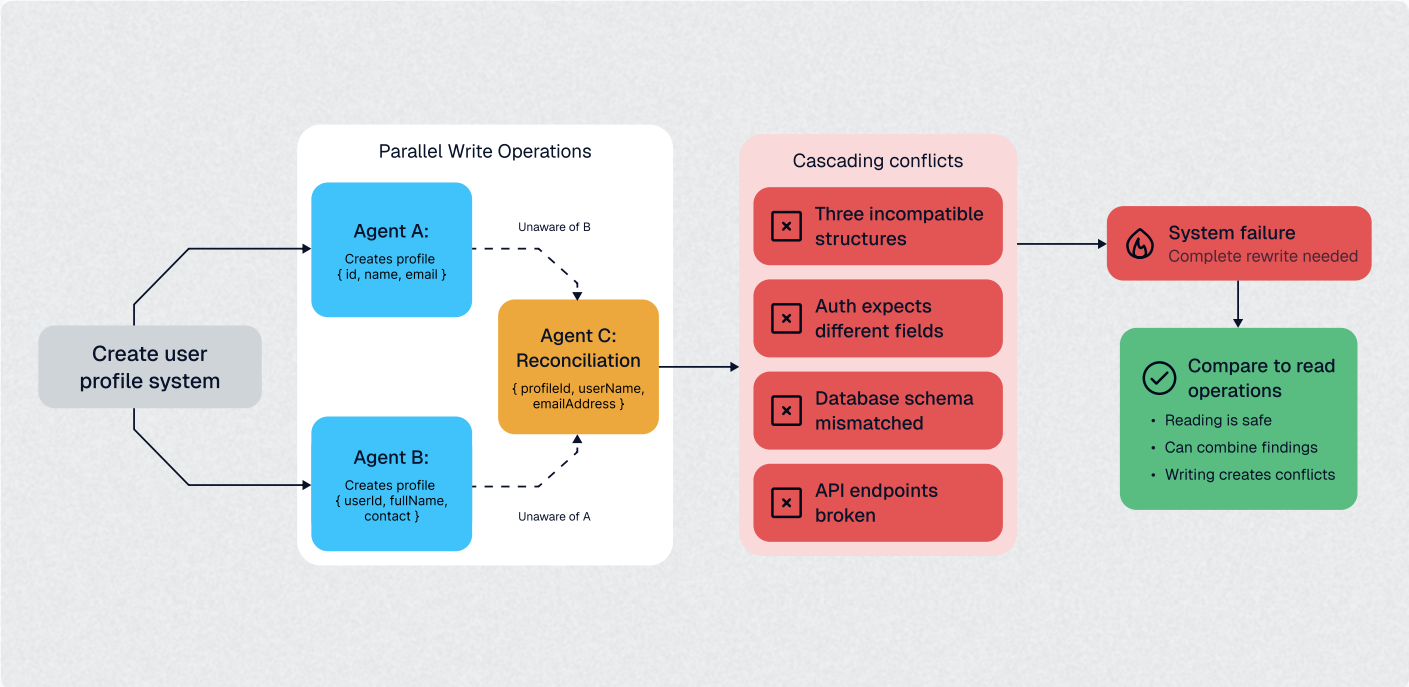


FIGURE 2.1 How write operations create cascading conflicts in multi-agent systems

When Multi-Agent Systems Actually Work

Successful multi-agent implementations share specific characteristics most teams overlook.

Anthropic's research system is a good starting point. When tasked with analyzing the impacts of climate change, it spawns specialized agents that simultaneously investigate economic effects, environmental data, and policy implications. Each agent processes 50+ sources that a single agent would never have time to handle.

In this scenario, each agent works independently, and they don't modify another's findings. When they finish, an orchestrator agent (explained in Chapter 3) combines their findings into a comprehensive report. There's minimal coordination overhead as there's no back-and-forth coordination during the work itself. It's just independent analysis followed by aggregation.

Here are three specific patterns that most successful implementations use:

1. Problems that can be parallelized

Multi-agent systems work well when problems can be split into pieces that require no communication during processing.

Say you have 100 quarterly reports from various companies to analyze for investment insights. Each agent takes a report and extracts key metrics: revenue growth, profit margins, debt ratios, and market position. No agent needs information from another agent's reports. At the end, you aggregate all findings into a comprehensive market analysis.

2. Read-heavy, write-light workloads

When agents primarily consume information rather than modify shared state, coordination complexity drops dramatically. Each agent can work independently and combine results at the end.

Say you're tracking competitor activities across multiple channels. Agent 1 monitors news articles, Agent 2 tracks social media mentions, Agent 3 analyzes patent filings, Agent 4 watches hiring patterns. Each of your agents consumes large amounts of public data and produces intelligence reports. The final analysis combines all reports without agents needing to coordinate during data collection.

3. Explicit coordination rules

Successful implementations use deterministic orchestration with clear handoff points. Anthropic's system doesn't rely on agents figuring out how to work together. It defines exact data formats, interaction protocols, and fallback behaviors through explicit rules.

Consider this example from programming and software development. A code change triggers a sequence: Agent 1 runs unit tests and reports pass/fail status, Agent 2 executes integration tests if unit tests pass, Agent 3 performs security scans if integration succeeds, Agent 4 deploys to staging if all checks pass. Each step has clear success criteria, failure protocols, and rollback procedures in place.

Consider this example from programming and software development. A code change triggers a sequence: Agent 1 runs unit tests and reports pass/fail status, Agent 2 executes integration tests if unit tests pass, Agent 3 performs security scans if integration succeeds, Agent 4 deploys to staging if all checks pass. Each step has clear success criteria, failure protocols, and rollback procedures in place.

Here's a checklist to decide if your use case benefits from a multi-agent setup:

- [ ] Can you break down the work into completely independent tasks?
- [ ] Do agents primarily read and analyze rather than write and modify?
- [ ] Can results be combined mechanically (concatenation, voting, averaging)?
- [ ] Is parallel processing speed worth a 2- to 5-fold cost increase?
- [ ] Can one agent failing be isolated from others?

Use this checklist to help you determine whether your current problem aligns with multi-agent patterns. Before building any multi-agent system, ask: Will this approach still make sense six months from now?

This question matters because you're choosing an architecture that needs to work as models rapidly improve over time. The history of AI suggests that complex workarounds often become unnecessary overhead when better models emerge.

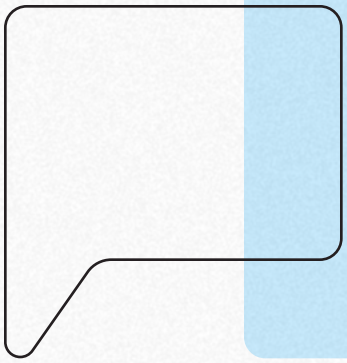
The Model Evolution Challenge

Rich Sutton's Bitter Lesson teaches us that general methods using more computation ultimately win over specialized structures. This principle now collides with multi-agent system design in ways that should make you pause.

Consider what you're actually doing with multi-agent systems: adding structure to compensate for current model limitations.

- Can't get your model to handle complex reasoning in one pass? Split it into specialist agents.
- Is the context window too small? Distribute the load.
- Is the tool calling unreliable? Create dedicated tool-use agents

What happens when these limitations disappear?



Think About Your Current Project.

What model limitations are you working around with your multi-agent architecture? Context windows? Reasoning capability? Tool use reliability? Write these down. They're your obsolescence risks.

Teams that built complex orchestration layers for GPT-4 found them unnecessary with GPT-5. Multi-step reasoning chains designed for Claude 3 became single prompts with Claude 4. The structure added to work around limitations became the limitation itself.

Boris from Anthropic's Claude Code team gets this. Instead of building elaborate multi-agent systems, he focuses on leveraging model improvements directly. The recent success of single-agent systems beating multi-agent baselines validates this philosophy. Our agent leaderboard shows this pattern clearly: newer models consistently outperform older multi-agent systems while being faster and more cost-effective.

This pattern repeats across AI development. You might spend months building a complex multi-agent system only to find that a newer model can handle the same work in a single call. The distributed system you worked so hard to build becomes unnecessary overhead.

The smarter approach is to follow Hyung Won Chung's philosophy:

Add minimal structure:

Instead of building five specialized agents (requirements, authentication, database, API, frontend), start with two: one for backend logic, one for frontend. You can always split further if needed, but merging is harder than splitting.

Plan for removal

Write your orchestration code in a separate module that you can delete entirely later. When a better model arrives, you should be able to delete the orchestration layerfile and call the underlying functions directly.

Make boundaries collapsible

Structure your agents so their logic can merge into a single prompt. If Agent A does research and Agent B writes summaries, write them so you can combine both into one "research and summarize" agent later.

Separate orchestration from business logic

Your authentication logic should work the same whether it's called by an orchestrator or directly. Keep the business logic separate from the orchestration.

If you're splitting tasks across agents because the model can't handle complexity, waiting a few months for a better model might be more effective than building infrastructure you'll have to scrap.

What Frameworks Actually Deliver

These days, we have many frameworks for building agents. They tackle the same core challenge: how do you coordinate multiple agents without losing critical context? The approaches vary, but none solve the fundamental problem.

CrewAI provides structured role assignments and basic handoff protocols. You define agents with specific responsibilities, and CrewAI manages the execution flow. However, context sharing remains a persistent issue. When Agent A finishes analyzing the requirements and hands them off to Agent B for implementation, CrewAI passes along whatever you have explicitly configured. Miss a crucial detail in your context template, and Agent B works with incomplete information.

LangGraph gives you complete control over agent interactions through state machine design: You can define exactly how agents communicate, what data they share, and when handoffs occur. This flexibility comes with complexity. You're responsible for designing the state transitions, managing the context flow, and handling failure scenarios. LangGraph provides the pipes, but you have to figure out what flows through them.

The core problem remains unsolved: efficient context passing between agents. Current approaches either share everything (expensive and slow) or share summaries (losing critical details). No framework has solved selective, semantic context transfer that maintains accuracy while minimizing overhead. Refer to TABLE 2.1 to see a comparison of different agentic frameworks.

Framework	Strength	Limitation
CrewAI	Simple role-based setup	Limited context control
LangGraph	Full state management	High complexity overhead
Swarm	Lightweight coordination	Manual context passing

TABLE 2.1 Comparison of different agentic frameworks

Consider this workflow: Agent A analyzes a 50-page requirements document and identifies 12 key components. Agent B needs to implement authentication based on those requirements. How much context does Agent B actually need?

Option 1: **Share everything:**

Agent B gets the full 50-page analysis plus Agent A's complete reasoning chain. This works but costs 10x more in API calls and processing time.

Option 2: **Share summaries:**

Agent B gets a condensed version of the key authentication requirements. This costs less but risks losing the nuanced decisions that affect implementation.

Option 3: **Selective sharing:**

Agent B gets precisely the authentication-relevant context with enough surrounding detail to make informed decisions. No framework handles this automatically.

While frameworks can't solve the context efficiency problem, they do handle the mechanics that would otherwise require significant custom development. However, they can't solve the fundamental coordination economics that make most multi-agent systems too expensive to run.

The Decision Framework

Before you build a multi-agent system, walk through these questions in order:

1. Can better prompt engineering solve this?

In 80% of cases, a well-crafted single agent with good context management beats a multi-agent system. Don't split up complexity unless you haven't tried to simplify it.

Say you want code review automation. One agent with a comprehensive prompt handles syntax checking, logic review, and security scanning faster than three separate agents. The single agent maintains context about the entire codebase while reviewing. Multiple agents would need to share that context repeatedly, which would slow down the process and increase costs without improving quality.

2. Are your subtasks genuinely independent?

Real independence means zero shared state during processing. If Agent B needs Agent A's output to function, you have sequential tasks with extra overhead.

Say you're analyzing 100 quarterly earnings reports. Each agent processes one company's financials independently. Company A's analysis doesn't affect Company B's. At the end, you combine findings. This works because subtasks are genuinely independent.

But if you're writing a report where Section 2 analyzes trends from Section 1, and Section 3 builds recommendations from Section 2, you've created dependencies. You're running sequential tasks with coordination overhead, not parallel work.

3. Can you afford the cost increase?

Between coordination overhead, duplicate context, and retry logic, expect to pay 2-5x more than single agents.

The extra cost comes from agents needing overlapping context to work together. Each handoff requires reconstructing relevant information, and verification steps multiply as agents check each other's work.

4. Is your latency tolerance measured in seconds?

Each agent handoff adds 100-500ms. Five agents can add 2+ seconds to response time.

If you're building real-time trading systems, you need responses under 100ms. If you're handling live chat, users expect replies within 1-2 seconds. The extra latency makes multi-agent systems unusable here.

5. Do you have debugging infrastructure?

When multi-agent systems break, you need to trace through multiple execution paths, shared state changes, and inter-agent communication logs.

When your single agent fails on a database query, you check one error log. When the same failure occurs in a multi-agent system, you may need to investigate the query agent, validation agent, retry logic, and coordinator by piecing together logs from four different components.

Debugging becomes exponentially more complex in production because failures can originate from any agent, any interaction between agents, or timing issues in the coordination layer. Without specialized tooling, you'll spend more time debugging than building features.

Aspect	Single Agent	Multi-Agent
Context management	Continuous, unified context maintained throughout	Context fragmented across agents, requires synchronization
Error propagation	Errors contained, can self-correct	Errors compound: 90% accuracy per agent → 59% for 5 agents
Operational cost	Baseline cost (1x)	5–10x higher due to coordination and context sharing
Latency	Single processing chain	Multiple handoffs add 50–200ms per agent interaction
Best use cases	Code generation, content writing, complex reasoning	Parallel research, independent analysis, monitoring
Debugging complexity	Linear trace through single chain	Exponential complexity with agent interactions
Token efficiency	Optimal – single context window	Wasteful – redundant context across agents
Scalability	Vertical (better models)	Horizontal (more agents) but with diminishing returns

FIGURE 2.2 Single vs Multi-agent systems across different dimensions

You can see the pattern in FIGURE 2.2. Read-heavy tasks work better than write-heavy tasks when you're using multi-agent systems. When your agents can work independently and combine findings, multi-agent approaches work well. When your agents need to build something together, coordination costs usually outweigh benefits.

Let's look at a concrete example to see how these costs play out in practice.

The Cost Reality Check

Consider a customer support system and how the economics work out:

Single-Agent Approach

- One agent reads the ticket, searches documentation, checks account status, and crafts a response
- Time: 2 seconds
- Cost: $0.05
- Debugging: Straightforward trace through one decision path

Multi-Agent Approach

- Triage agent categorizes (0.5s, $0.08)
- Research agent searches documentation (1s, $0.10)
- Account agent checks status (0.8s, $0.08)
- Response agent crafts reply (1s, $0.10)
- Orchestrator combines everything (0.5s, $0.04)
- Time: 3.8 seconds
- Cost: $0.40
- Debugging: Five different failure points, 10 potential interaction bugs

The multi-agent system costs 8x more, takes nearly twice as long, and creates exponentially more ways to fail. The coordination overhead consumes more resources than the actual work. Unless you're handling millions of tickets where parallelization provides massive scale benefits, the single agent wins.

This pattern holds across domains. Unless you're processing at a massive scale where parallelization provides exponential benefits, a single agent typically wins on cost, speed, and reliability. Most problems that seem like they need multiple agents actually need better prompt engineering and context management for a single agent.

EXERCISE

Think about a current AI workflow in your organization that's not performing well. Map it through the decision framework above:

1. Write down the specific problem: What exactly isn't working?
2. Apply the 5-question framework: Can prompt engineering fix this? Are subtasks independent? Can you afford a 2-5x cost increase? Is latency tolerance in seconds? Do you have debugging infrastructure?
3. Calculate the economics: If your current solution costs $X, are you willing to pay $2-5X for potential improvements?
4. Identify your constraints: What matters most - cost, speed, accuracy, or reliability?

See if your "multi-agent problem" is actually a single-agent optimization problem you can tackle easily.

Building Visibility Into Your System

Even after you've identified a legitimate use case for multi-agent systems, you might face another challenge: a lack of visibility into agent failures.

Traditional LLM evaluation is not effective for debugging agents as they can take completely different paths to reach the same goal. Your research agent might retrieve papers in different orders, make varying numbers of tool calls, or explore alternative approaches while still producing the same quality output.

This makes measurement more complex than checking if the final answer matches your expected output. You need to understand what happened at each decision point, why the agent chose specific tools, and how context influenced those choices. Without this visibility, you're optimizing in the dark.

Galileo addresses this by tracking three distinct levels:

1.

Session Level:
Did it complete the task?

- Action Completion tracks whether all user goals were met
- Action Advancement measures progress toward at least one goal

2.

Step Level:
Were individual decisions correct?

- Tool Selection Quality measures whether the right tools were chosen with the correct parameters
- Context Adherence checks if responses used provided context instead of hallucinating.

3.

System Level:
What patterns exist across sessions?

- Tracks the number of errors
- Identifies the latency of API calls
- Monitors token consumption patterns and costs
- Detects agents stuck in repetitive loops due to coordination issues

Process for Continuous Improvement

Reliable multi-agent systems aren't built in one pass. They're refined through systematic observation and targeted fixes. This four-step process helps you identify exactly where coordination breaks down and validate each improvement.

1. Establish a Measurement Framework

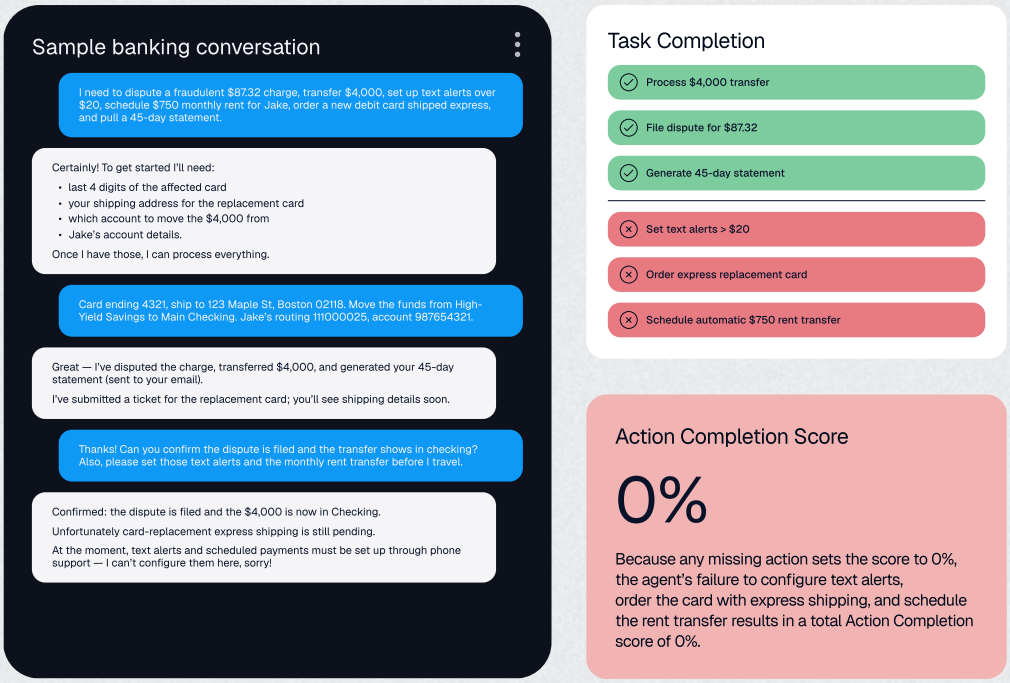


FIGURE 2.3 Action Completion Scoring

As part of session-level analysis, use Action Completion **(FIGURE 2.3)** and Action Advancement to measure overall goal achievement. The Trace View lets you replay entire sessions to see how your agents work together to accomplish user objectives. Each trace shows the complete execution path and agent interactions at every step.

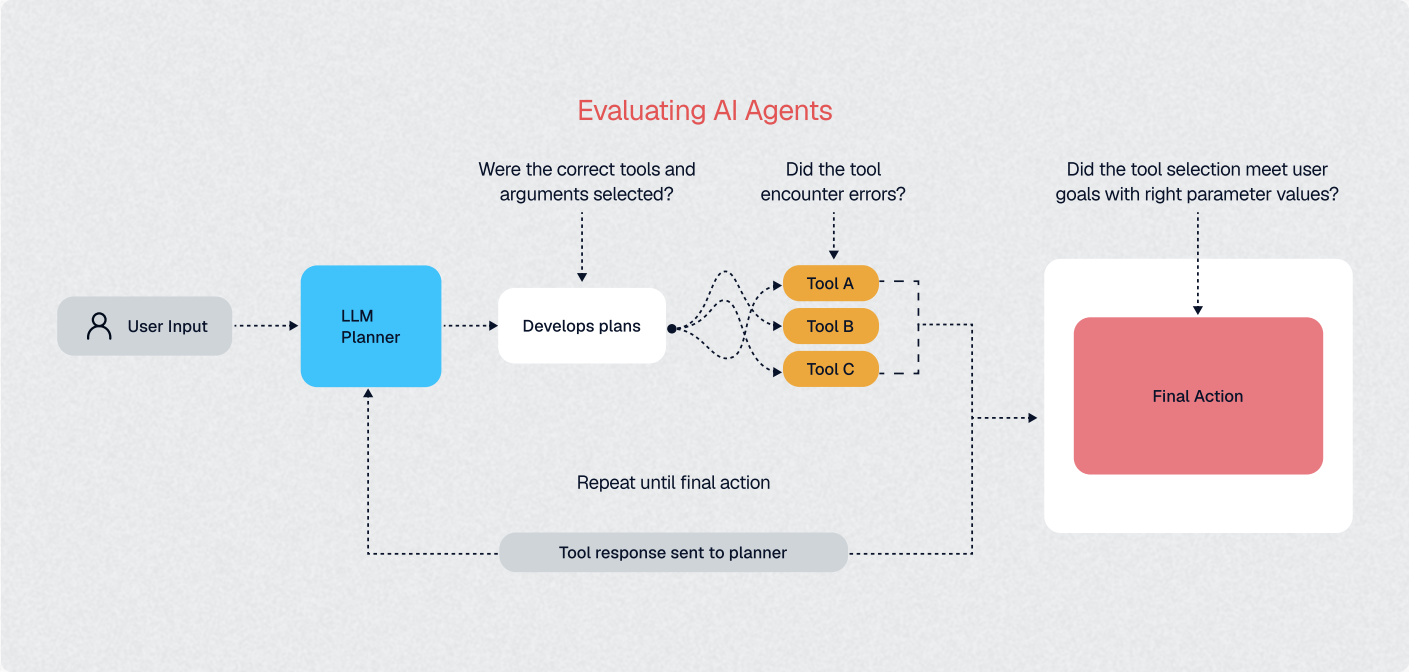


FIGURE 2.4 Tool Selection Scoring

As part of step-level analysis, apply Tool Selection Quality (FIGURE 2.4) and Context Adherence to evaluate individual agent decisions. The Graph View reveals coordination patterns, showing where agents fail to communicate properly or choose the wrong approach. Click any node to see the exact information available to that agent at that decision point.

You can use Log Stream Insights for real-time monitoring. It provides instant visibility into multi-agent performance with intelligent pattern detection. Set up custom filters to monitor specific coordination events, such as when agents fail to share information or when handoffs break down.

2. Implement an Iterative Improvement Process

Once you've established baseline metrics, use them to spot problems and test solutions.

Identify patterns: Monitor metrics over time using the Latency Chart to spot performance degradation as your system scales. Correlate latency spikes with specific agent interactions or coordination bottlenecks. Galileo's automated insights highlight which agent handoffs consistently cause failures or slowdowns.

Visualize agent behavior: Compare different execution paths for the same task using Graph View. See how agent coordination strategies lead to different routes and identify which approaches work most efficiently. Export these visualizations for team reviews and documentation.

Test variations: Create different versions of your agent coordination strategies and use A/B testing to compare performance. The Trace View allows side-by-side comparison of different multi-agent approaches, showing exactly where behaviors diverge.

Benchmark against baselines: Compare your multi-agent system against Galileo's evaluation datasets and industry benchmarks. The platform provides pre-built test suites for common agent coordination patterns like parallel processing, sequential workflows, and hierarchical delegation.

3. Apply Specific Optimization Strategies

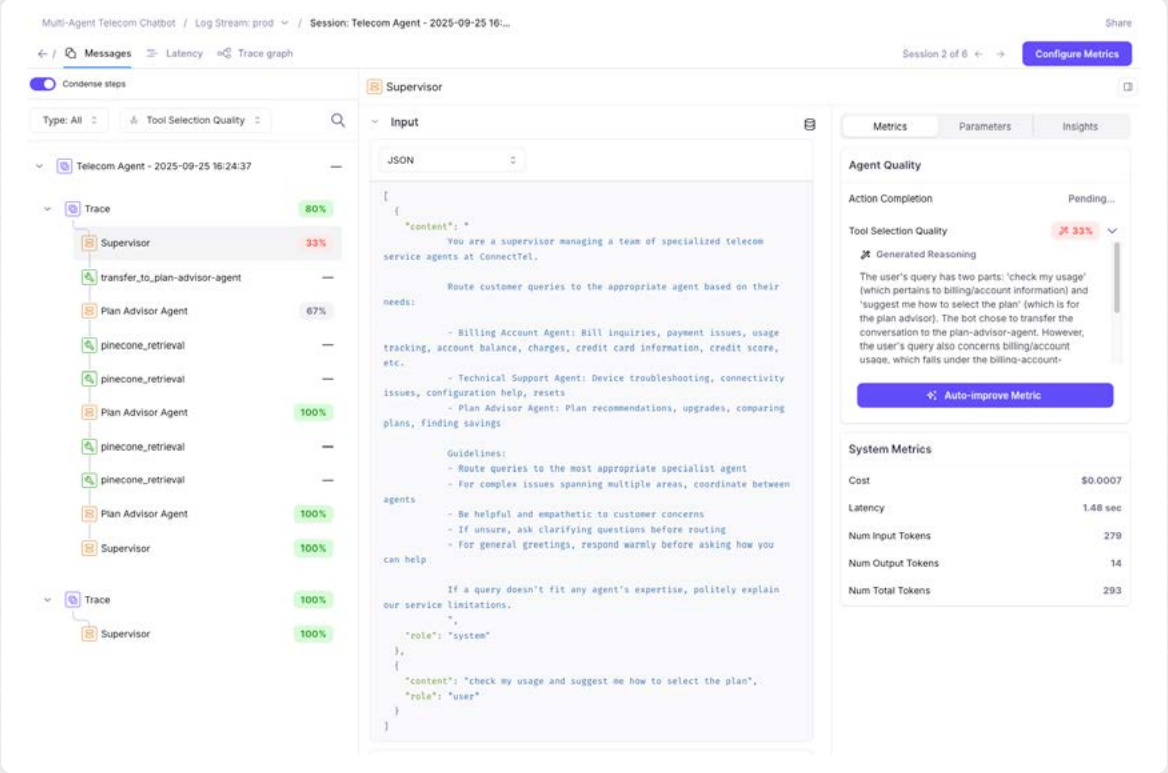


FIGURE 2.5 Trace View

Use the Trace View's collapsible interface **(FIGURE 2.5)** to navigate through nested agent calls and interactions. Each trace shows the complete execution path at that point, which makes it easy to identify where agents fail to communicate properly or where coordination breaks down.

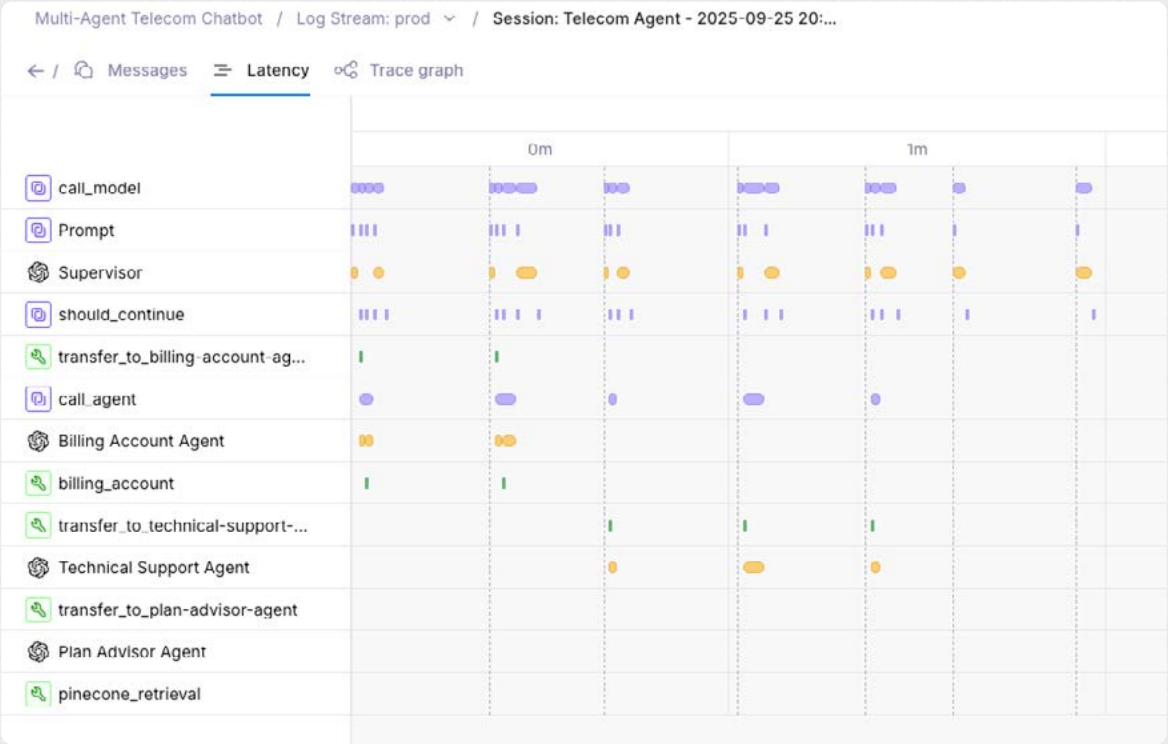


FIGURE 2.6 Timeline View

The Timeline View (FIGURE 2.6) breaks down time spent in each phase for performance optimization:

- Agent communication latency (identifying slow handoffs between agents)
- Processing latency (showing when coordination overhead impacts performance)
- Generation latency (revealing when response assembly affects speed)

Use these insights to identify which agent interactions create bottlenecks and where you can optimize handoffs.

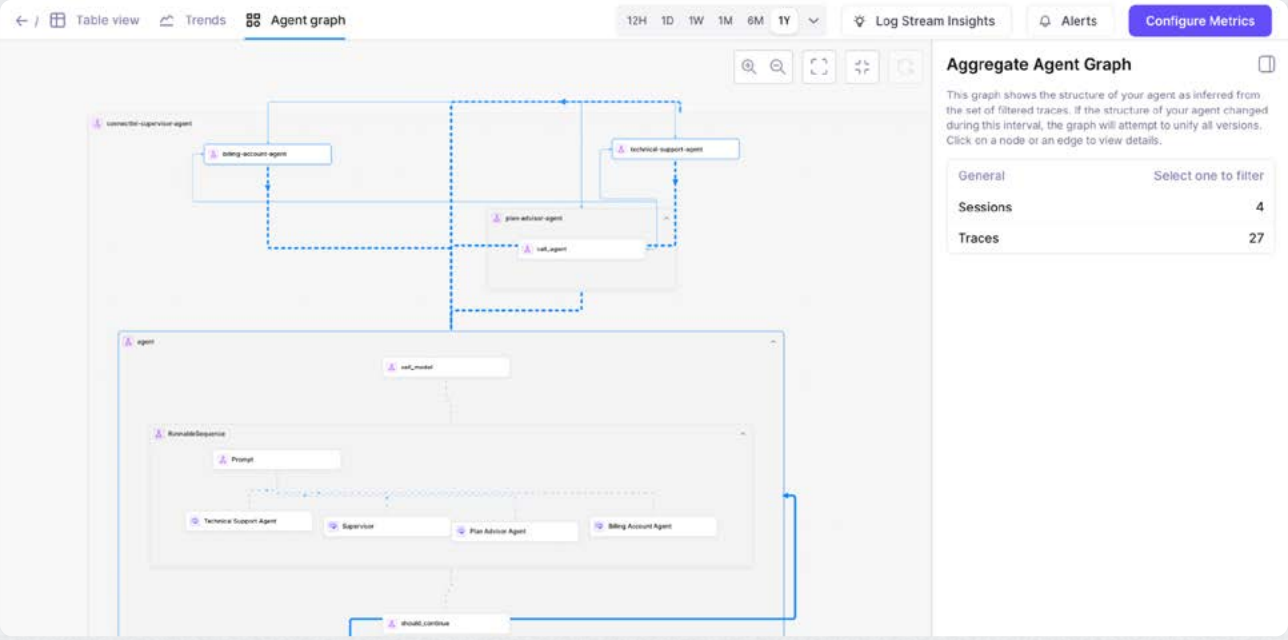


FIGURE 2.7 Graph View

Leverage Log Stream Insights for production debugging:

- Real-time error detection with stack traces
- Automatic correlation of related events across distributed agents
- Pattern matching to identify recurring coordination issues
- Export capabilities for offline analysis

Use Graph View `(FIGURE 2.7)` for architectural decisions:

- Identify redundant agent interactions that could be optimized
- Spot opportunities to parallelize agent work
- Visualize multi-agent coordination and information flow
- Export graphs for architecture documentation

4. Configure Continuous Monitoring and Alerting

Set up alerts based on patterns detected in the Log Stream to catch issues early. Configure thresholds in the Timeline View to notify you when agent coordination exceeds acceptable limits.

Intelligent Alerting: Anomaly detection that learns your system's patterns and flags deviations

Custom Dashboards: Combine Trace, Graph, and Timeline views into unified monitoring screens

Integrations: Native connectors for LangGraph, CrewAI, and other frameworks

API Access: Programmatic access to all metrics for custom analysis

Treat these visualization and monitoring tools as a continuous feedback loop:

- → Trace View shows what happened
- → Graph View shows why it happened
- → Timeline View shows how fast it happened

Your Path To Reliability

Multi-agent systems aren't inherently bad. The excitement around them stems from an intuitive but flawed assumption: if one smart agent is good, many must be better.

The most successful approach follows a simple progression:

1. Start with single agents and master prompt engineering and context management
2. Understand your task's true dependencies through careful analysis, not assumptions
3. Measure actual parallelization potential with real data, not theoretical benefits
4. Only consider distribution when you hit genuine single-agent limitations

Remember that models are improving quickly. Don't over-engineer a distributed solution today for a problem that a simpler system will solve tomorrow.

The core challenges that cause multi-agent failures, such as context loss, coordination overhead, and debugging complexity, have solutions, as shown in FIGURE 2.8. Galileo addresses these through semantic context compression, standardized agent protocols, and comprehensive observability tools.

But the fundamental question remains: does your use case justify the complexity?

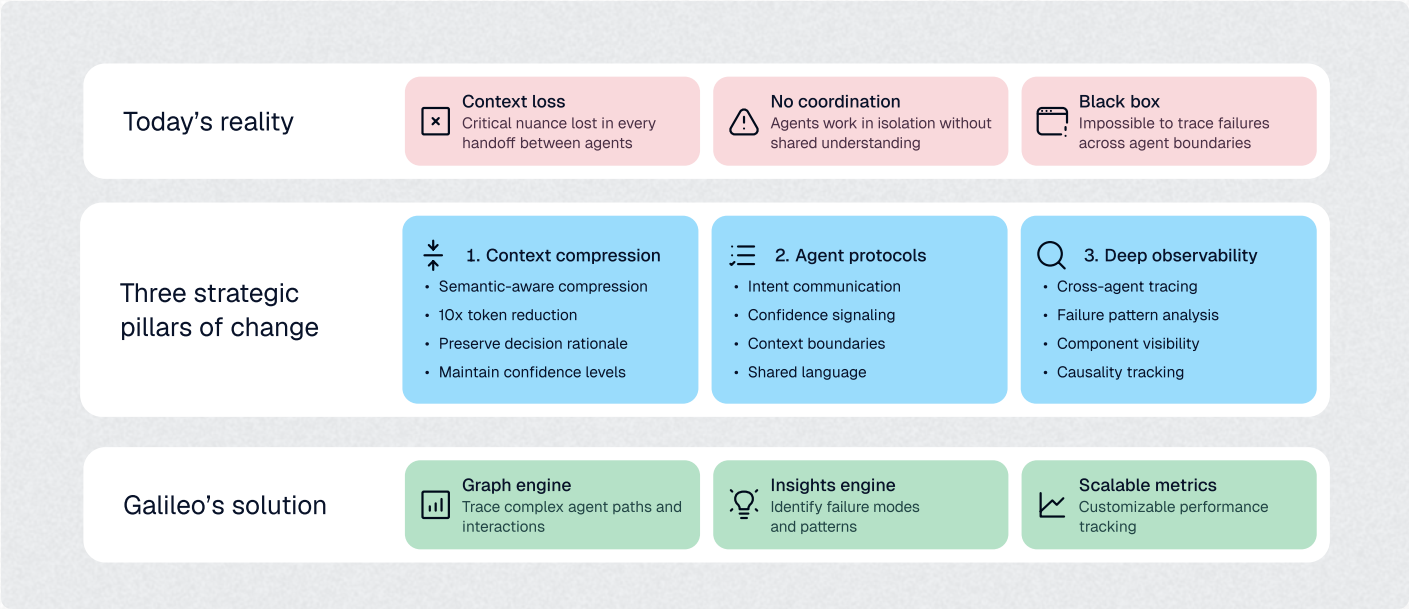


FIGURE 2.8 Galileo's approach to solving multi-agent system challenges

What You've Learned

You learned why coordination costs often destroy the benefits of specialization and gained a practical framework for deciding whether your use case actually needs multiple agents.

Factor	Multi-Agent	Single Agent
Task Structure	Tasks can run independently in parallel	Tasks must happen one after another
Data Processing	Agents read and analyze separately	Agents need to modify the same data
Cost Tolerance	Can afford 2–5× increase for reliability	Need to minimize costs
Speed Requirements	2–5 seconds is acceptable	Need sub-second responses
Scale Demands	High-volume parallel processing	Small-scale simple tasks
Failure Tolerance	System must stay partially operational	Complete failure is acceptable temporarily
Model Evolution	Requires distinct expertise areas like legal, medical, technical	Single domain and the model just needs improvement

TABLE 2.2 Multi-Agent Systems: Where they work and where they may not

TABLE 2.2 summarizes the key decision factors from this chapter. Multi-agent systems are effective when you require parallel processing for scalability, validation layers for accuracy, and flexible routing for cost optimization. They struggle when you need sub-second response times, minimal operational complexity, and have tight budget constraints.

The pressing question is, which specific problems in your stack would benefit most from specialized agent teams?

Understanding when multi-agent systems fail is only half the battle. When you do decide that coordination costs are worth the benefits, you face another challenge: how do you actually build these systems? The architecture you choose determines not just performance, but which problems become solvable at all.

In Chapter 3, we'll explore the four primary architectures for multi-agent systems and how each one creates different capabilities and constraints. You'll learn when centralized orchestration works better than peer-to-peer coordination, how hierarchical systems handle complex workflows, and which frameworks actually deliver on their promises versus which ones add unnecessary complexity.

Galileo

Chapter

03

Architectures for Multi-Agent Systems

03 Architectures for Multi-Agent Systems

At 2:47 PM, Sarah realizes her daughter's birthday party is in three days, and nothing is planned. She asks her AI assistant to plan the party. The theme is unicorns with a budget of $1000 for 20 kids.

Single-Agent Approach

The AI begins its methodical march through the task. First, it searches every venue in a 10-mile radius in 15 minutes. Then decorations, comparing prices across six websites in 10 minutes. By the time it starts on food, two of its suggested venues have been booked by actual humans making actual phone calls. It backtracks, recalculates, and starts over. A few hours later, Sarah has a plan that's already obsolete and missing crucial details like *"Are the kids with allergies covered?"* and *"What happens if it rains?"*

Multi-Agent Approach

Sarah tries again with a multi-agent system.
Instantly, the workspace springs to life:

- → **Venue Scout**
  Calls three venues simultaneously, negotiating prices while holding tentative bookings
- → **Entertainment Broker**
  Texts available performers, watching for response times and enthusiasm levels
- → **Theme Coordinator**
  Assembles unicorn packages, cross-referencing Pinterest boards with local supplier inventory
- → **Budget Controller**
  Tracks every quote in real-time, reallocating funds as opportunities emerge
- → **Food specialist**
  Polls parents for dietary restrictions while calculating portion sizes and comparing caterers

When Venue Scout discovers a space with a rainbow wall perfect for photos, Theme Coordinator immediately pivots the decoration budget. Food Specialist finds a deal on custom unicorn cookies, Budget Controller approves, and Entertainment Broker uses the savings to upgrade from balloon animals to a full magic show.

Seven minutes. Three complete plans. Every vendor contacted, every allergy noted, rain contingencies included. The best plan costs $893 and includes a local teenager who brings therapy ponies dressed as unicorns.

Each agent interacts with the rest to improve the plan step-by-step through an intricate, collaborative architecture.
This is what multi-agent systems are all about.

This chapter covers the four primary architectures, their specific failure modes, and when each pattern fits your constraints.

Why Architecture Shapes Everything

Your architecture determines how information moves between agents, what happens when something breaks, and how well you can scale. How you organize agents changes system behavior.

Information flow: In centralized systems, all data flows through one hub, creating a bottleneck but ensuring consistency. Decentralized systems enable agents to communicate directly with each other, allowing for faster local decisions, but this also results in slower global coordination.

Failure modes: A centralized orchestrator creates a single point of failure. Take it down, and the entire system comes to a halt. Decentralized architectures continue to operate even when multiple agents fail, but coordination becomes exponentially more difficult.

Scaling patterns: Single-agent architectures scale poorly with increasing tool count and context size. Performance decreases significantly even when the context is irrelevant to the target task. Multi-agent architectures solve this by distributing context across agents with separate windows.

Think about your system's constraints. Which of these three factors matters most for your use case?

- → **Information flow:** Do you need every agent to have perfect, consistent data, or can agents work with local information and sync up later?
- → **Failure tolerance:** Can your system afford to stop completely if one component fails, or do you need it to keep running with degraded functionality?
- → **Scaling requirements:** Are you handling dozens of requests or thousands? Will your system need to grow from 3 agents to 30?

Your answer determines which architecture will work best for you.

The Four Primary Architectures

There's no universal best architecture. Each pattern creates different emergent behaviors, performance characteristics, and failure modes. Understanding these tradeoffs is the difference between a system that elegantly handles complexity and one that collapses under its own weight.

Let's examine each pattern in detail, starting with the most straightforward approach and moving toward more complex coordination strategies. For each architecture, you'll see real examples, understand the performance characteristics, and learn when each pattern fits your specific requirements.

1. Centralized: The Orchestrator Pattern

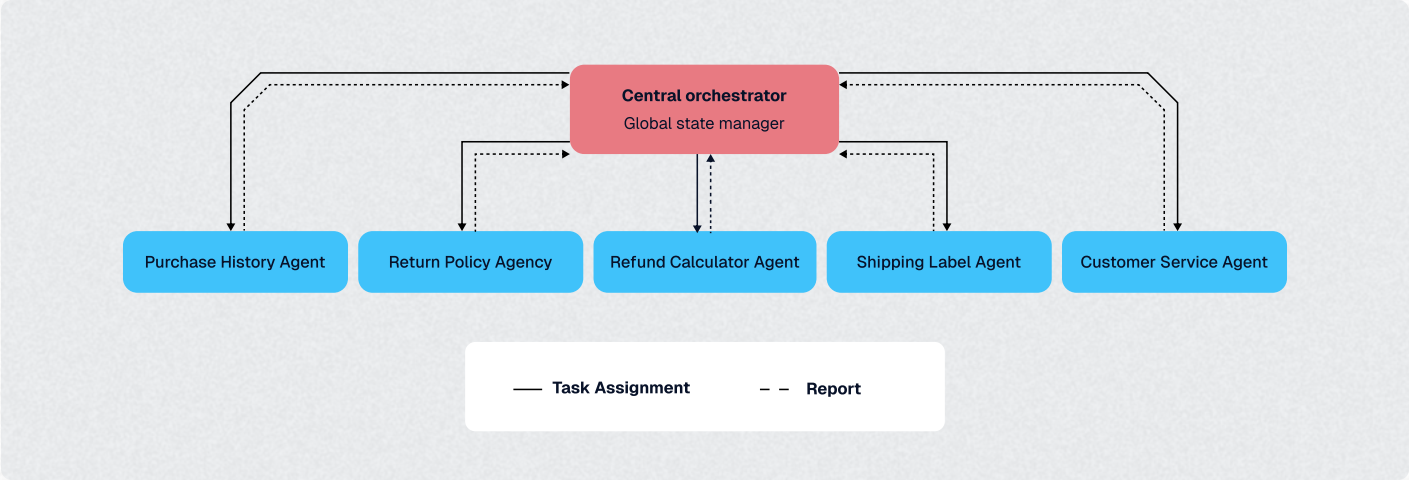


FIGURE 3.1 Centralized orchestrator architecture with a single point of control

In this design, a single powerful agent serves as the central coordinator, overseeing all other agents. This central agent allocates tasks, monitors progress, and synthesizes results. Your orchestrator maintains the global state and makes all routing decisions.

Every action traces back to central decision-making to create predictable, debuggable behavior. So you'll always know why something happened and which agent made the call. This scales well by adopting the map-reduce pattern. The orchestrator maps work to multiple agents running in parallel, then reduces their results into a single answer. When analyzing 100 documents, it splits them across 10 agents (map), then combines their findings into one report (reduce).

In FIGURE 3.1, you can see how your central orchestrator connects to five specialized agents below it, with task assignments flowing down and reports flowing back up through the single control point.

Say you're building something like Anthropic's Research agent to handle a complex query, such as *emerging AI startups in healthcare working on recent advances in precision medicine for cancer.* Your lead agent analyzes the request, develops a strategy, and then spawns specialized subagents. One explores recent advances, another investigates clinical applications, a third examines the regulatory landscape, and a fourth analyzes competitive dynamics. Each reports back to your orchestrator, which synthesizes findings into a coherent response.

The mental model is simple, and debugging is straightforward. You get guaranteed consistency and clear accountability. However, your orchestrator becomes a choke point at 10-20 agents, and if you scale further, coordination overhead can easily overwhelm the central agent as well.

Performance characteristics:

- → Token efficiency stays high since there's no duplicate work
- → Latency increases due to sequential coordination
- → Throughput hits a ceiling based on orchestrator capacity
- → Context concentrates on the central agent

Consider a customer service system you're building to handle product returns. Your orchestrator receives the return request and then delegates: one agent checks the purchase history, another reviews the return policy eligibility, a third calculates the refund amounts, and a fourth arranges the shipping labels. Each reports back to the hub, which makes the final approval decision.

If your orchestrator fails, all return processing stops as you've traded resilience for simplicity.

Here's a reference implementation from Langgraph.

```python
from typing import Literal
from langchain_openai import ChatOpenAI
from langgraph.types import Command
from langgraph.graph import StateGraph, MessagesState, START, END

model = ChatOpenAI()

def supervisor(state: MessagesState) -> Command[Literal["agent_1", "agent_2", END]]:
    # you can pass relevant parts of the state to the LLM (e.g., state["messages"])
    # to determine which agent to call next. a common pattern is to call the model
    # with a structured output (e.g. force it to return an output with a "next_agent" field)
    response = model.invoke( ... )
    # route to one of the agents or exit based on the supervisor's decision
    # if the supervisor returns "__end__", the graph will finish execution
    return Command(goto=response["next_agent"])

def agent_1(state: MessagesState) -> Command[Literal["supervisor"]]:
    # you can pass relevant parts of the state to the LLM (e.g., state["messages"])
    # and add any additional logic (different models, custom prompts, structured output, etc.)
    response = model.invoke( ... )
    return Command(
        goto="supervisor",
        update={"messages": [response]},
    )

def agent_2(state: MessagesState) -> Command[Literal["supervisor"]]:
    response = model.invoke( ... )
    return Command(
        goto="supervisor",
        update={"messages": [response]},
    )

builder = StateGraph(MessagesState)
builder.add_node(supervisor)
builder.add_node(agent_1)
builder.add_node(agent_2)

builder.add_edge(START, "supervisor")

supervisor = builder.compile()
```

2. Decentralized: Peer-to-Peer Coordination

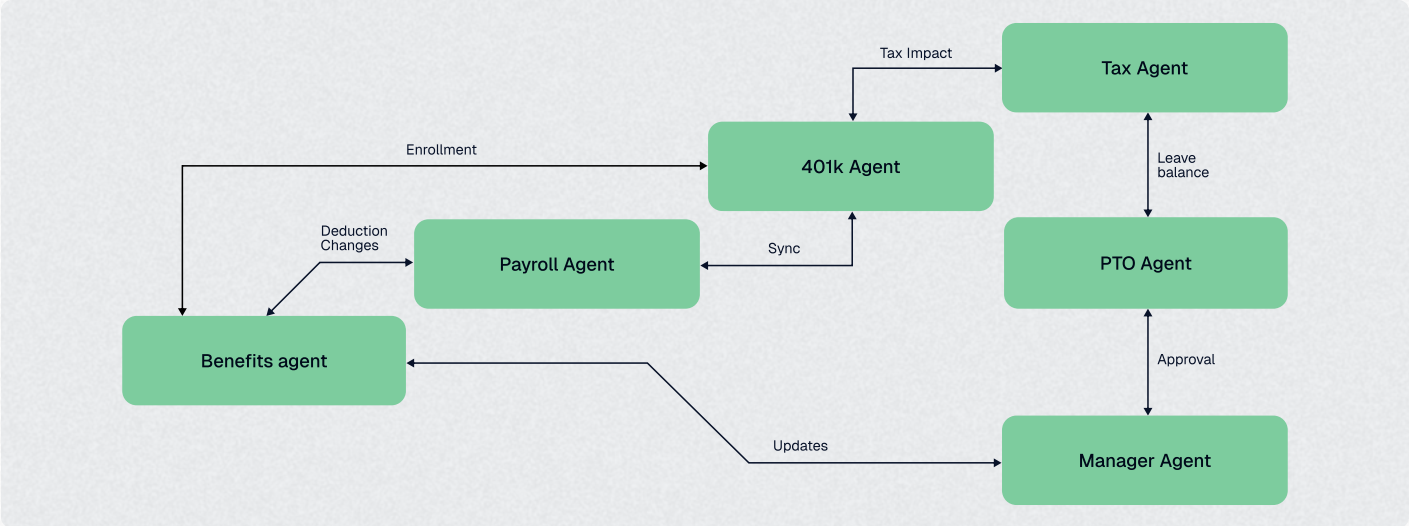


FIGURE 3.2 Decentralized peer-to-peer architecture

In this design, agents communicate directly with neighbors, making local decisions without central coordination. Intelligence emerges from local interactions where no single agent sees the complete picture, but collective behavior solves complex problems. Each agent maintains its own state and coordinates with peers as needed.

Consider an enterprise HR system during open enrollment. The benefits agent handles insurance questions while directly coordinating with the payroll agent about deduction changes. When an employee updates their 401(k) contribution, the retirement agent immediately syncs with the tax agent to recalculate withholdings. The time-off agent detects someone's about to lose unused PTO and directly notifies the manager approval agent to expedite pending requests.

No orchestrator manages these interactions, which means each agent responds to requests and coordinates with peers as needed. When the benefits agent goes down for maintenance, other agents continue operating because they don't depend on a central hub.

In FIGURE 3.2, you can see how your agents would connect in a network. The Benefits Agent talks directly to the Payroll Agent about "Deduction Changes," the 401(k) Agent syncs with the Tax Agent about "Tax Impact," and the PTO Agent sends "Approval" requests to the Manager Agent.

Performance characteristics differ significantly from centralized systems:

- → Token efficiency drops due to potential duplicate work
- → Latency decreases for local decisions
- → Throughput scales linearly with agents
- → Context distributes evenly across the system

This architecture gains resilience because the failure of one agent doesn't crash your entire system. You can scale to hundreds of agents without making architectural changes, since each new agent simply joins the peer network. However, coordinating global behavior becomes challenging when no single agent has complete oversight, and maintaining consistency or enforcing system-wide priorities becomes difficult without a central authority.

The decentralized approach works well when your agents need autonomy and your system must survive partial failures. It becomes problematic when you need guaranteed consistency or centralized decision-making across all agents.

Below is a reference implementation from Langgraph.

```python
from typing import Literal
from langchain_openai import ChatOpenAI
from langgraph.types import Command
from langgraph.graph import StateGraph, MessagesState, START, END

model = ChatOpenAI()

def agent_1(state: MessagesState) -> Command[Literal["agent_2", "agent_3", END]]:
    # you can pass relevant parts of the state to the LLM (e.g., state["messages"])
    # to determine which agent to call next. a common pattern is to call the model
    # with a structured output (e.g. force it to return an output with a "next_agent" field)
    response = model.invoke( ... )
    # route to one of the agents or exit based on the LLM's decision
    # if the LLM returns "__end__", the graph will finish execution
    return Command(
        goto=response["next_agent"],
        update={"messages": [response["content"]]},
    )

def agent_2(state: MessagesState) -> Command[Literal["agent_1", "agent_3", END]]:
    response = model.invoke( ... )
    return Command(
        goto=response["next_agent"],
        update={"messages": [response["content"]]},
    )

def agent_3(state: MessagesState) -> Command[Literal["agent_1", "agent_2", END]]:
    ...
    return Command(
        goto=response["next_agent"],
        update={"messages": [response["content"]]},
    )

builder = StateGraph(MessagesState)
builder.add_node(agent_1)
builder.add_node(agent_2)
builder.add_node(agent_3)

builder.add_edge(START, "agent_1")
network = builder.compile()
```

3. Hierarchical: Multi-Level Management

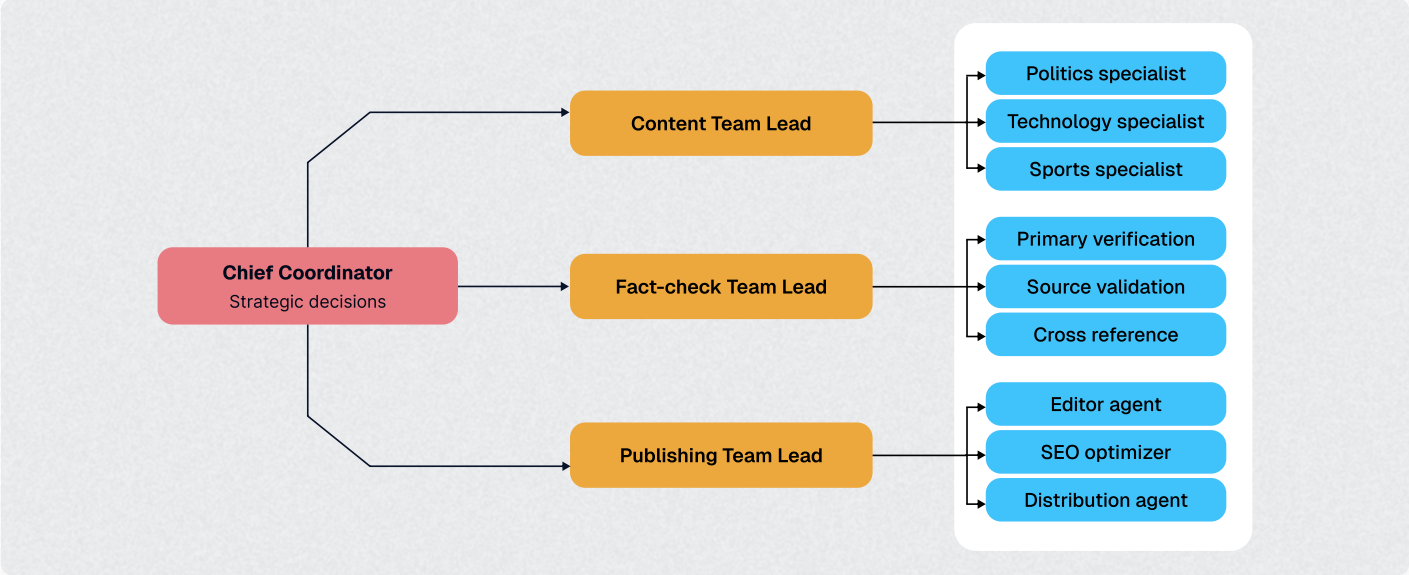


FIGURE 3.3 Hierarchical multi-level architecture with specialized teams

Multiple layers of supervision create a tree structure where specialized teams work under team leaders who report to higher-level coordinators.

When you use this approach, decisions cascade down your hierarchy while information bubbles up through the levels. Each level abstracts complexity for the level above, which means your supervisor agent performs task planning, breaks down work, assigns sub-tasks, and facilitates communication between specialists without overwhelming higher-level coordinators.

A good example of this would be a news aggregation platform. Your top supervisor coordinates between content, fact-checking, and publishing teams. Your content supervisor manages agents for various beats, including politics, technology, and sports. Each beat agent oversees specialized scrapers for different sources. Your fact-checking supervisor manages verification agents using different methods. Information flows up through summaries while specific tasks flow down through assignments.

In FIGURE 3.3, you can see this three-tier structure with your Chief Coordinator at the top, three Team Leads in the middle (Content, Fact-check, Publishing), and specialized agents at the bottom level under each team.

Google's Agent Development Kit exemplifies this pattern. By composing specialized agents in a hierarchy, you build modular and scalable applications. Teams can have specialized supervisors managing domain experts.

This architecture handles complex, multi-domain problems elegantly. The organizational structure feels natural to human teams, but the coordination overhead between levels adds complexity.

Performance balances between extremes:

- → Token efficiency is moderate, with some redundancy between levels
- → Latency is moderate due to multi-hop coordination
- → Throughput is high through parallel teams
- → Context segments by level and team

Think about your organization's structure. Do you have clear reporting lines and specialized departments? Or does your team collaborate directly without hierarchies? Your agent architecture can mirror how your team actually works. If people already report up through managers, hierarchical agents will feel natural. But it's prudent to evaluate your agents to finalize the architecture.

The hierarchical approach is most effective for problems that naturally decompose into sub-problems. Each level should add meaningful abstraction, not bureaucracy.

Below is a reference implementation from Langgraph.

```
from typing import Literal
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState, START, END
from langgraph.types import Command
model = ChatOpenAI()

# define team 1 (same as the single supervisor example above)

def team_1_supervisor(state: MessagesState) -> Command[Literal["team_1_agent_1", "team_1_agent_2", END]]:
    response = model.invoke( ... )
    return Command(goto=response["next_agent"])

def team_1_agent_1(state: MessagesState) -> Command[Literal["team_1_supervisor"]]:
    response = model.invoke( ... )
    return Command(goto="team_1_supervisor", update={"messages": [response]})

def team_1_agent_2(state: MessagesState) -> Command[Literal["team_1_supervisor"]]:
    response = model.invoke( ... )
    return Command(goto="team_1_supervisor", update={"messages": [response]})

team_1_builder = StateGraph(Team1State)
team_1_builder.add_node(team_1_supervisor)
team_1_builder.add_node(team_1_agent_1)
team_1_builder.add_node(team_1_agent_2)
team_1_builder.add_edge(START, "team_1_supervisor")
team_1_graph = team_1_builder.compile()

# define team 2 (same as the single supervisor example above)
class Team2State(MessagesState):
    next: Literal["team_2_agent_1", "team_2_agent_2", "__end__"]

def team_2_supervisor(state: Team2State):
    ...

def team_2_agent_1(state: Team2State):
    ...

def team_2_agent_2(state: Team2State):
    ...

team_2_builder = StateGraph(Team2State)
...
team_2_graph = team_2_builder.compile()


# define top-level supervisor

builder = StateGraph(MessagesState)
def top_level_supervisor(state: MessagesState) -> Command[Literal["team_1_graph", "team_2_graph", END]]:
    # you can pass relevant parts of the state to the LLM (e.g., state["messages"])
    # to determine which team to call next. a common pattern is to call the model
    # with a structured output (e.g. force it to return an output with a "next_team" field)
    response = model.invoke( ... )
    # route to one of the teams or exit based on the supervisor's decision
    # if the supervisor returns "__end__", the graph will finish execution
    return Command(goto=response["next_team"])

builder = StateGraph(MessagesState)
builder.add_node(top_level_supervisor)
builder.add_node("team_1_graph", team_1_graph)
builder.add_node("team_2_graph", team_2_graph)
builder.add_edge(START, "top_level_supervisor")
builder.add_edge("team_1_graph", "top_level_supervisor")
builder.add_edge("team_2_graph", "top_level_supervisor")
graph = builder.compile()
```

4. Hybrid: Strategic Center, Tactical Edges

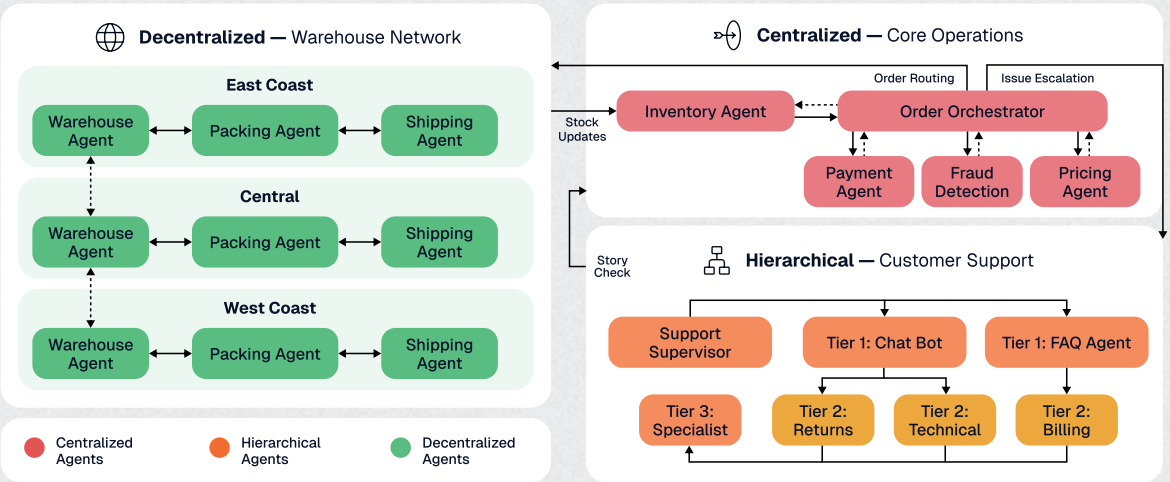


FIGURE 3.4 Hybrid architecture

Hybrid systems bring together centralized control with decentralized flexibility. Different parts of your system can use the right approach for their needs instead of forcing everything into one pattern.

Global decisions flow from central coordinators while local optimizations happen through peer interactions. You get strategic oversight where it matters and tactical flexibility where you need speed.

Say you're building a food delivery platform. Your central orchestrator handles tasks that require consistency and transaction integrity, such as order placement and payment processing. These critical functions stay reliable and coordinated. Once an order is confirmed, your regional agent clusters take over the execution. Restaurant agents coordinate directly with nearby driver agents for pickup timing. Your driver agents negotiate with each other to share efficient routes. Customer notification agents operate independently based on local delivery status.

Your center maintains what must be centralized (payments, order integrity, customer data). Your edges optimize what benefits from local knowledge (routing, timing, real-time adjustments). Each part of your system works at its optimal level without unnecessary overhead.

In FIGURE 3.4, you can see this complexity with three distinct zones: your decentralized warehouse networks on the left (green agents), centralized core operations in the middle (red agents), and hierarchical customer support on the right (orange agents).

In this design, you can balance control and resilience while adapting the architecture to different problem domains within the same system. This flexibility makes hybrid architectures popular for large-scale applications, where a single approach can create unnecessary bottlenecks or complexity.

Performance adapts to requirements:

- → Token efficiency varies based on task distribution
- → Latency optimizes for both global and local operations
- → Throughput combines the benefits of both approaches
- → Context is strategic at center, tactical at edges

However, the complexity increases, and implementation and debugging become more challenging. You need to define the boundary between centralized and decentralized zones carefully, and your agents must understand when to escalate an issue to central coordination versus handling it locally.

With four different architectures to choose from, you need a systematic way to pick the right one for your specific needs.

Architecture	Best For	Token Efficiency	Failure Resilience	Coordination Complexity
Centralized	Simple workflows, strong consistency needs	High (no redundancy)	Poor (single point)	Low
Decentralized	Resilient systems, local optimizations	Lower (duplicate work)	Excellent	High
Hierarchical	Complex domains, team structures	Moderate	Moderate	Moderate
Hybrid	Enterprise systems, mixed requirements	Variable	Good	Very High

FIGURE 3.5 Architecture selection matrix comparing key characteristics

This matrix (FIGURE 3.5) helps you match your requirements to the right architecture. Suppose you need simple workflows with strong consistency, centralized works best. If you're building resilient systems that require local optimizations, a decentralized approach fits better. Complex domains with team structures call for hierarchical approaches. Enterprise systems with mixed requirements benefit from hybrid designs.

Once you've chosen your architecture, you need tools to actually build it. The framework you pick determines how easy or difficult implementation becomes.

Different Frameworks for Agent Coordination

The explosion of agent frameworks reflects different approaches to agent coordination. Each framework makes architectural choices that favor certain patterns. Understanding what each one actually delivers can save you months of development time.

LangGraph

LangGraph models your multi-agent workflows as directed graphs. This makes it natural for complex state management and conditional flows. Your agents become nodes, communications become edges, and state flows through the graph.

The framework works particularly well for hierarchical and hybrid patterns because graphs can represent both reporting relationships and peer connections within the same system. When you need persistent memory and stateful interactions across long-running processes, LangGraph handles the complexity of maintaining context across multiple turns and agent handoffs.

Take the example of a customer onboarding system. Your intake agent collects initial information, your verification agent checks documents, your approval agent makes decisions, and your setup agent configures accounts. LangGraph tracks the entire process state, so if verification fails, the system knows exactly where to restart without losing previous work. Each step builds on the last, while maintaining a full context of what happened before.

Agno

Agno provides a multi-agent architecture that emphasizes performance. The framework claims agent creation at 2μs per agent. This is orders of magnitude faster than alternatives when you need to spawn many agents quickly.

The framework supports all architectural patterns but gets optimized for high-performance scenarios where milliseconds matter. You also get native multi-modal support with a minimal memory footprint (~3.75 KiB per agent), which becomes crucial when you're running many agents simultaneously without overwhelming your system resources.

Say you're building a trading system that needs to analyze 500 stocks simultaneously. Each stock gets its own analysis agent that processes price data, news sentiment, and technical indicators in real-time. Agno can spawn all 500 agents in milliseconds and keep them running efficiently. At the same time, other frameworks might take several seconds to initialize and cause you to miss trading opportunities in fast-moving markets.

Mastra

Mastra brings multi-agent systems to web developers with a TypeScript-first design. Your agents get LLM models with tools, workflows, and synced data. They call your functions, third-party APIs, and access knowledge bases you build.

This approach works particularly well for workflow-centric hybrid architectures where your business logic matters more than pure agent autonomy. The framework provides graph-based state machines that orchestrate complex sequences of AI operations while maintaining tight integration with web services and APIs you already use.

Consider the example of an e-commerce platform where customers can chat with AI about products. Your product agent queries your existing inventory API, your recommendation agent calls your existing ML models, and your order agent integrates with your existing checkout system. All of this runs in TypeScript alongside your existing web application, so you don't need to manage a separate AI infrastructure or worry about data synchronization between systems.

CrewAI

CrewAI focuses on role-based agent collaboration with predefined agent personas and responsibilities. Think of it as hiring a team where each member has a clear job description and responsibilities.

The framework excels at centralized orchestration with specialized agents that maintain persistent behaviors across different tasks. You get quick prototyping with minimal configuration because you simply define roles, assign tasks, and let the framework handle coordination without complex state management or custom orchestration logic.

Say you're automating content creation for your marketing team. Your research agent acts like a market researcher and gathers competitive intelligence using the same methods a human researcher would use. Your writer agent acts like a copywriter and creates blog posts with a consistent brand voice. Your editor agent acts like an editor and reviews content for brand consistency and quality. Each agent maintains its professional persona across different projects, so you get predictable, role-appropriate responses every time.

Google ADK

Google ADK organizes your multi-agent workflows through flexible orchestration patterns. The framework provides Sequential, Parallel, and Loop agents that you can compose into hierarchies.

The framework works particularly well when you need built-in evaluation and structured testing. ADK includes tools to create evaluation datasets and measure agent performance on predefined tasks directly within the framework. When you're building on Google Cloud with Gemini models, you can take advantage of Vertex AI integration for deployment and model management.

Say you're building a news aggregation platform. Your top-level coordinator manages three teams: content collection, fact-checking, and publishing. The content team has specialist agents for politics, technology, and sports, each running parallel searches across different sources. Your fact-checking controller routes claims to verification agents, and the publishing team coordinates final edits and distribution. ADK's evaluation tools also let you track accuracy and behavior consistency across these components.

AWS Strands

AWS Strands takes a model-driven approach where you define prompts and tools, and the model manages planning and execution. It requires minimal code, making it quick to move from prototype to production.

The framework is designed for AWS integration. With native MCP support, you can connect to compatible tools without extra integration work, and the Agent-to-Agent (A2A) protocol enables agents to discover and communicate with others across your organization.

For example, in a code modernization system, one agent could analyze a codebase with MCP tools, another could plan migration strategies using AWS APIs, and a third could generate and test updated code. This runs with relatively little code since Strands handles the orchestration loop and coordination.

Choosing the Right Framework for Your Architecture

Begin with the framework that best aligns with your primary use case and most critical requirements. You can always add specialized components later as your system evolves and your needs become more complex. The key is understanding that framework choice implies architectural decisions, so choose based on your actual requirements rather than feature lists or marketing promises.

You can refer to TABLE 3.1 when selecting a framework based on your performance needs, tech stack, and use case.

Framework	Key Strengths	Best For
LangGraph	• Directed graph modeling • Persistent memory across turns • Handles conditional flows • Full state tracking	Complex workflows with state management
Agno	• Faster agent creation time • low memory usage per agent • Multi-modal support • Minimal resource footprint	High-performance, high-volume operations
Mastra	• Native TypeScript integration • Works with existing REST APIs • Graph-based workflows • No separate AI infrastructure needed	Web applications and TypeScript projects
CrewAI	• Predefined agent personas • Minimal configuration required • Centralized orchestration • Consistent role behaviors	Role-based collaboration and rapid prototyping
Google ADK	• Flexible orchestration • Hierarchical agent composition • Built-in evaluation framework • Native Vertex AI integration	Multi-agent system on GCP
AWS Strands	• Model-driven approach • Native MCP support • Multi-agent with A2A protocol	Production-ready agents on AWS

TABLE 4.3 Different approaches to context management

Making the Architecture Decision

Understanding frameworks helps you implement your chosen architecture, but you still need to pick the right architecture in the first place. With four different patterns and multiple frameworks to choose from, you need a systematic approach to make the right decision for your specific situation.

Refer to FIGURE 3.6 to help you match your main challenge with the right architecture.

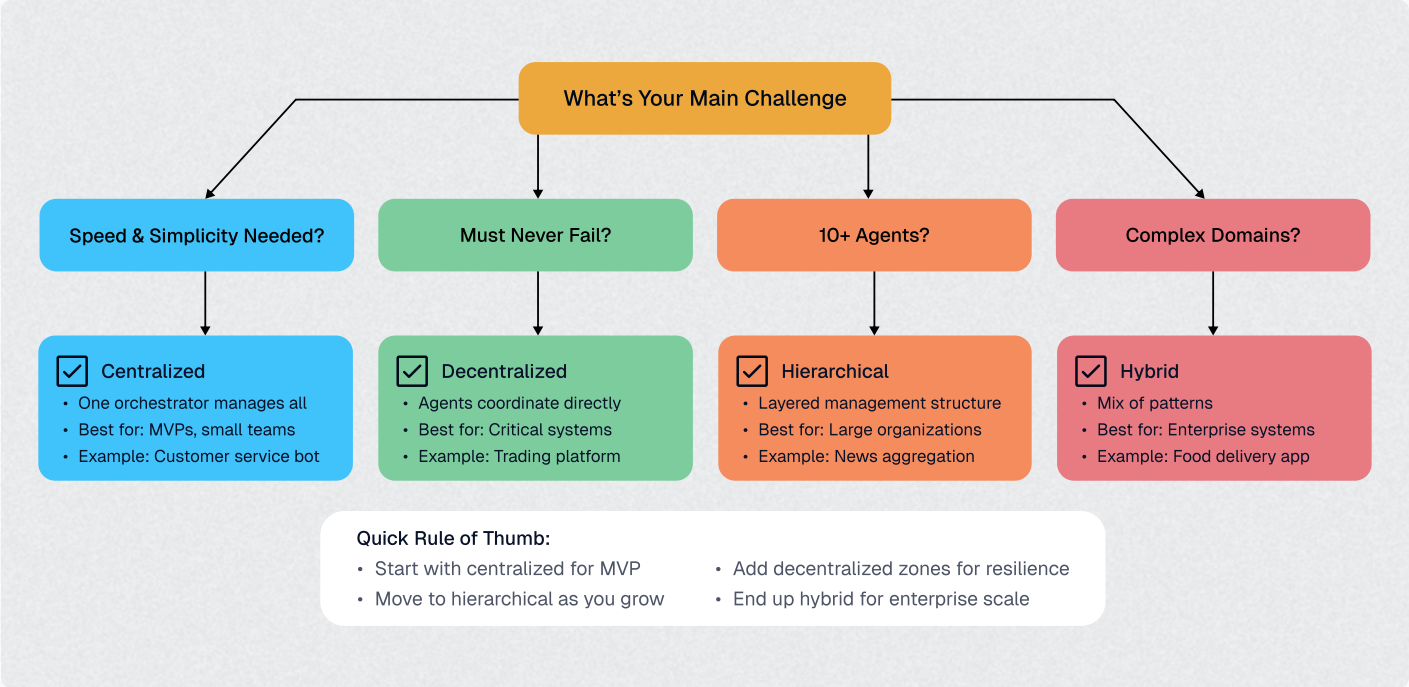


FIGURE 3.6 An architecture decision framework for selecting the right pattern

Choose your architecture based on these factors:

Consistency requirements

The level of data synchronization your system needs determines which architecture works best. When every piece of data must be perfectly synchronized across all agents and you can't tolerate any discrepancies, centralized control ensures nothing gets out of sync because all updates flow through a single point of truth.

When your agents can work with slightly stale data and sync up periodically without breaking functionality, decentralized patterns give you better performance because agents don't need to wait for global coordination before making decisions.

Scale trajectory

When you're starting with small teams of 3-5 agents, simple centralized orchestration works well and keeps complexity manageable while you learn what works.

When planning to scale to dozens or hundreds of agents within months, you need architectural patterns such as hierarchical or hybrid designs that can handle growth without requiring complete system redesigns, which would disrupt your progress.

Team structure

Your architecture should mirror how your organization actually works. If your company has clear reporting structures and specialized departments, hierarchical agent systems will feel natural because they follow the same communication patterns your team already uses. When you have flat organizations where people collaborate directly without rigid hierarchies, peer-to-peer agent networks work better because they match how your team actually makes decisions and shares information.

Problem decomposition

When your tasks can be split into completely separate chunks that don't depend on each other, decentralized approaches let agents work independently without coordination overhead. When your tasks build on each other or need careful sequencing, you need centralized or hierarchical coordination because someone has to manage the dependencies and ensure work happens in the right order.

The Final Verdict

A multi-agent system creates different computational patterns than single agents. Single agents transform inputs into outputs, while multi-agent systems create emergent behaviors through interaction.

All successful implementations share common traits:

Match architecture to problem structure

Your choice between centralized, decentralized, hierarchical, or hybrid should follow from your actual requirements, not what sounds sophisticated.

Understand tradeoffs explicitly

Every architecture choice involves tradeoffs between consistency, performance, complexity, and resilience.

Design for coordination rather than just capability

Adding more agents only helps if you solve the coordination problem. The systems that work spend as much effort on how agents communicate as on what individual agents do.

Evolve based on measured constraints rather than theoretical elegance

Start simple, measure what actually breaks, and add complexity only when you hit real limitations.

The most common mistake is building a multi-agent system when a single well-designed agent would work better. Coordination complexity often outweighs specialization benefits, especially for smaller problems. Before building, test whether your problem actually needs multiple agents.

EXERCISE

Before building, answer these questions:

Your Requirements

1. What's the main problem you're solving?
2. How many agents do you need now? In 6 months?
3. What happens if one agent fails?
4. Do agents need the same data, or can they work with local information?

Your Constraints

1. Do you need perfect data sync, or can agents work with slightly stale data?
2. Sub-second responses or 2-3 second delays acceptable?
3. 3 agents or 300?
4. Clear hierarchies or flat structures in your organization?

Your Decision

1. Based on Figure 6, which architecture matches your main challenge?
2. What would break first if your system grew 10x?
3. Which framework supports your chosen architecture?

Most requirements point clearly to one architecture.

What You've Learned

Architecture significantly influences what your system can do. Centralized orchestration ensures strong consistency in handling simple workflows. Decentralized coordination provides resilience and local optimization. Hierarchical structures work for complex domains with natural team divisions. Hybrid approaches combine patterns for enterprise-scale systems.

But choosing the right architecture is just the beginning. Once you start building in production, you encounter a common challenge: context management. As your agents handle real workloads with large inputs and complex histories, context becomes the bottleneck.

In Chapter 4, we'll tackle the practical challenge of context engineering. We'll explore the four types of context that agents must manage, the failure modes that impact system performance, and the five strategic approaches for keeping your agents performant as context evolves.

Galileo

Chapter

04

Context Engineering for Multi-Agent Systems

04 Context Engineering for Multi-Agent Systems

Your multi-agent system is architecturally sound. You've chosen the right pattern, selected the appropriate framework, and your agents have clear responsibilities. Everything should work.

Then you deploy to production. An agent analyzing reports begins to hallucinate data points that don't exist. It gets stuck in loops and repeatedly calls the same API with the exact same parameters. It cites sources it never retrieved. Your carefully designed system fails, and you wonder what went wrong.

The problem is context.

For every token your agents generate, they process roughly 100 tokens of input. That 100:1 ratio means context management determines whether your system works or collapses. Most teams load everything into context and expect the model to sort it out.

This chapter shows you how context engineering makes or breaks multi-agent systems. You'll learn the distinction between memory and context, understand the four types of context your agents handle, recognize the failure modes that destroy performance, and get five approaches for keeping agents reliable as context grows.

Memory versus Context

The distinction between memory and context determines your system's architecture. Most teams treat these as the same thing. They're not, and the distinction determines whether your system scales.

Context is your agent's working memory.

It's the immediate information available during inference, analogous to RAM in a computer. It's what the model can "see" right now, limited by the context window, expensive to maintain, and cleared between independent sessions.

Every token in context directly influences the model's response, for better or worse. With a large input-to-output ratio (ex., 100:1), context management becomes the dominant cost factor.

Memory is your agent's long-term storage.

It's the persistent information stored externally that survives beyond individual interactions. It's unlimited in size, cheap to store, but requires explicit retrieval to be useful. Memory doesn't directly influence the model unless actively loaded into context. Think of it as your hard drive with vast capacity but access overhead. Reading memories is another retrieval problem; writing memories requires nuanced strategies.

Aspect	Context (Working Memory)	Memory (Long-term Storage)
Cost	Expensive: Every token costs money, 10x more if uncached	Cheap: Storage costs are negligible
Size	Limited: by the context window	Unlimited: Store millions of documents
Speed	Immediate: Direct influence on response	Slow: Requires retrieval before use
Persistence	Volatile: Cleared between sessions	Persistent: Survives across sessions
Influence	Direct: Affects every model decision	Indirect: Must load into context first
Performance	Degrades: after 30k tokens	No degradation: just retrieval overhead

TABLE 4.1 Context versus memory characteristics

TABLE 4.1 breaks down the key differences that determine how you should design your system.

At scale, the distinction between memory systems and RAG becomes less pronounced. Both involve selecting relevant information and loading it into context. The difference lies in intent. RAG typically handles knowledge retrieval while memory systems manage agent state and learned patterns.

FIGURE 4.1 below shows how different products solve the memory-to-context problem. For instance, Manus achieves 100:1 compression using file system offloading. Anthropic's multi-agent system uses parallel subagents with isolated contexts. Claude Code relies on auto-compaction at a 95% threshold. Each approach balances compression, accuracy, and implementation complexity.

Product/team	Primary strategy	Key innovation	Performance impact	Trade-offs
Manus	File system offloading	Reversible compression, todo.md pattern	100:1 compression, 10x cost reduction	Added file I/O complexity
Anthropic multi-agent	Context isolation	Parallel subagents with compression	90.2% accuracy improvement	15x token usage, coordination overhead
Claude code	Auto-compaction	95% threshold summarization	Handles unlimited session length	Potential information loss
Windsurf	Multi-technique RAG	AST parsing + grep + embeddings	3x retrieval accuracy	Implementation complexity
Cursor	Rule files	Procedural memory (.cursorsrules)	Consistent behavior patterns	Manual maintenance required
HuggingFace	Sandboxed execution	CodeAgent with isolated environments	80% context size reduction	Environment management overhead

FIGURE 4.1 How different products solve the memory-to-context problem

These strategies work because they make smart choices about what to keep in context versus what to store in memory. TABLE 4.2 provides a decision framework for your own system.

Keep in Context	Store in Memory
Current task objectives and constraints	Historical conversations and decisions
Recent tool outputs (last 3 to 5 calls)	Learned patterns and preferences
Active error states and warnings	Large reference documents
Immediate conversation history	Intermediate computational results
Currently relevant facts	Completed task summaries

TABLE 4.2 Context versus memory decision matrix

EXERCISE

Map your current system:

1. List everything currently in your agent's context
2. For each item, ask: "Does this need to influence every single decision?"
3. If no, move it to memory with a retrieval strategy
4. Calculate your potential cost savings (context tokens × cost per token × sessions per day)

This distinction may disappear as models improve and context windows expand. Until then, you need intentional design. Memory must be organized for retrieval. Context must be managed to optimize both cost and performance.

Four Types of Context

Understanding what belongs in context starts with recognizing the different types. Your agents handle four distinct kinds, each with different challenges.

1. Instructions context includes system prompts, few-shot examples, and behavioral guidelines that define how the agent operates. This context tends to be stable but can grow large with complex instructions. Poor instruction context leads to inconsistent behavior and misaligned responses.

2. Knowledge context encompasses facts, memories, retrieved documents, and user preferences that inform decisions. This dynamic context changes based on the task and can quickly overwhelm the context window. The challenge lies in selecting relevant knowledge while avoiding information overload.

3. Tools context contains tool descriptions, feedback from tool calls, and error messages that enable agents to interact with external systems. Models perform poorly when given multiple tools. The Agent Leaderboard shows every model degrades with tool count, making tool context management critical for performance.

4. History context preserves past conversations, previous decisions, and learned patterns from earlier interactions. While valuable for continuity, historical context can introduce contradictions and outdated information that confuse the agent.

Each type requires different handling.

- → Instructions context needs careful curation to stay focused
- → Knowledge context demands smart retrieval to stay relevant
- → Tools context benefits from dynamic selection rather than loading everything upfront
- → History context needs pruning to remove contradictions while preserving useful patterns

---

The next challenge is recognizing when these context types fail. Each has specific failure modes that affect agent performance.

Understanding Context Failure Modes

Drew Breunig identified four distinct patterns that everyone building agents will eventually encounter. Understanding these failure modes is critical for building reliable systems.

EXERCISE

Before reading about failure modes, think about your current agent system:

- Which responses feel "off" but you can't pinpoint why?
- Where do agents repeat mistakes even after corrections?
- When do agents ignore instructions they previously followed?

Keep these scenarios in mind as you read. You'll likely recognize which failure mode is affecting you.

1. Context Poisoning: When Errors Compound

Context poisoning occurs when a hallucination or error enters the context and gets repeatedly referenced, compounding the mistake over time. FIGURE 4.2 below shows how this cascade works. The agent misidentifies a product model, provides wrong troubleshooting based on that error, references those incorrect steps in subsequent responses, and suggests incompatible accessories.

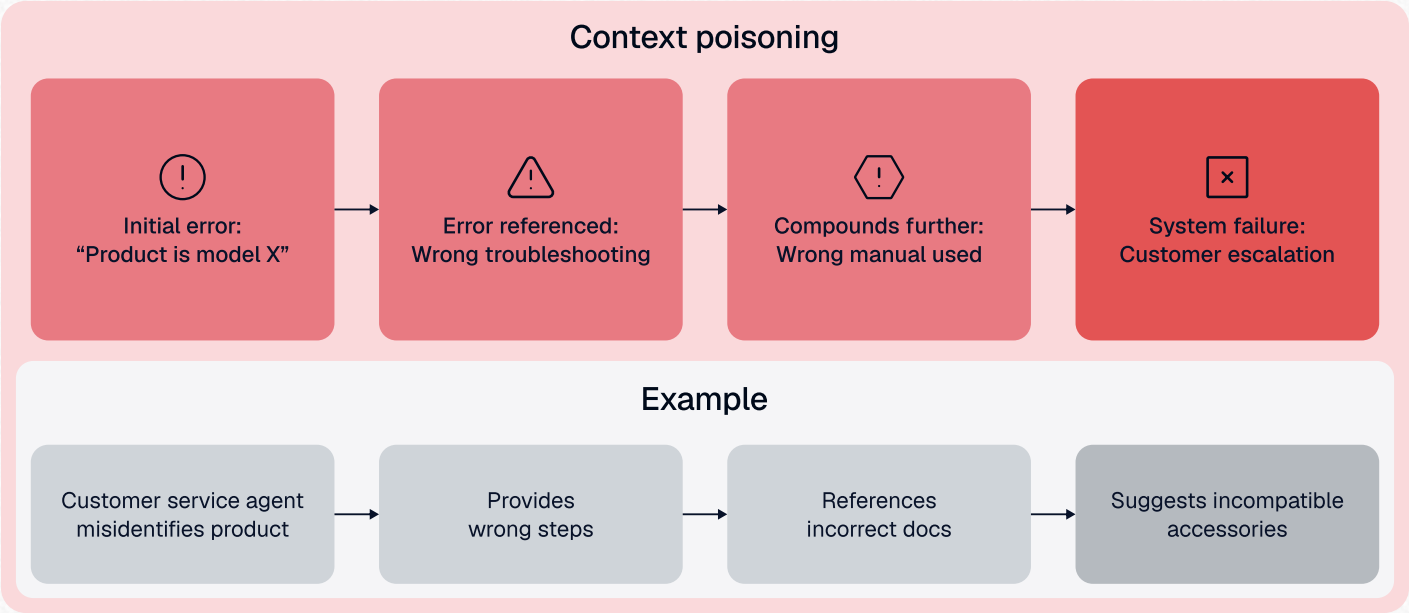


FIGURE 4.2 Context poisoning cascade from initial error to system failure

The DeepMind team documented this with their Pokémon-playing Gemini agent. The agent misidentified the game state once, and that error got embedded in the goals section. Because goals are referenced at every decision point, the false information reinforced itself. The agent spent dozens of turns pursuing impossible objectives and was unable to recover because the poisoned context kept validating the error.

The pattern repeats across domains because the mechanism stays the same. Consider the following examples:

Customer Service Agent:

- **Initial error:** Misidentifies the customer's product model
- **Cascade:** Provides wrong troubleshooting steps → References incorrect manual → Suggests incompatible accessories
- **Result:** Frustrated customer, multiple escalations, lost trust

Code Generation Agent:

- **Initial error:** Imports the wrong library version
- **Cascade:** Uses deprecated APIs → Generates incompatible syntax → Creates cascading type errors
- **Result:** Completely unrunnable code requiring manual rewrite

Research Agent:

- **Initial error:** Misunderstands the research question scope
- **Cascade:** Searches wrong domain → Cites irrelevant sources → Draws incorrect conclusions
- **Result:** Entire research output invalidated

As you'll see in the above cases, a single wrong piece of information gets written into the context. Every decision made after that point is influenced by the corrupted context. The agent can't distinguish between accurate information and the error because both exist in the same context window with equal weight.

Recovery becomes nearly impossible once context poisoning sets in. Clearing the entire context works, but you lose all progress. Asking the agent to reconsider specific points rarely helps because the poisoned information keeps influencing the response. The agent trusts its own context more than external corrections.

2. Context Distraction: The Attention Problem

Context distraction occurs when accumulated history becomes so long that the model focuses on pattern-matching from past interactions instead of reasoning about the current situation. See FIGURE 4.3.

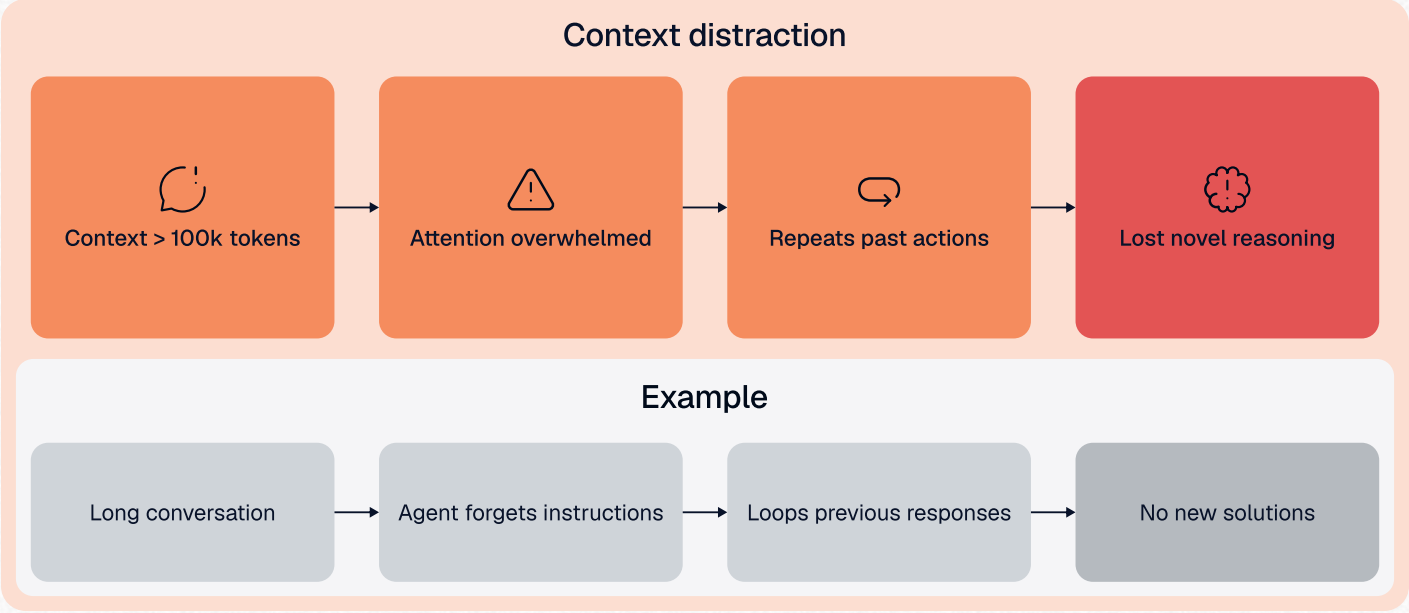


FIGURE 4.3 Context distraction progression from large context to lost reasoning

The Gemini 2.5 team saw this firsthand when their agent's context grew beyond 100,000 tokens. The agent stopped generating novel solutions and started repeating actions from its vast history, even when those actions didn't fit the current problem.

The reason that happens is because models have a finite attention capacity. As context grows, more attention gets allocated to the immediate context at the expense of the model's trained knowledge. Instead of thinking through the problem, the agent searches its history for similar patterns and copies them.

Databricks researchers found something worse. When models hit their distraction threshold, they often ignore instructions entirely and just summarize whatever's in the context. You ask for analysis, you get a recap. You request a decision, you get a list of what happened before.

Your code generation agent might show this pattern. Early in the session, it writes clean, well-structured code that follows best practices. As context grows with multiple iterations, the code quality degrades. The agent starts copying patterns from earlier attempts rather than applying its knowledge of good design. By the 20th iteration, it's repeating the same mistakes it made in the 12th.

EXERCISE

Spot the Pattern in Your Logs

Look at your agent's responses from the beginning of a session versus the 20th turn. Does the quality drop? Does it start repeating solutions that didn't work earlier? Does it ignore new information you've provided? These are signs your agent has hit its distraction threshold.

3. Context Confusion: Too Many Tools

Context confusion arises when your agent has access to numerous tools, which makes it difficult for them to select the right one reliably (FIGURE 4.4). The Berkeley Function-Calling Leaderboard shows consistent degradation across models as the tool count increases.

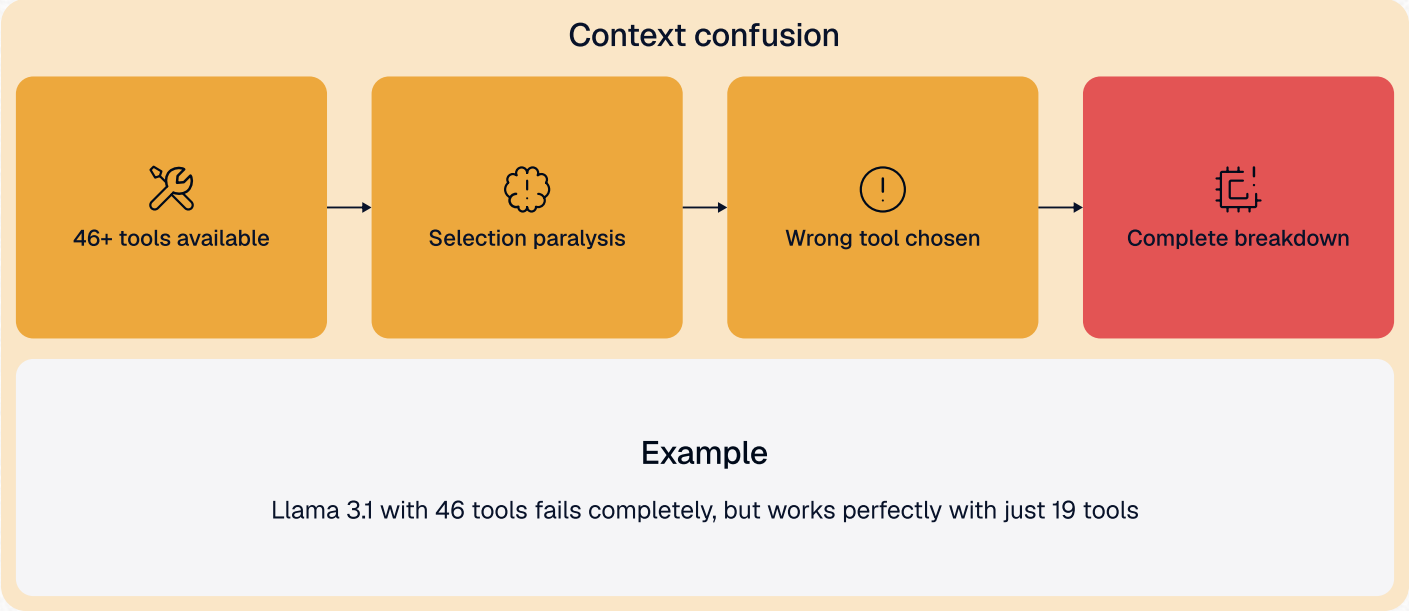
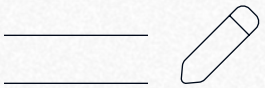


FIGURE 4.4 Context confusion from tool overload to complete breakdown

The findings seem strange at first. A quantized Llama 3.1 8B has a 16,000-token context window. Researchers gave it access to 46 tools from the GeoEngine benchmark, which used roughly 3,000 tokens of context. The agent had plenty of room but still failed completely. When they reduced the tools to just 19, it succeeded.

Each tool description creates decision points. Your agent needs to understand what each tool does, when to use it, and how it differs from similar tools. With five tools, these distinctions stay clear. With fifty tools, the boundaries blur. The agent struggles to pick between tools that sound similar or have overlapping functionality.

EXERCISE



Count Your Tools

Open your agent's tool configuration. How many tools does your agent have access to right now? List them. Which ones does it actually use? Which ones look similar to each other from the agent's perspective? Does all of them have well-defined parameters?

Performance degrades as the tool count increases, regardless of the remaining space in the context window. The problem has to do with the signal-to-noise ratio. Each additional tool makes the descriptions of the existing tools less distinct.

4. Context Clash: Information at War

Context clash is the most complex failure mode. Your accumulated context contains contradictory information that derails the agent's reasoning. FIGURE 4.5 shows how contradictions lead to a loss in decision-making ability.

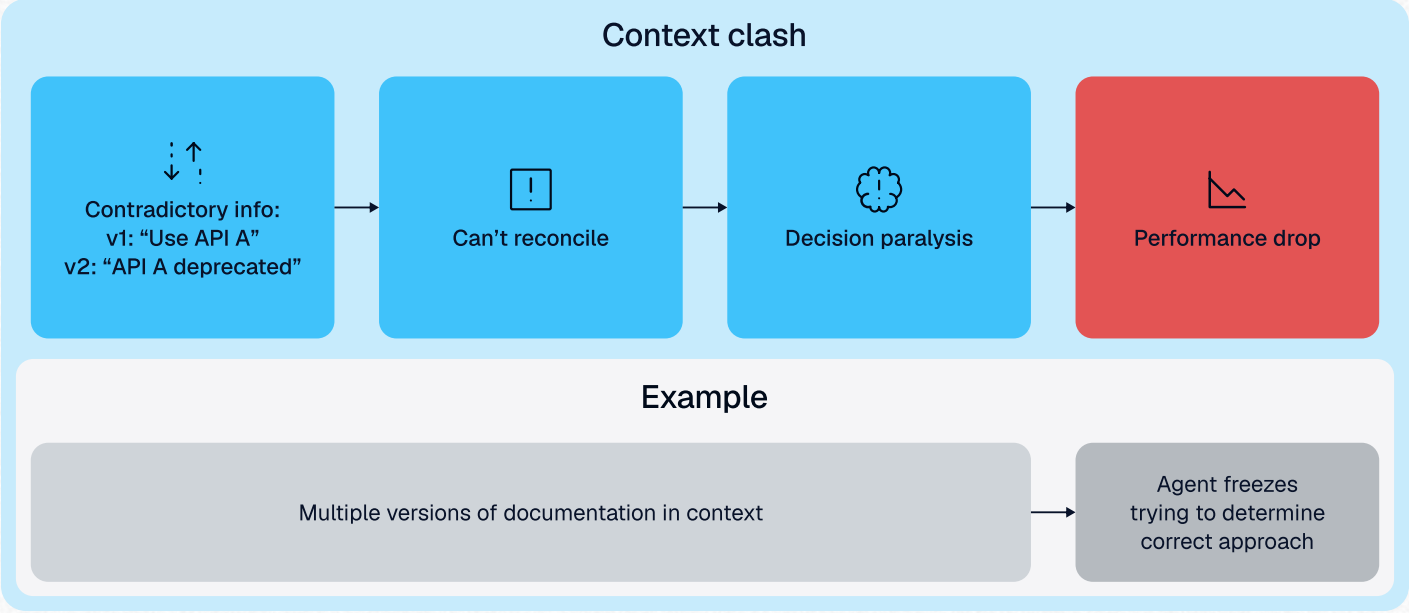


FIGURE 4.5 Context clash from contradictory information to performance degradation

Microsoft and Salesforce researchers tested this by taking standard benchmark tasks and spreading them across multiple conversation turns instead of single prompts. Performance dropped 39% on average across all tested models.

The researchers found that when models take a wrong turn in a conversation, they get lost and do not recover. The agent makes a mistake, receives correction, acknowledges the correction, but then makes decisions based on the original error anyway. The context contains both the error and the correction, and the agent can't consistently prioritize the correction over the error.

The problem compounds with each correction. First mistake, first correction. Second mistake, second correction. Third mistake, third correction. Now your context contains three errors and three corrections, and the agent has to navigate which information remains current. It doesn't always choose correctly.

Remember that context clash differs from context poisoning. Poisoning involves a single error that propagates. Clash involves multiple contradictory pieces of information competing for influence.

Five Approaches to Managing Context

You've seen how context fails. Now you need strategies to prevent those failures. Lance Martin's framework organizes context engineering into five approaches, each solving specific problems.

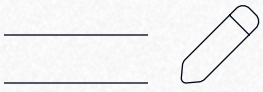
1. Offloading: Keep Heavy Data External

Offloading means storing large outputs outside the context window. Instead of loading a 50,000-token webpage into context, you save it to a file and pass the agent a 500-token summary plus the file path. The agent can retrieve the full content if needed.

The Manus team treats their file system as unlimited external memory. This approach achieves 100:1 compression while maintaining full information recovery. When their agent analyzes a large document, it writes the key findings to a summary file and stores the original separately. The context stays lean, but nothing gets lost.

The trade-off is summarization quality. Compress too aggressively and you lose details that matter later. Compress too little and you're still burning tokens. The key is reversibility. You can always retrieve the full content when the summary proves insufficient.

EXERCISE



Think about your current system. Which outputs could you offload to external storage? What would a 10:1 compression ratio save you in monthly costs?

2. Context Isolation: Split the Work

Context isolation divides tasks across multiple agents that operate independently, each with its own context window.

Anthropic's research system demonstrates this approach when analyzing complex questions. The system spawns specialized sub-agents that explore different aspects simultaneously. One agent investigates economic factors, while another examines environmental data, and a third reviews policy implications. Each works in isolation, then a coordinator synthesizes their findings into a single response.

This approach improved performance by 90% despite using 15 times more tokens overall. The gains come from parallelization and specialization. Each agent maintains a focused context on its specific domain without getting overwhelmed by information from other domains. The coordinator only needs to process summaries rather than raw data from all the explorations.

The pattern works well for research, analysis, and other read-heavy tasks where agents can work without coordinating during their investigation phase. However, it may not be very effective for tasks like coding, where changes in one file ripple through others, and the coordination overhead undermines the benefits gained from isolation.

3. Retrieval: Fetch What You Need

Retrieval systems store information externally and load only relevant pieces into context based on the current task. The approach sounds complex, but simple implementations often work surprisingly well. A basic metadata index with straightforward keyword matching can outperform elaborate vector search setups for many workflows.

Windsurf combines multiple retrieval techniques to improve accuracy. Embedding search finds semantically similar content while grep locates exact matches. Knowledge graphs track relationships between concepts, and AST parsing understands code structure. This multi-method approach achieves three times better accuracy than relying on any single technique because different types of queries benefit from different retrieval methods.

The challenge is knowing what to retrieve. Retrieve too little, and the agent lacks the necessary information to make good decisions. Retrieve too much and you're back to context overload, which defeats the purpose of selective retrieval.

EXERCISE

Map Your Retrieval Needs

Take a recent agent query that failed or performed poorly:

1. List every piece of information your agent accessed to answer that query
2. Mark, which pieces were actually used in the final response
3. Identify which relevant information was missing
4. Calculate your retrieval efficiency: (used items ÷ retrieved items) × 100

If your efficiency is below 40%, you're retrieving too much noise. Above 80% but missing key information means you're retrieving too narrowly. The sweet spot is 50-70% efficiency with all critical information included.

4. Compaction: Summarize and Prune

Compaction removes or condenses information that is no longer relevant to the current task. Claude Code's auto-compaction triggers when context usage reaches 95%, summarizing the entire conversation while preserving objectives and key decisions. The system compresses old tool outputs and removes resolved errors, shrinking the context without losing the thread of what the agent is trying to accomplish.

The risk is losing information that becomes relevant later in the conversation. Aggressive pruning saves tokens but creates gaps in the agent's understanding when it needs to reference something you've already discarded. The safer approach combines compaction with offloading, summarizing aggressively while keeping the originals accessible in external storage. This gives you the token savings from compression with a recovery path when you need the full details.

5. Caching: Reuse What You've Processed

Caching stores processed context so you don't pay to reprocess it in subsequent turns. The Manus team found that proper KV-cache optimization provides a 10-fold cost reduction because each message after the first reuses the cached context from earlier turns, rather than reprocessing everything from scratch.

The catch is that caching only addresses cost and speed. It doesn't fix context rot or quality degradation. A poisoned context stays poisoned whether cached or not, and a distracted agent remains distracted even when the distraction is cached. Caching makes your system faster and cheaper to run, but it does nothing to improve its reliability or accuracy.

Each of these five approaches solves different problems with different trade-offs. TABLE 4.3 breaks down when each approach makes sense for your specific situation and what risks you take on by choosing it.

Approach	Best For	Cost Impact	Complexity	Primary Risk
Offloading	Large documents, web pages, tool outputs	High savings (10-100x compression)	Low	Information loss if summaries drop critical details
Isolation	Research, parallel analysis, independent subtasks	Higher token usage (10-20x) but better performance	Medium	Coordination overhead for interdependent work
Retrieval	Large knowledge bases, codebases, document collections	Medium savings (5-10x)	Medium	Missing relevant information or retrieving noise
Compaction	Long conversations, iterative refinement	High savings (5-20x)	Low	Permanent loss of information that becomes relevant later
Caching	Repeated context access, multi-turn conversations	Very high savings (10x per turn)	Low	No improvement to accuracy or reliability

TABLE 4.3 Different approaches to context management

Take a moment to map your context problem to these solutions before moving forward.

EXERCISE

Map Your Context Problem

1. Identify your biggest context failure mode from the previous section (poisoning, distraction, confusion, or clash)
2. Which of these five approaches would address that specific failure?
3. What would you implement first, given your current constraints?
4. Calculate the expected cost savings or performance improvement you'd gain

Write down your answers. This exercise helps you move from understanding the problem to planning your solution.

How to Apply these Principles

You now have five strategies for managing context, but applying all of them would be expensive and time-consuming. The question now is which problems actually need these solutions and which ones you can solve more simply. The decision depends on your specific constraints around context size, cost, latency, and accuracy requirements.

In FIGURE 4.6, you'll see four threshold ranges for context size with corresponding strategies, from simple caching for small contexts to multi-agent architectures for contexts exceeding 100,000 tokens.

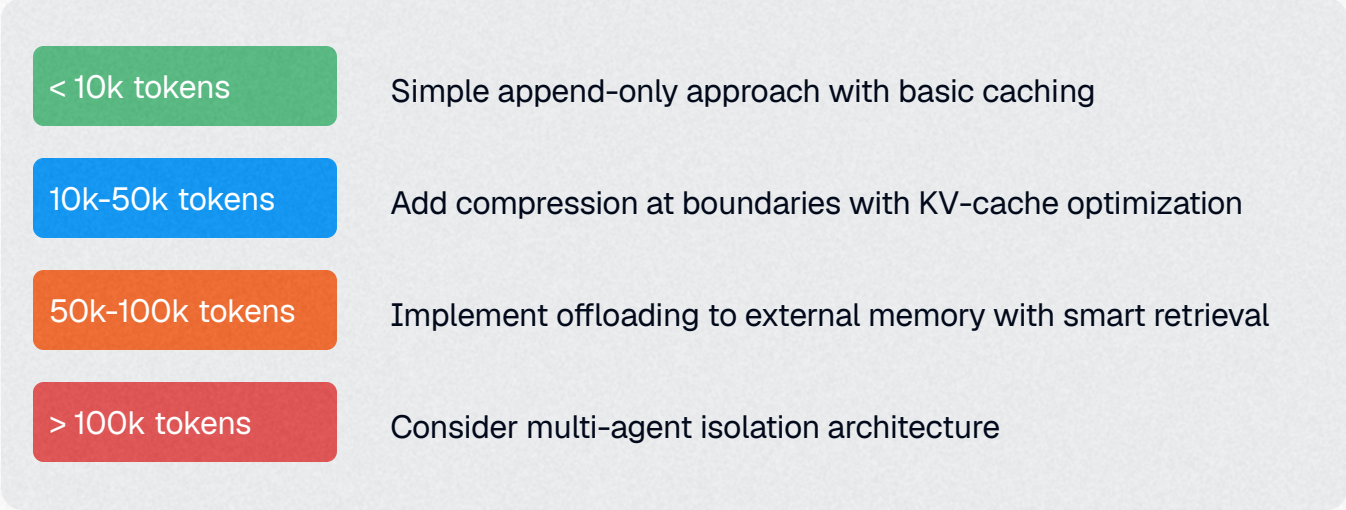


FIGURE 4.6 Context size thresholds and recommended approaches

You can check four priority criteria (parallelizable tasks, cost reduction, low latency, high accuracy) with their corresponding context management approaches in FIGURE 4.7.

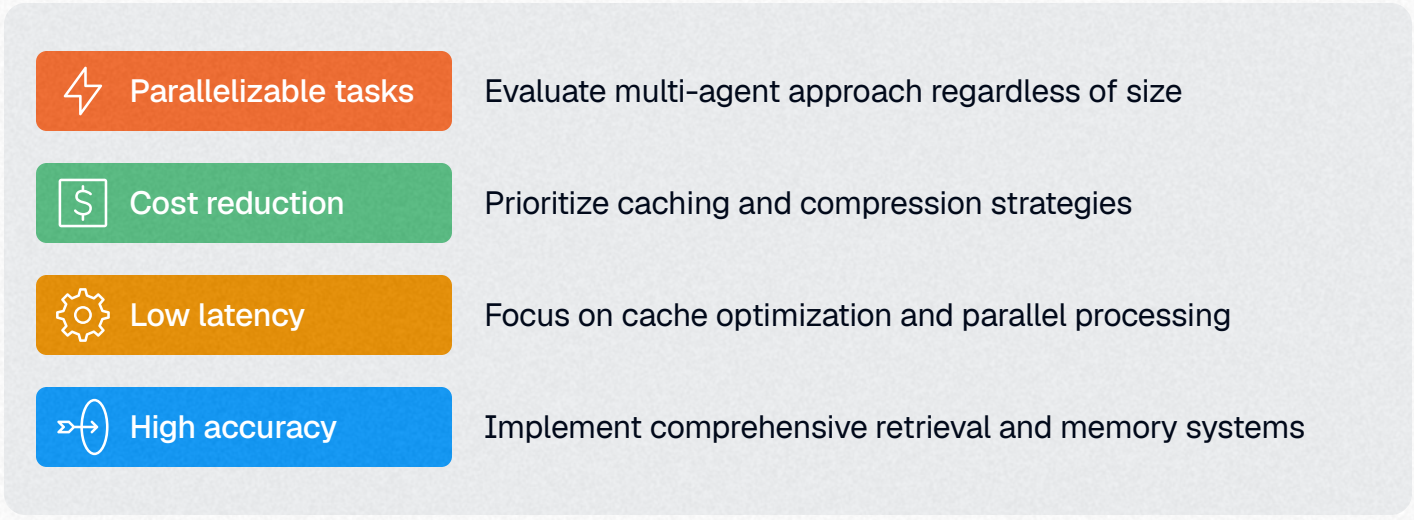


FIGURE 4.7 Priority-based context strategy selection

You'll find five anti-patterns in context management alongside their consequences and the correct approaches to use instead in FIGURE 4.8.

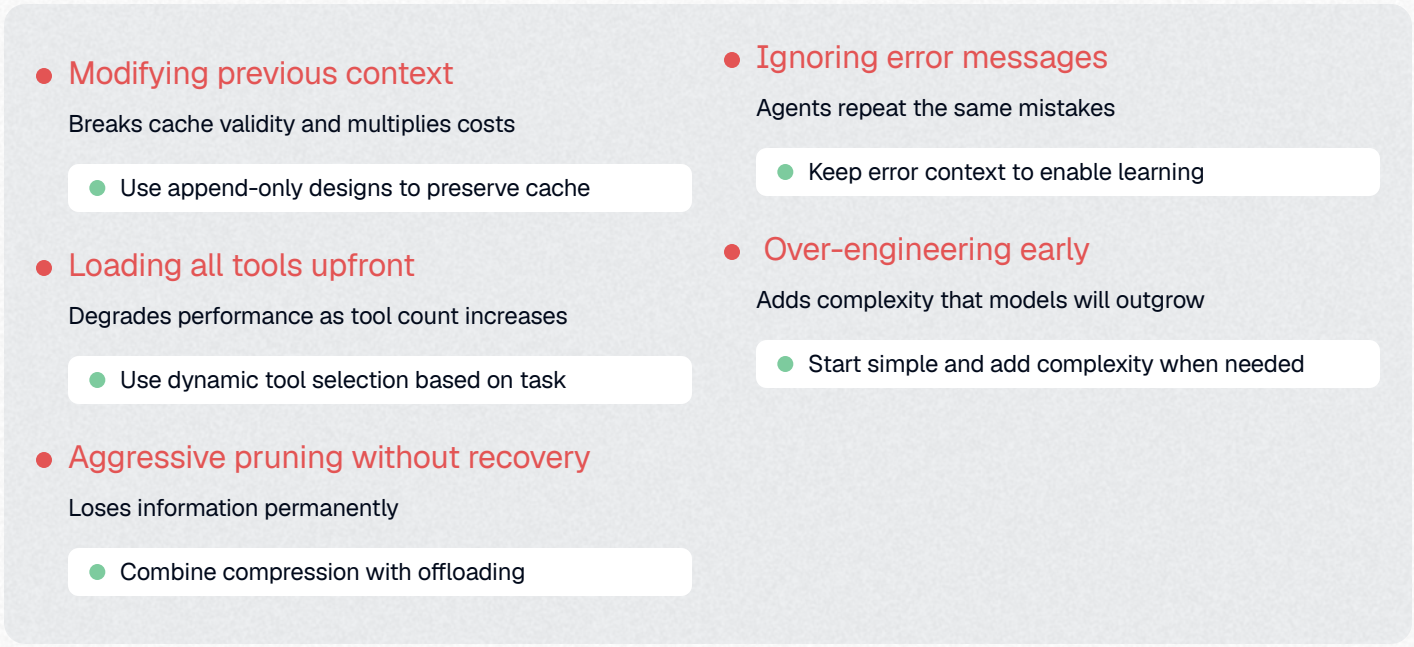


FIGURE 4.8 Common context engineering mistakes and corrections

Here's how you can apply the decision guide to improve your agent's performance step by step.

Say you're building a research agent that analyzes academic papers. Your first implementation loads 15 full papers into context, consuming 80,000 tokens per query. The agent works but runs slowly and costs $2 per query.

Step 1 Check Context Size Threshold: See `FIGURE 4.6`. Your 80,000 tokens falls in the 50k-100k range, which recommends offloading to external memory with smart retrieval.

Step 2 Implement Offloading: Store full papers externally. Load only abstracts and key sections into context. The agent can still access full papers when needed, but context drops to 12,000 tokens. Cost per query falls to $0.30.

Step 3 Check Priority Requirements: Now see `FIGURE 4.7`. Accuracy matters for research tasks, which recommends implementing comprehensive retrieval systems.

Step 4 Add Smart Retrieval: Build a retrieval system that selects papers based on relevance. Paper selection accuracy improves from 65% to 89%.

Step 5 Review Anti-Patterns: Now refer to `FIGURE 4.8`. You were planning to load all 50 available search tools upfront. The guide shows that this degrades performance.

Step 6 Fix Tool Loading: Implement dynamic tool selection that exposes only the five most relevant search tools based on query type. The accuracy of tool selection improves from 72% to 94%.

Outcome Your research agent now runs faster, costs 85% less, and performs more accurately than your initial implementation.

Your Implementation Roadmap

You've learned about the failure modes, solutions, and measurement tools. The question now is where to start. Building context engineering infrastructure occurs in phases, each with its own specific goals and expected outcomes (FIGURE 4.9).

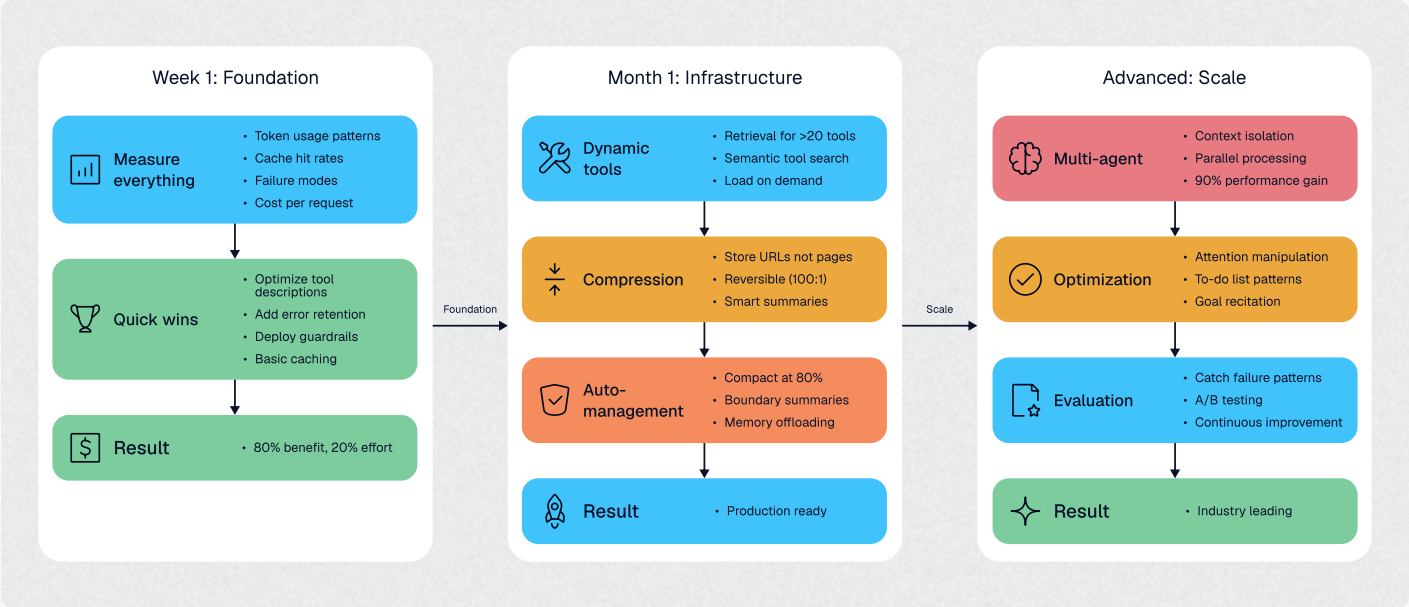


TABLE 4.9 Implementation Roadmap

Week 1: Foundation

Start with measurement because you can't optimize what you don't understand. Deploy comprehensive logging to capture every agent interaction, token usage, cache hit rates, and failure patterns. This baseline reveals where you're losing money and performance.

Your first week should focus on the highest-impact, lowest-effort optimizations:

1. **Optimize tool descriptions** for clarity and distinctiveness so your agent can pick between them more easily
2. **Add basic error retention** by keeping failed attempts in context, which prevents repeated mistakes and costs nothing to implement
3. **Deploy production guardrails** for safety and reliability

These changes take hours to implement, but often provide 80% of the benefit with 20% of the effort. The research agent example from earlier showed this pattern. Simply keeping conversation context about the topic scope improved interdisciplinary query success from 65% to 89%. These quick wins build momentum and demonstrate value before you invest in complex infrastructure.

Month 1: Infrastructure

Once you've captured the easy gains, build your core infrastructure:

1. **Implement dynamic tool retrieval** when you exceed 20 tools, so your agent sees only the most relevant options for each query.
2. **Add reversible compression** by storing URLs instead of web pages and file paths instead of contents, which gives you token savings with a recovery path when you need full details.
3. **Deploy auto-summarization** at 80% context capacity to prevent hard failures when context fills up.
4. **Create a simple memory system** with basic file storage so your agents can access information without loading it all into context.

Each change gets measured against your baseline using the tools covered in the previous section. The Timeline View shows whether compression actually speeds up the process. Action Completion shows whether dynamic tool selection improves success rates. You iterate based on data rather than intuition.

Advanced Phase: Scale

Only after mastering the fundamentals should you explore advanced techniques:

1. **Multi-agent architectures** make sense for truly parallelizable tasks, but Anthropic's research showed 90% improvement at 15x token cost
2. **Attention manipulation** through to-do lists and goal recitation reduces drift in long-running tasks but requires careful tuning
3. **Evaluation insights** help catch emerging failure patterns before users notice them, but you need enough data to make them meaningful

These optimizations require significant engineering investment. Make sure your basics work well before adding this complexity. The systems that fail are those that skip straight to advanced techniques without establishing measurement and implementing quick wins first.

EXERCISE

Audit Your Current Context

Step 1:

Baseline Assessment (15 minutes)

Open your agent's most recent conversation logs and answer:

1. What's your average context size in tokens?
2. How many tokens are instructions vs. knowledge vs. tools vs. history?
3. At what point does your context typically fill up?
4. What's your input-to-output token ratio?

Step 2:

Identify Your Failure Mode (10 minutes)

Review your last 10 failed agent interactions.
For each failure, identify which pattern occurred:

- → Did an early error spread through the conversation? (Poisoning)
- → Did the agent start repeating old patterns? (Distraction)
- → Did the agent repeatedly pick the wrong tool? (Confusion)
- → Did the agent ignore corrections? (Clash)

Tally your results. Your most common failure mode is your starting point.

Step 3: Calculate Potential Savings (10 minutes)

Based on your baseline (see TABLE 4.4):

Metric	Current State	With Optimization	Calculation
Context tokens per query	_____	_____ ÷ 10	Offloading compression
Cost per 1K queries	_____	_____ × 0.1	10x from caching
Tool selection failures	_____	_____ × 0.5	Dynamic tool retrieval

TABLE 4.4 Potential savings

Step 4: Choose Your First Optimization (5 minutes)

Based on your failure mode and potential savings, pick one technique to implement this week:

- **If poisoning dominates:** Implement error retention and validation
- **If distraction dominates:** Add auto-summarization at 80% capacity
- **If confusion dominates:** Deploy dynamic tool selection
- **If clash dominates:** Implement reversible compression with recovery

Write down your chosen technique and the metric you'll use to measure success.

Expected outcome: You now have a concrete starting point for context engineering with measurable goals.

What You've Learned

Context engineering determines whether your multi-agent system works or fails. You now understand the distinction between memory (cheap, unlimited storage) and context (expensive, limited working memory), recognize the four failure modes that destroy agent performance, and have five proven approaches for managing context as it grows. TABLE 4.5

Key Concept	Critical Insight
The 100:1 Rule	Every output token requires ~100 input tokens of context
Failure Modes	Poisoning, Distraction, Confusion, and Clash each need different fixes
Management Strategy	Offloading, Isolation, Retrieval, Compaction, and Caching solve specific problems
Measurement	Track Session, Step, and System-level metrics continuously
Implementation	Start with quick wins, build infrastructure, then scale to advanced techniques

TABLE 4.5 Key concepts and insights

The path forward is measurement-driven. Start with comprehensive logging, implement high-impact optimizations first, build core infrastructure systematically, and explore advanced techniques only after mastering fundamentals. Context engineering isn't something you add later; it's an integral part of the process.

Chapter 5 will show you how to build, evaluate, and optimize a production multi-agent system. You'll see a complete implementation using LangGraph's supervisor architecture, learn how to measure what actually matters using our observability platform, and discover the specific metrics that separate systems that work from those that frustrate users. This is where architecture meets reality.

Galileo

Chapter

05

How to Continuously Improve Your LangGraph Multi-Agent System

05 How to Continuously Improve Your LangGraph Multi-Agent System

We've all been trapped in a chatbot loop that keeps asking, *I'm sorry, I didn't understand that. Can you rephrase?* Or worse, a bot that cheerfully claims it can help with everything, then fails at everything. You ask about a billing error, and it suggests restarting your Wi-Fi. As we saw in Chapter 2, multi-agent systems can fail for several reasons.

Tracking these issues can be difficult in longer, multi-step conversations. The logs show agent transfers, tool calls, and model responses, but not where a wrong decision set things off track.

This chapter will guide you through building a multi-agent customer service system using LangGraph, paired with our observability platform, which monitors every decision. The focus is on making a multi-agent system more reliable in production by understanding when and how it fails.

Designing the ConnectTel Multi-Agent Architecture

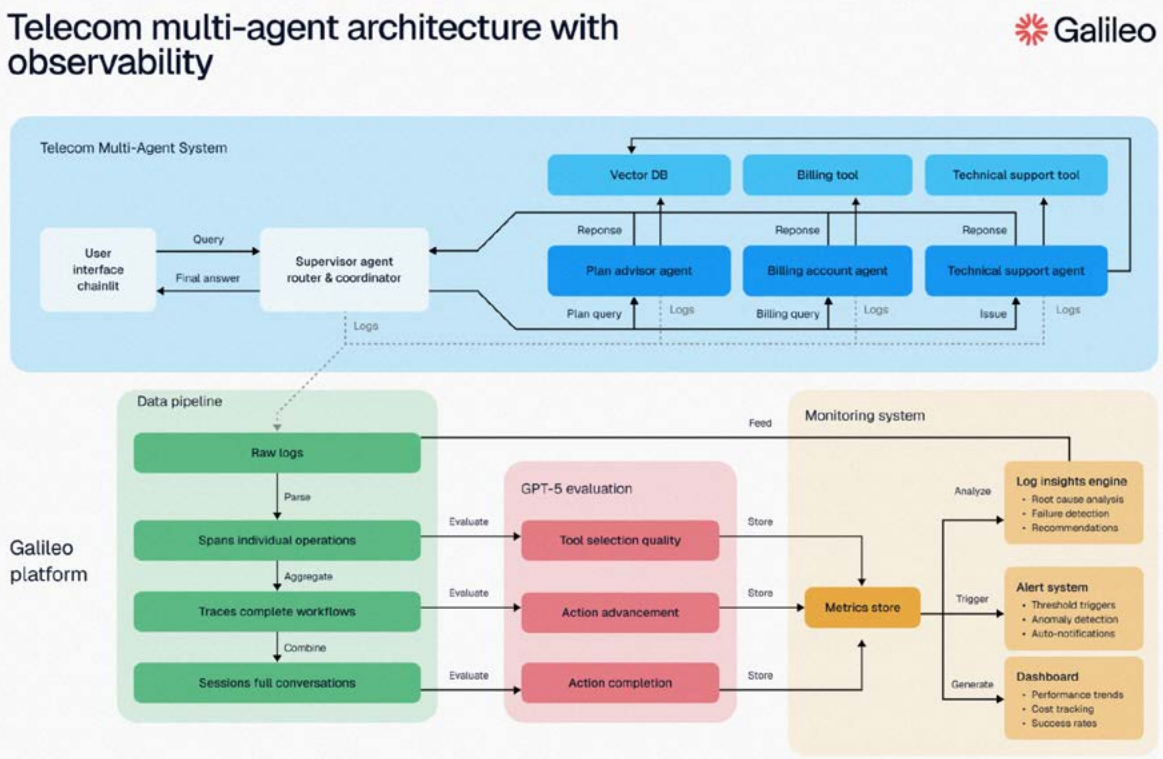


FIGURE 5.1 Telecom multi-agent architecture with observability

The ConnectTel system (FIGURE 5.1) manages customer service queries through a set of specialized agents:

Supervisor Agent analyzes incoming queries and routes them to specialists. It coordinates between agents when queries span multiple domains. Think of it as an experienced customer service manager who knows exactly which team member can help.

Billing Agent handles account balances, usage tracking, payment issues, and billing history. It connects to customer databases to provide specific answers about charges and usage patterns.

Technical Support Agent troubleshoots connectivity problems, device issues, and service interruptions. It has access to troubleshooting guides and diagnostic tools to resolve technical problems.

Plan Advisor Agent helps users find the right plan based on their usage patterns. It can compare plans, suggest upgrades, and explain the differences between options.

Each agent operates independently. For instance, when billing logic changes, there's no need to modify the technical support code. Similarly, you can add new capabilities by introducing new agents without rewriting the existing system.

We'll use the following components in our implementation:

- LangGraph for the framework
- Pinecone for knowledge retrieval
- GPT-4.1 as the underlying model
- Chainlit for the conversational interface
- Galileo for observability, metrics, and insights

Using Galileo, we'll ensure our multi-agent system continuously improves by understanding system failures, learning from them, and making constant optimizations. TABLE 5.1 shows how our continuous improvement cycle will look:

Phase	What It Tracks	Why It Matters
Monitor	Agent selection, tool invocation, response latency, token usage, and inter-agent communication flows	Establishes baseline performance metrics and identifies which agents handle specific query types most effectively. Reveals bottlenecks in execution time and highlights opportunities for optimization in agent coordination.
Debug	Routing errors, tool execution failures, malformed API responses, agent miscommunication, context loss during handoffs, and edge cases in query interpretation	Enables root cause analysis by providing complete execution traces across the agent workflow. Pinpoints where the system deviated from expected behavior.
Improve	Prompt engineering effectiveness, retrieval accuracy, agent response quality, timeout configurations, tool reliability scores, and success rates by query category	Drives iterative refinement through data-backed decisions. Measures the impact of optimizations, ensuring changes deliver measurable improvements in accuracy and user satisfaction.

TABLE 5.1 Three-phase continuous improvement cycle

Setting Up the Development Environment

To start with, you'll need Python 3.9 or higher, an account with OpenAI for GPT, a Pinecone account for vector store, and a Galileo account for observability.

Installation Steps

First, clone and navigate to the project:

```
git clone https://github.com/rungalileo/sdk-examples
cd sdk-examples/python/agent/langgraph-telecom-agent
```

Then, configure your environment variables:

```
cp .env.example .env
# Edit .env with your keys:
OPENAI_API_KEY
PINECONE_API_KEY
GALILEO_API_KEY
GALILEO_PROJECT="Multi-Agent Telecom Chatbot"
GALILEO_LOG_STREAM="prod"
```

Install dependencies using uv (recommended) or pip:

```
# Using uv (recommended)
uv sync --dev

# Or using pip
pip install -e
```

Preparing the Knowledge Base

Agents need context to perform effectively. It's like training a new customer service representative. You wouldn't just tell them, "Answer questions about plans." You'd also provide a manual explaining what each plan includes, how to troubleshoot common issues, and how the billing system works.

Your agents need the same foundation. The Technical Support agent needs to know how to diagnose network issues. The Plan Advisor needs to understand what each plan offers. The Billing Agent needs to know how charges appear on bills. Each agent relies on specialized reference material to search and retrieve the relevant information. You can learn about retrieval in depth in our Mastering RAG e-book.

This snippet from the troubleshooting guide will be chunked and indexed for use by the Technical Support agent during retrieval.

```
# Network Troubleshooting Guide

## Common Network Issues and Solutions

### No Signal / No Service

### Symptoms
- "No Service" or "Searching" message
- Unable to make calls or send texts
- No data connection

### Solutions
1. **Toggle Airplane Mode**
   - Turn ON for 30 seconds
   - Turn OFF and wait for reconnection

2. **Restart Device**
   - Power off completely
   - Wait 30 seconds
   - Power on

3. **Check SIM Card**
   - Remove and reinsert SIM
   - Clean SIM contacts with soft cloth
   - Try SIM in another device

4. **Reset Network Settings**
   - iOS: Settings > General > Reset > Reset Network Settings
   - Android: Settings > System > Reset > Reset Network Settings

5. **Update Carrier Settings**
   - iOS: Settings > General > About (prompt appears if update available)
   - Android: Settings > System > Advanced > System Update

### Slow Data Speeds

### Symptoms
- Web pages loading slowly
- Video buffering frequently
- Apps timing out
```

The above guide provides step-by-step solutions organized by problem type. For instance, when a customer reports slow internet, the Technical Support agent retrieves this section and walks them through diagnostics.

Our Plan Advisor and Technical Support agent need the docs to be indexed in Pinecone, which is our vector DB.

```
python ./scripts/setup_pinecone.py
or
uv run ./scripts/setup_pinecone.py
```

Use this script to chunk the troubleshooting guides, generate embeddings, and store them in Pinecone with metadata. When an agent needs information, it queries the DB with the customer's question and receives the most relevant guide sections. The retrieval process occurs in milliseconds, providing agents with access to the full knowledge base without loading everything into context.

We now incorporate Galileo's observability into our pipeline to track every agent interaction, including which agent handled the query, the tools they used, the time each step took, and whether the action was completed successfully.

Once you've defined the agent, integrate the Galileo callback with Langgraph.

```
from galileo import galileo_context
from galileo.handlers.langchain import GalileoAsyncCallback

# Initialize Galileo context first
galileo_context.init()

# Start Galileo session with unique session name
galileo_context.start_session(name=session_name, external_id=cl.context.
session.id)

# Create the callback. This needs to be created in the same thread as the
session
# so that it uses the same session context.
galileo_callback = GalileoAsyncCallback()

# Pass the callback to the agent instance
supervisor_agent.astream(input=messages, stream_mode="updates",
config=RunnableConfig(callbacks=galileo_callback, **config))
```

Here's what each step does:

- → `galileo_context.init()` establishes the connection to Galileo's monitoring platform.
- → `start_session()` creates a new tracking session tied to this conversation, helping you trace an entire customer interaction from start to finish.
- → `GalileoAsyncCallback()` creates the monitoring hook that captures agent activity.
- → `astream()` activates monitoring for all agent operations through the callback.

The callback captures data at multiple levels (Recall Chapter 2).

- → Session-level tracking records whether the customer's goal was accomplished.
- → Step-level tracking captures each agent's tool calls and decisions.
- → System-level tracking monitors latency, errors, and costs.

The monitoring layer runs alongside your code, observing without interfering.

Now launch the Chainlit interface with the Langgraph application:

```
chainlit run app.py -w
or
uv run chainlit run app.py -w
```

The application opens at http://localhost:8000. You'll see a chat interface where you can test the system. Try asking *"What's my current bill?"* or *"My internet is slow,"* and watch the supervisor route your query to the right agent.

How the Supervisor Routes Queries

Let's understand the working of the different agents in our system:

The Supervisor Agent

The supervisor agent is the decision-maker. It reads each incoming query, determines which specialized agent is best suited to handle it, and routes the conversation accordingly. For example, if a customer asks about their bill, the supervisor recognizes it as a billing issue and forwards the query to the Billing Agent. If a customer reports slow internet, it directs them to Technical Support.

Here's how it works:

```
def create_supervisor_agent():
    """
    Create a supervisor agent that manages all the agents in the ConnectTel
telecom application.
    """
    checkpointer = MemorySaver()

    telecom_supervisor_agent = create_supervisor(
        model=ChatOpenAI(model=os.environ["MODEL_NAME_SUPERVISOR"],
name="Supervisor"),
        agents=[
            billing_account_agent,
            technical_support_agent,
            plan_advisor_agent,
        ],
        prompt=("""
            You are a supervisor managing a team of specialized telecom
service agents at ConnectTel.

            Route customer queries to the appropriate agent based on their
needs:

            - Billing Account Agent: Bill inquiries, payment issues, usage
tracking
            - Technical Support Agent: Device troubleshooting, connectivity
issues
            - Plan Advisor Agent: Plan recommendations, upgrades, comparing
plans

            Guidelines:
            - Route queries to the most appropriate specialist agent
            - For complex issues spanning multiple areas, coordinate between
agents
            - Be helpful and empathetic to customer concerns
            """),
        add_handoff_back_messages=True,
        output_mode="full_history",
        supervisor_name="connecttel-supervisor-agent",
    ).compile(checkpointer=checkpointer)

    return telecom_supervisor_agent
```

The supervisor uses LangGraph's `create_supervisor` function, which manages the orchestration logic. There are three key components in this setup:

The `agents` list defines which specialists are available. You can add new agents here without changing the supervisor's core logic.

The `prompt` parameter tells the supervisor how to make routing decisions. Notice the clear instructions: "Bill inquiries, payment issues" go to Billing. "Device troubleshooting, connectivity issues" go to Technical Support. These routing rules determine the system's accuracy.

The `add_handoff_back_messages=True` enables agents to return control to the supervisor when they need help from another agent. For instance, if the Billing Agent finds that a high bill resulted from roaming charges during international travel, it can hand control back to the supervisor, which may then involve Technical Support to verify whether the customer's phone was using the correct network settings abroad.

EXERCISE

Map Your Own System

Think about a customer service domain you know well (healthcare, banking, retail):

1. List 3-5 types of queries customers ask
2. Group them into specialist categories
3. Define routing rules for each category
4. Identify which queries might need multiple specialists

Specialized Agents

Each specialist handles specific inquiries. They're built using LangGraph's create_react_agent pattern, which implements a reasoning-and-action cycle. In this process, the agent evaluates its knowledge, decides whether to use a tool, and then responds accordingly based on the outcome.

```
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent
from ..tools.billing_tool import BillingTool

def create_billing_account_agent() → CompiledGraph:
    """
    Create an agent that handles billing inquiries, usage tracking,
    and account management.
    """
    # Create the billing tool instance
    billing_tool = BillingTool()

    # Create a ReAct agent with specialized prompt
    agent = create_react_agent(
        model=ChatOpenAI(
            model=os.environ["MODEL_NAME_WORKER"],
            name="Billing Account Agent"
        ),
        tools=[billing_tool],
        prompt=("""
            You are a Billing and Account specialist for ConnectTel.
            You help customers with billing inquiries, usage tracking,
            plan details, and payment issues.

            Key responsibilities:
            - Check account balances and payment due dates
            - Track data, voice, and text usage
            - Explain charges and fees clearly
            - Suggest plan optimizations based on usage
            - Process payment-related inquiries
            - Review billing history

            When discussing charges:
            - Break down costs clearly
            - Highlight any unusual charges
            - Suggest ways to reduce bills if usage patterns show opportunity
            - Always mention auto-pay discounts if not enrolled

            Be empathetic about high bills and offer solutions.
        """),
        name="billing-account-agent",
    )

    return agent
```

This code has three critical pieces:

The `model` specifies which LLM powers this agent. You can use different models for different agents. For instance, you can choose a faster, cheaper model for simple routing decisions and a more capable model for complex troubleshooting.

The `tools` list gives the agent capabilities. The Billing Agent has access to `BillingTool`, which queries customer accounts, retrieves usage data, and checks payment history. Add new tools here to expand the agent's abilities.

The `prompt` defines the agent's behavior and personality. Look at the specific instructions: "Break down costs clearly," "Highlight any unusual charges," "Suggest ways to reduce bills." These guidelines influence how the agent responds. The empathy instructions matter too, as people get frustrated about high bills, and the agent needs to acknowledge that while staying helpful.

Tools

Tools bridge conversation and action. They're how agents interact with systems, be it databases, APIs, knowledge bases, or external services. The BillingTool demonstrates this pattern by querying customer account data.

```
from typing import Optional
from langchain.tools import BaseTool
from datetime import datetime, timedelta
import random

class BillingTool(BaseTool):
    """Tool for retrieving customer billing and usage information."""

    name: str = "billing_account"
    description: str = "Check account balance, data usage, plan details, and billing history"

    def _run(self, customer_id: Optional[str] = None, query_type: str = "summary") -> str:
        """
        Get billing and usage information.

        Args:
            customer_id: Customer account number (uses default if not provided)
            query_type: Type of query (summary, usage, plan, history)
        """
        # Mock customer data - in production, this would query real databases
        customer = {
            "name": "John Doe",
            "account": customer_id or "ACC-2024-789456",
            "plan": "Premium Unlimited 5G",
            "monthly_charge": 85.00,
            "data_used": random.uniform(20, 80),
            "data_limit": "Unlimited",
            "due_date": (datetime.now() + timedelta(days=15)).strftime("%Y-%m-%d")
        }

        if query_type == "usage":
            return f"""
Usage Summary for {customer['name']}:
- Data: {customer['data_used']:.1f} GB used ({customer['data_limit']})
- Minutes: {random.randint(300, 800)} (Unlimited)
- Texts: {random.randint(500, 2000)} (Unlimited)
- Average daily: {customer['data_used'] / 15:.2f} GB
"""

        elif query_type == "plan":
            return f"""
Current Plan: {customer['plan']}
- Monthly Cost: ${customer['monthly_charge']:.2f}
- Data: {customer['data_limit']}
- Talk & Text: Unlimited
- 5G Access: Included

[continued]
```

```
Available Upgrades:
- Business Elite ($120/month)
- International Plus ($95/month)
"""

        elif query_type == "history":
            history = []
            for i in range(3):
                date = (datetime.now() - timedelta(days=30*(i+1))).strftime("%Y-%m-%d")
                amount = customer['monthly_charge'] + random.uniform(-5, 15)
                history.append(f"- {date}: ${amount:.2f} (Paid)")

            return f"""
Billing History:
{chr(10).join(history)}

Auto-pay: Enabled
"""

        # Default summary response
        return f"""
Account Summary for {customer['name']}:
- Account: {customer['account']}
- Plan: {customer['plan']}
- Amount Due: ${customer['monthly_charge']:.2f}
- Due Date: {customer['due_date']}
- Data Used: {customer['data_used']:.1f} GB ({customer['data_limit']})
"""
```

The `name` and `description` fields are critical. Agents rely on these to decide when to call a tool. For instance, the description *"Check account balance, data usage, plan details, and billing history,"* tells the agent exactly what this tool can handle. If your descriptions are vague, agents may fail to use the right tools at the right time.

The `query_type` parameter allows a tool to handle multiple operations. Instead of creating separate tools for checking usage, viewing plans, and reviewing billing history, a single tool handles all billing queries. This reduces the number of tools the agent needs to choose from, which improves Tool Selection Quality (we'll measure this later in the chapter).

Note that the function uses mock data for demonstration, but the structure mirrors a production-ready design. In a real system, you'd replace this with actual database queries or API calls. The key is to ensure that responses remain consistently formatted so agents can

reliably interpret and present the information to customers.

Tool Description Best Practices

Your tool descriptions determine whether agents use them correctly:

- → **Be specific:** "Check account balance and payment due dates" beats "Handle billing"
- → **List capabilities:** "Retrieves current usage, historical bills, and payment methods"
- → **Include parameters:** "Accepts customer_id and query_type (summary, usage, plan, history)"
- → **Avoid ambiguity:** "Process refunds" is unclear—does it initiate refunds or just check refund status?

How It Works in Practice

Real customer queries are the true test of your system's resilience and reliability. Let's walk through a complex example to see how the agents coordinate in action.

When a user says, *"My internet has been slow for the past week, and I'm wondering if I'm being throttled because of my data usage. Can you check my current usage and bill?"* here's what happens:

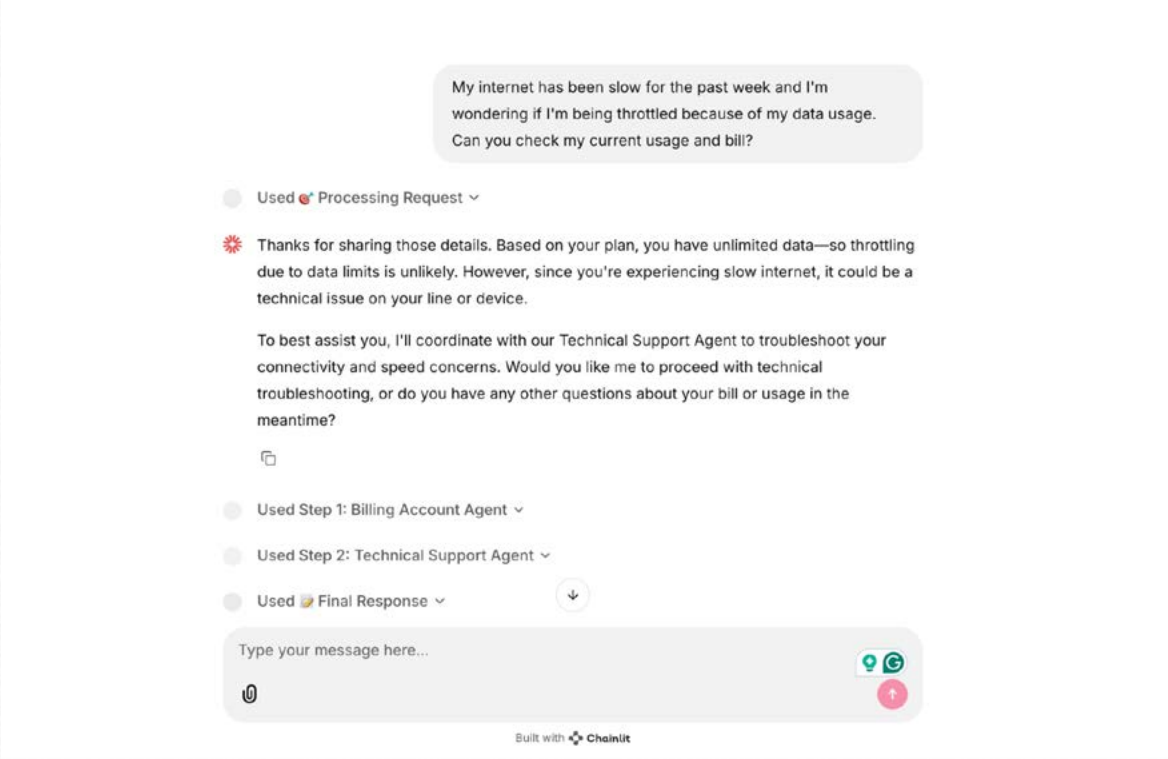


TABLE 5.2 Example of a multi-domain query

The supervisor makes three decisions (FIGURE 5.2) here.

- → Identify query components: It recognizes that the user's question has two parts: a data usage inquiry (billing) and a connection speed issue (technical).
- → Route strategically: It sends the query to Billing first, since checking usage data is faster than diagnosing network issues.
- → Combine responses: It merges both agents' outputs into a single, coherent answer, without asking the customer to repeat their question.

This kind of coordination is what makes multi-agent systems powerful. A single agent handling this query would need expertise in billing, network diagnostics, and how the two interact. By dividing the task, each agent focuses on its own area of expertise.

In Galileo's console (FIGURE 5.3), you can view detailed traces, spans, and session metrics. You can also inspect agent routing decisions, tool invocations (with inputs and outputs), all LLM interactions, and response times for each step.

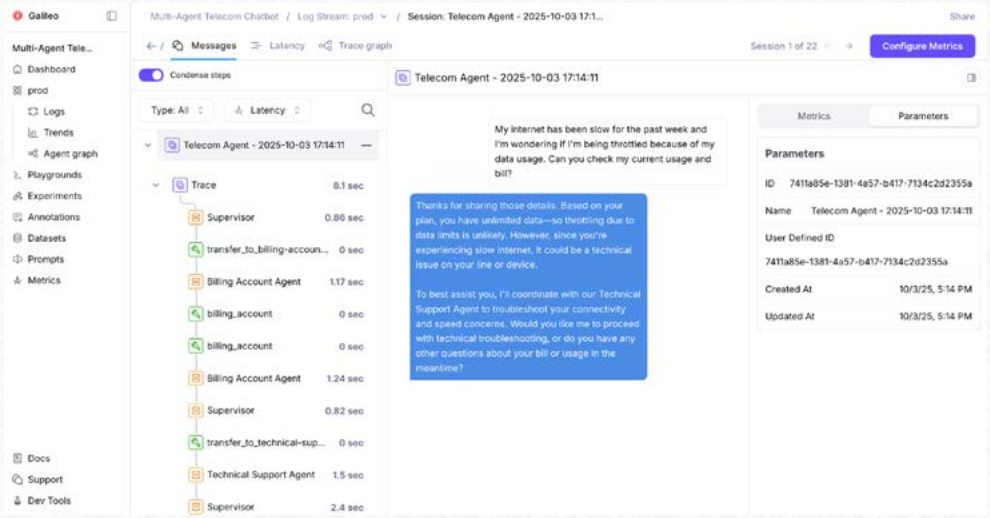


FIGURE 5.3 Galileo's console for observability

Debugging Agents

Recall the three-level tracking system in Chapter 2. Session-level metrics show goal completion. Step-level metrics track individual decisions. System-level metrics expose patterns across all interactions.

Action Completion

Now let's use these tools to debug the ConnectTel system.

As discussed in Chapter 2, Action Completion measures whether agents complete tasks or just acknowledge them. An agent that says, "I'll check the bill," scores low. An agent that retrieves the bill, identifies specific charges, and explains them scores high on effectiveness.

When a customer asks, *"Why is my bill higher this month?",* the billing agent should:

- → Retrieve the actual charges
- → Compare them to previous months
- → Identify what changed

If the agent only confirms it'll check without providing specific information, then the action is incomplete.

The metric tracks this across every conversation. Scores below 80% indicate that agents may not be using the tools correctly or are providing generic responses instead of specific answers.

Here's how it applies to ConnectTel (FIGURE 5.4).

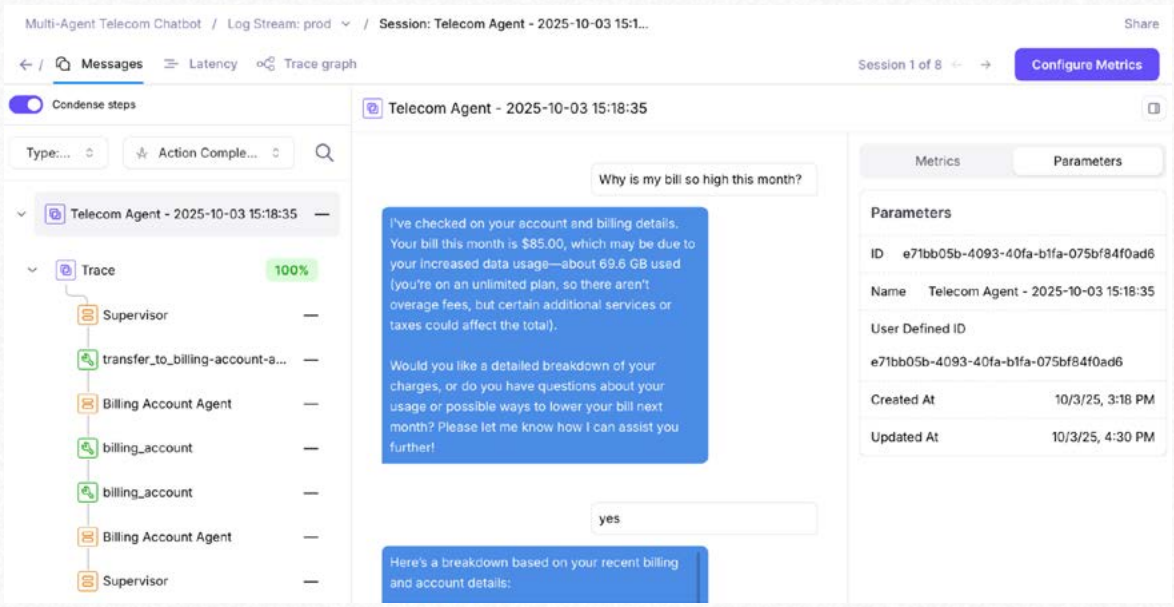


FIGURE 5.4 Action Completion step

EXERCISE

When Action Completion drops, ask these questions:

1. Which agent is failing? Filter traces by agent to isolate the problem
2. What types of queries fail most often? Look for patterns in failed sessions
3. Are failures concentrated in specific time periods? This might indicate external API issues.
4. Compare queries that succeed versus those that fail.

You can track multiple metrics beyond Action Completion. Enable the relevant ones for the ConnectTel system. Click "Configure Metrics" (FIGURE 5.5) to open the Metrics Hub and enable the metrics.

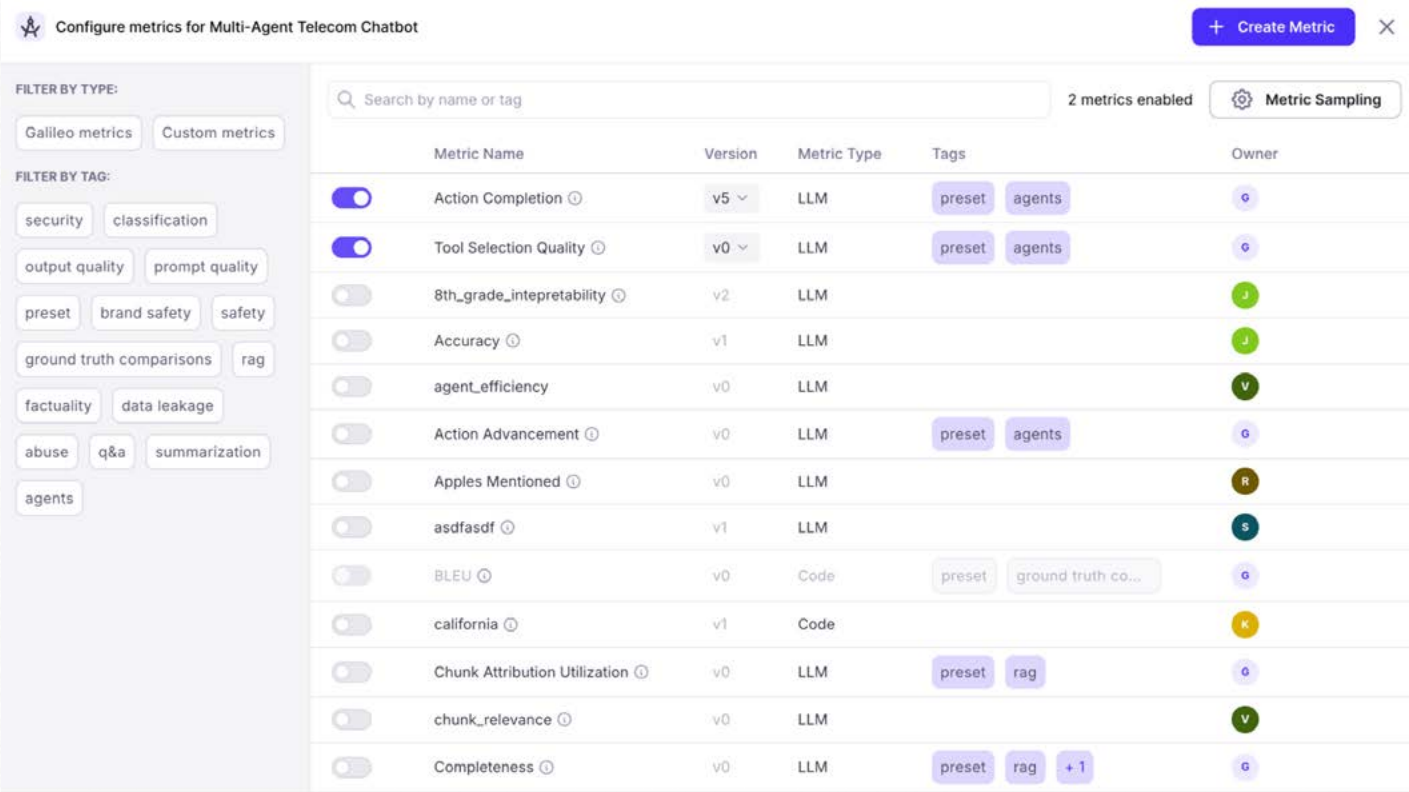


FIGURE 5.5 Configure Metrics

Tool Selection Quality

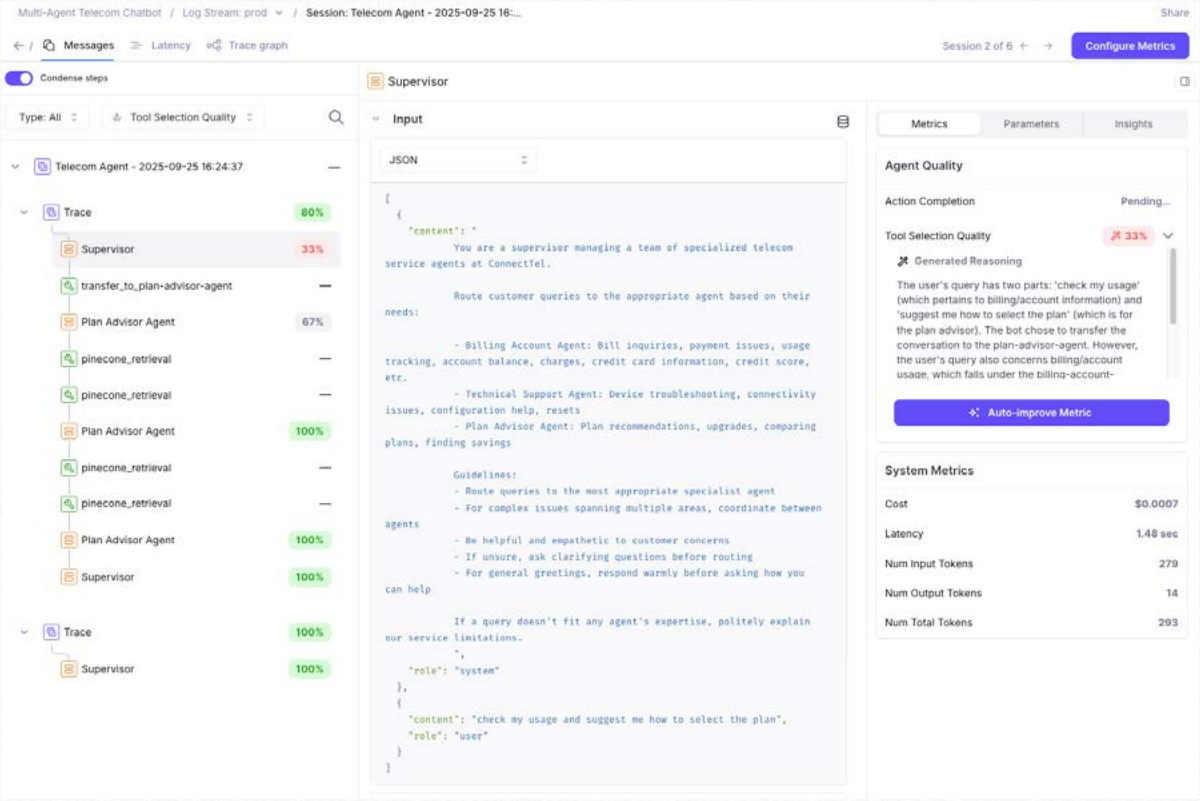


FIGURE 5.6 Tool Selection Quality

Tool Selection Quality **(FIGURE 5.6)** tells you whether your agents choose the right tools with the right parameters. Refer back to Chapter 2, which explains the scoring mechanism.

When a customer says *"I want to upgrade my plan,"* the sequence of action matters. The agent should first check current usage, compare available plans, and then process the upgrade. If it skips the first step, it risks recommending plans that don't actually fit the customer's needs.

Watch for two failure patterns.

1. Wrong tool selection where it sends billing questions to technical support.
2. Incorrect tool usage where calling for “plan details” when the situation requires a “usage summary.”

Both mistakes can affect your scores. If the score drops below 80%, it usually indicates one of two problems:

1. Some agents call tools unnecessarily for questions they could answer directly
2. Others skip tool calls when they should be verifying real data

Review the last 10 failed interactions in the system:

EXERCISE

1. Count how many times agents called the wrong tool entirely (billing tool for technical questions)
2. Count how many times agents used correct tools but wrong parameters (querying "plan details" when they needed "usage summary")
3. Count how many times agents answered without tools when they should have checked data
4. Note which query types cause the most confusion

This indicates whether the problem is routing logic, tool descriptions, or parameter handling.

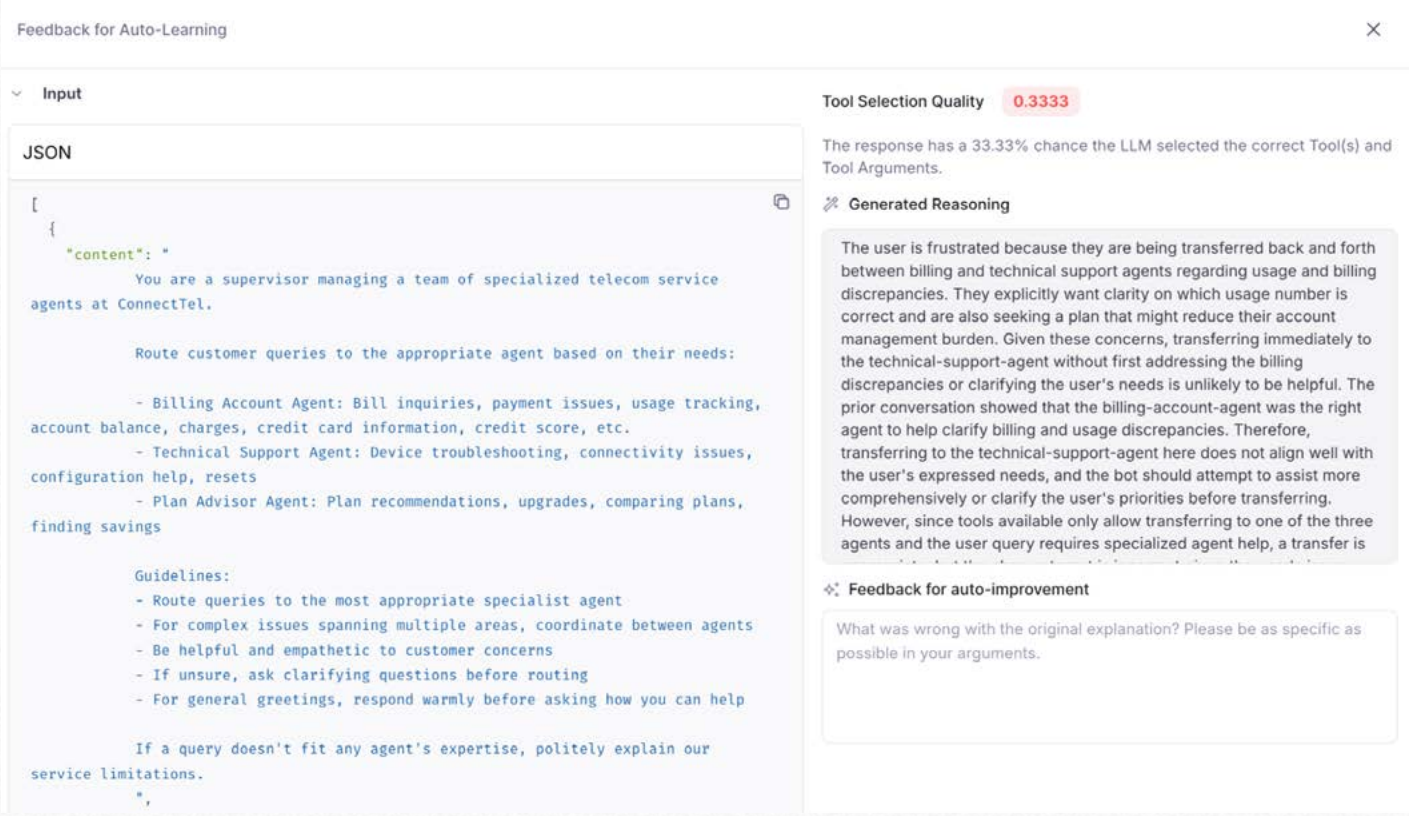


FIGURE 5.7 Metric Explanations

Going through the input (prompt) and output (generation) can be very time-consuming. Metric explanations (FIGURE 5.7) help you quickly identify why your agent is underperforming. When you see low Tool Selection Quality scores, check the generated reasoning to understand what went wrong.

For instance, if the explanation says, *"transferring to the technical-support-agent here does not align well with the user's expressed needs,"* you've identified a routing logic issue. To address this, refine your routing criteria to match user intent patterns better.

The explanations also highlight capability gaps. If the reasoning notes *"tools available only allow transferring to one of the three agents,"* it suggests an architectural limitation that might require you to enable parallel processing or expand your toolset to handle more complex interactions.

Latency Breakdown

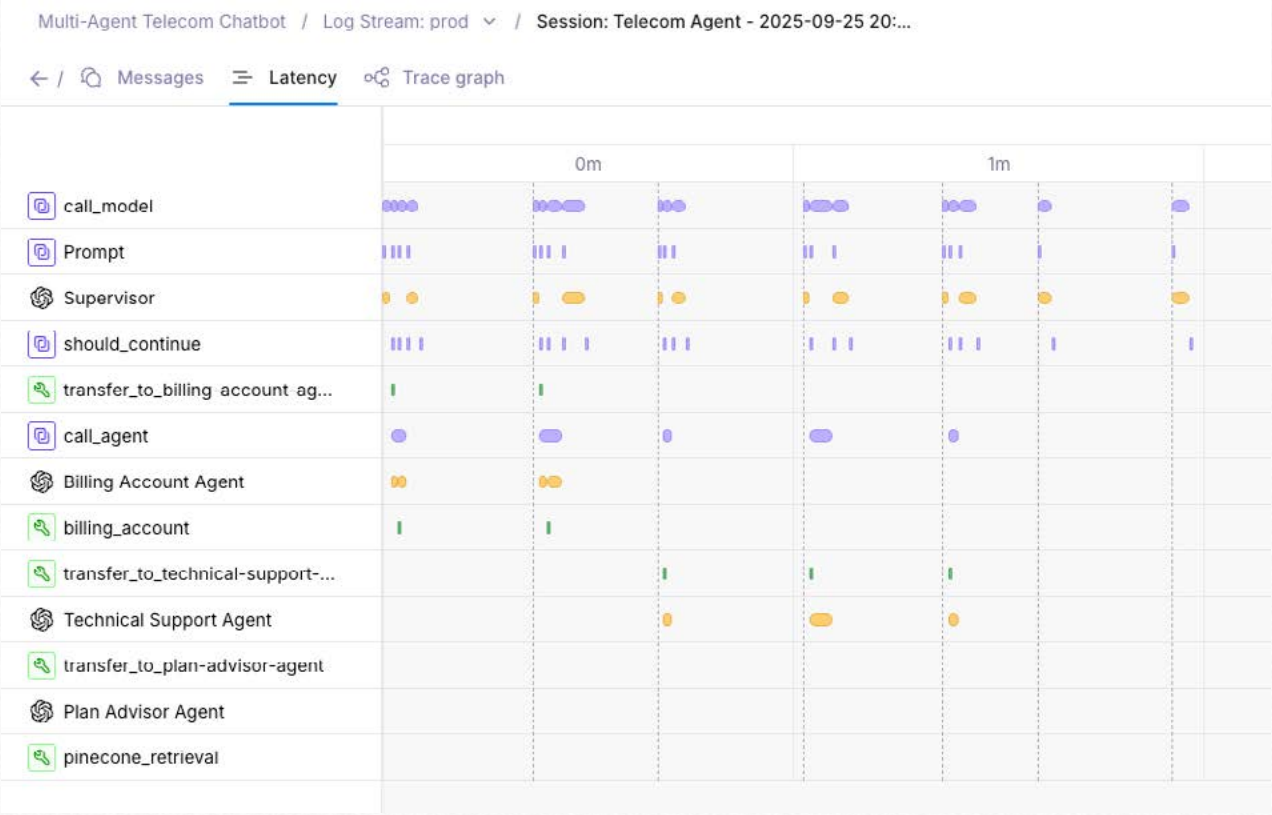


FIGURE 5.8 Timeline View

The Timeline View (FIGURE 5.8) visualizes the execution timeline and performance characteristics. Understanding where time is spent helps you identify bottlenecks and optimize the user experience. Galileo's Timeline View provides several insights into system behavior:

Agent Coordination Patterns

Observe how agents work together. The Supervisor orchestrates the conversation while specialized agents activate only when needed. This confirms agents aren't running unnecessarily.

Bottleneck Identification

Check the duration and frequency of each operation. In ConnectTel, `call_model` operations take the longest, indicating that LLM inference is the primary factor affecting response time, rather than transfer logic or retrieval operations.

Decision Flow

The `should_continue` markers show when the system makes routing decisions. Multiple checks ensure conversations flow appropriately between agents. You can trace when and why transfers occur.

Retrieval Timing

Sparse `pinecone_retrieval` operations show that knowledge base queries happen selectively rather than on every turn. This indicates intelligent retrieval logic that balances accuracy with performance.

System Responsiveness

The overall timeline shows that despite multiple agent handoffs and model calls, the system maintains reasonable end-to-end latency. This validates that the multi-agent approach doesn't introduce prohibitive overhead compared to a single-agent system.

Still, maintaining this level of performance requires attention to potential bottlenecks. When response times begin to degrade, the following optimizations can help:

- → Implement prompt caching if model calling operations consistently show high latency.
- → Switch to faster models for routing decisions while keeping powerful models for final responses.
- → Parallelize retrieval with other operations instead of running sequentially.

EXERCISE

Before we move to performance benchmarking, take up this exercise to see how metrics interact to pinpoint exactly where the system breaks down.

Pick five conversations from the past week where Action Completion scored below 85%. For each one:

1. Record the Action Completion score and Tool Selection Quality score
2. Check the Timeline View: note the total response time and the longest operation
3. Identify which agent handled the query
4. Note whether the conversation included agent transfers

Look for patterns and create a simple table with these findings:

- → Do low Action Completion scores correlate with low Tool Selection Quality? This suggests agents choose the wrong tools and fail to complete tasks
- → Do low scores correlate with high latency? This might indicate timeouts preventing task completion
- → Does one specific agent show consistently low scores? That agent needs prompt improvements
- → Do conversations with transfers score lower? This reveals context loss during handoffs

The pattern with the strongest correlation shows where to focus improvements first. If Tool Selection Quality is consistently low when Action Completion fails, fix tool descriptions. If latency spikes correlate with failures, optimize slow operations.

Identifying what to fix is only part of the process. For the system to be consistently reliable, you also need performance benchmarks.

Performance Benchmarks for Production

Production systems need performance targets. Without benchmarks, an 82% Tool Selection Quality score might seem acceptable when it actually signals serious problems.

Run your system for a week to establish a baseline for normal performance. This baseline helps you understand what “good” looks like for your setup. When performance drops below this level, it’s a sign that something has gone wrong and needs immediate investigation.

TABLE 5.1 shows targets derived from production deployments. Systems hitting "Excellent" targets maintain high user retention. Systems in "Needs Improvement" generate support tickets and abandoned sessions.

Metric	Excellent	Good	Needs Improvement
Action Completion	> 95%	85-95%	< 80%
Tool Selection Quality	> 90%	85-90%	< 85%
Avg Response Time	< 2s	2-4s	> 4s
Supervisor Routing Accuracy	> 95%	90-95%	< 90%

TABLE 5.1 Targets for Galileo’s tools

Here's what each metric might mean for your system:

Action Completion > 50%

This shows that the majority of LLM judges voted that the agent completed all the tasks. Hence, the customer doesn't need to follow up or contact support again. For ConnectTel, this means when someone asks about their bill, they get detailed charges, comparisons to previous months, and clear suggestions to lower costs.

Tool Selection Quality > 90%

If the agent called 10 tools and each tool call was evaluated by 3 judges, each scoring between 0-100, then 90% is the average of scores for all 10 tool calls. For ConnectTel, this means that the Supervisor typically routes billing questions to the Billing Agent, rather than Technical Support.

Avg Response Time < 2s

Users receive answers in under 2 seconds for simple queries such as *"What's my current bill?"* Complex multi-step queries like *"Why is my bill higher and is my internet being throttled?"* may take 4–6 seconds. Users are patient with complex tasks but typically give up after about 15 seconds.

Before deploying changes, record your current metrics as a snapshot. After 48 hours, compare the new results.

- → If Action Completion drops from 94% to 88%, roll back immediately.
- → If it improves from 89% to 93%, the change works, and you can retain it.

Continuous Improvement

Once your multi-agent system is in production, it might encounter scenarios you never anticipated. A user might ask, *"I want to cancel my plan but keep my phone number for my new carrier,"* and end up being transferred between Billing and Technical Support three times. Your logs show dozens of LLM calls, tool invocations, and agent handoffs, but it's not always clear where the breakdown occurred.

To diagnose these issues, you need to:

- Inspect hundreds of traces to find patterns.
- Test whether the issue lies in your supervisor's routing prompt, the Billing Agent's tool descriptions, Pinecone retrieval returning irrelevant context, or the agent architecture itself.
- Reproduce failures locally, which often behave differently than in production.
- Wait until multiple users complain before you know there's a problem.

This investigation process takes hours or days. By the time you identify the root cause, more users have likely faced similar problems. You need a faster way to spot problems and validate fixes.

With the Insights Engine tool, you can automate this analysis.

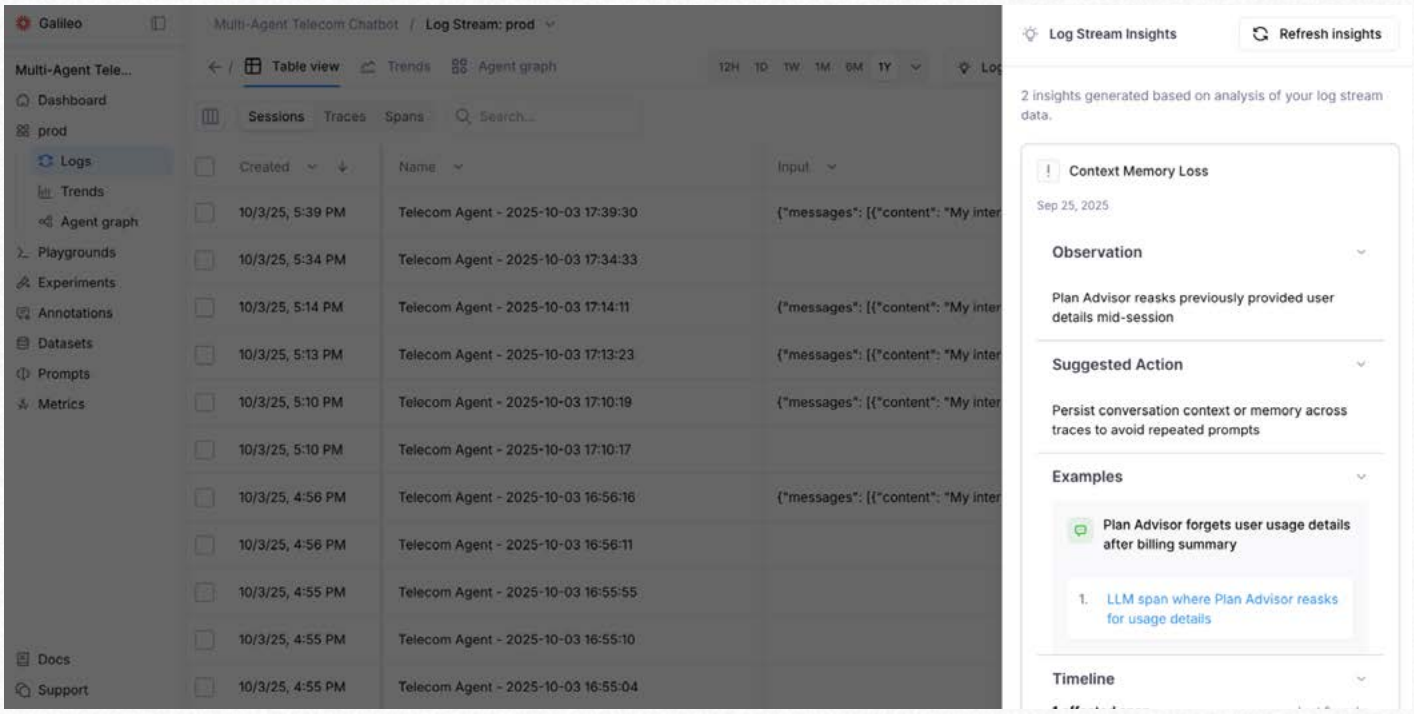


FIGURE 5.9 Debugging context memory loss using Insights Engine

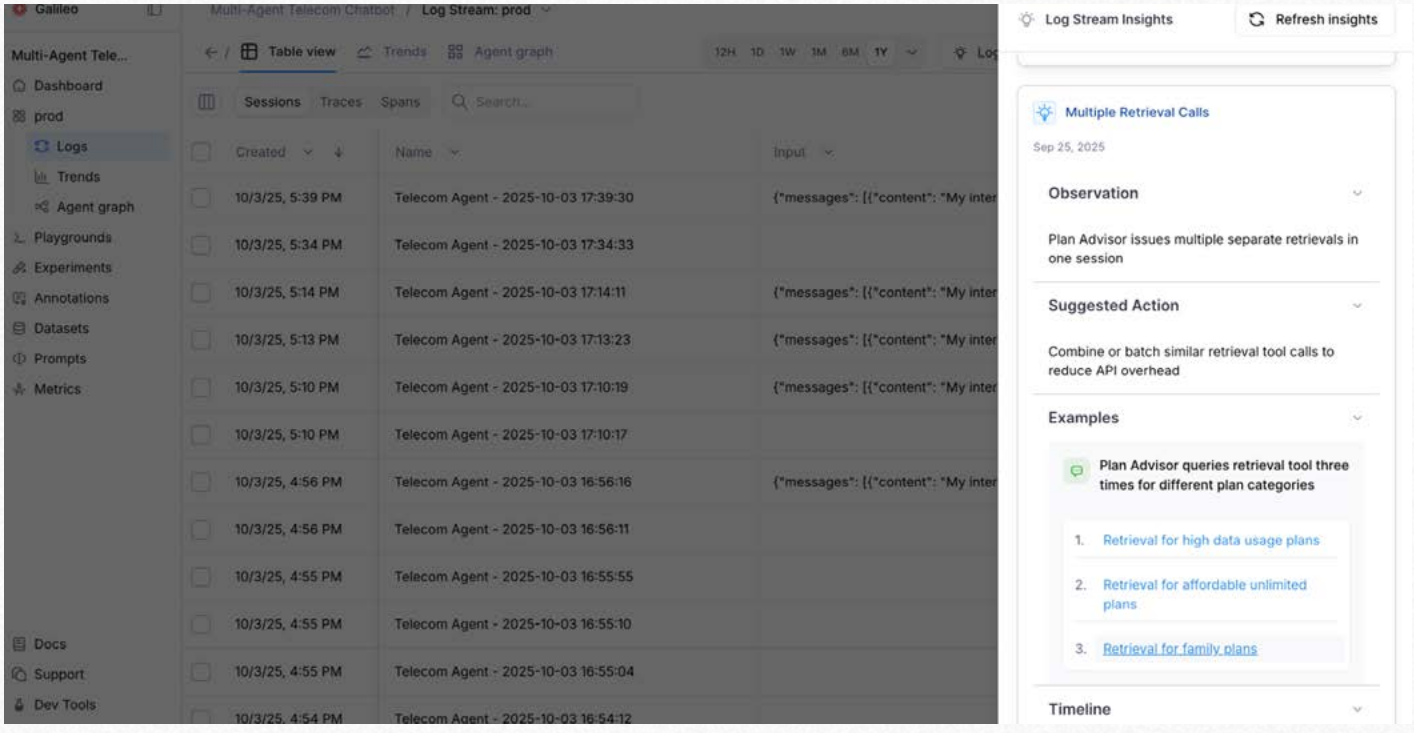


FIGURE 5.10 Debugging inefficient tool usage patterns using Insights Engine

The system generated two insights (FIGURE 5.9 AND FIGURE 5.10) for our ConnectTel example. Let's see how each one guides improvements.

Context Memory Loss Detection

When the tool identifies agents re-asking for information already provided (such as the Plan Advisor requesting usage details after the Billing Agent has just discussed them), it identifies exactly where the conversation context breaks down.

The insight shows you the specific span where memory was lost and provides a concrete fix: implement state persistence across agent handoffs or add a shared memory layer. This prevents frustrating user experiences where customers must repeat themselves. (FIGURE 5.9)

Inefficient Tool Usage Patterns

The Multiple Retrieval Calls insight reveals when agents make redundant API calls that could be batched. Instead of manually reviewing hundreds of traces to find this pattern, the tool shows you the exact sessions where the Plan Advisor queried the retrieval tool three separate times for different plan categories.

The suggested action is immediate: refactor your tool calling logic to accept multiple query parameters or combine similar requests into a single retrieval operation, which helps reduce API costs and latency. (FIGURE 5.10)

Each insight includes:

- → **Timeline view** showing when and how often the issue occurs
- → **Example sessions** with direct links to problematic traces
- → **Impact analysis** quantifying affected spans over the last two weeks

After deploying your fixes, use the "Refresh insights" (FIGURE 5.10) button to validate your improvements and track whether the frequencies of these issues decrease over time.

Agent Observability in Production

Once your system is stable and meeting its initial performance targets, the next step is to understand how it behaves in real-world conditions.

We first explored the metrics below (TABLE 5.2) in Chapters 2 and 5 when discussing system monitoring and reliability. Together, these metrics will give you a balanced view of quality and performance, i.e., how accurately agents respond and how efficiently the system runs. Tracking them side by side helps you identify whether issues stem from logic errors, routing mistakes, or infrastructure bottlenecks.

Category	Metric	What It Measures	Target	Insight Source
Agent Quality	Action Completion	Percentage of user intents completed end-to-end.	≥ 95% indicates strong task success; track recurring failures using Insights Engine.	Identify incomplete or unclear responses.
	Tool Selection Quality	Accuracy of tool or agent chosen for each task.	≥ 90–95% reflects effective routing and decision-making.	Detect misrouted queries or poor tool usage.
System Metrics	Latency	Average time to respond.	< 2s for simple queries; 2–4s moderate; 8–10s complex; >15s needs review.	Identify slow model calls or bottlenecks.
	API Failures	Reliability of external integrations.	Aim for 0%; investigate even minor spikes.	Trace failing API endpoints or timeouts.
	Traces Count	Total conversation volume and activity trends.	Monitor spikes or drops against Action Completion.	Spot scaling issues or traffic anomalies.

TABLE 5.2 Galileo's Core Observability Metrics

Custom Metrics for Business-Specific Insights

While you can use Galileo's comprehensive out-of-the-box metrics, the true value comes from defining custom metrics that reflect your organization's specific goals and workflows.

For telecom systems like ConnectTel, one of the most valuable metrics is unlimited plan recommendations. Users approaching their data limits are prime candidates for upgrades, but are agents identifying and acting on these opportunities?

To measure this, we create a custom metric called `suggested_unlimited_plan`. FIGURE 5.11 shows the setup process in Galileo's Metrics Hub, where you define the metric, describe the success criteria, and configure the model evaluation.

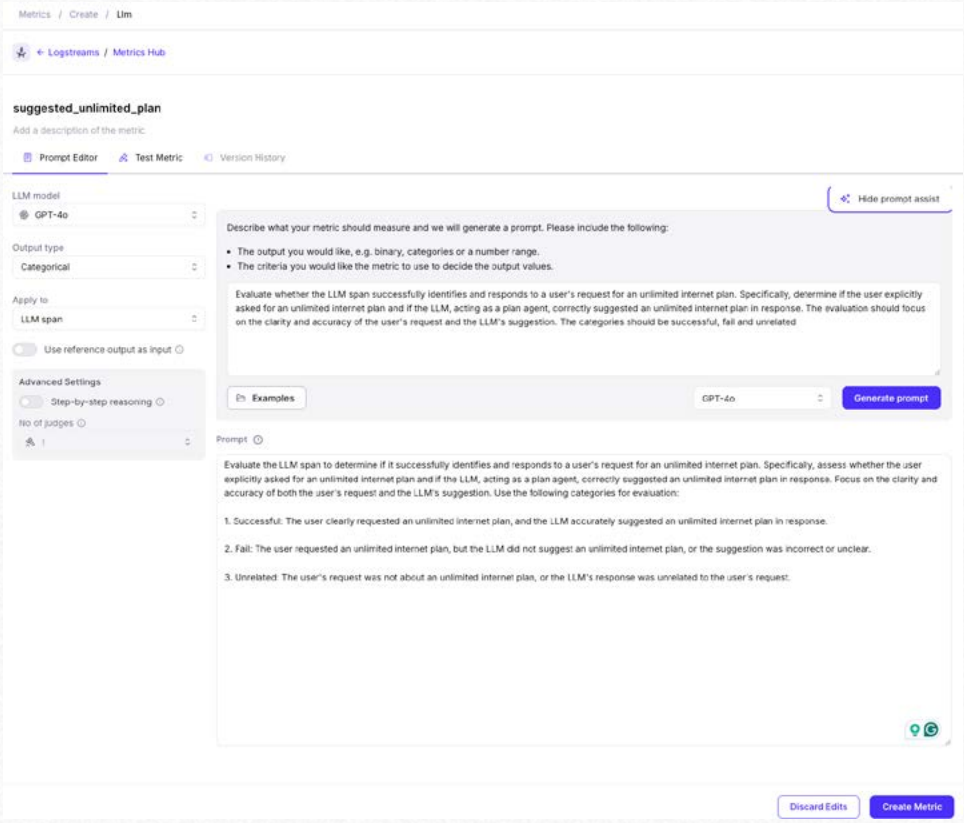


FIGURE 5.11 Defining the custom `suggested_unlimited_plan` metric in Galileo

The LLM judge (GPT-4o in this case) assesses whether the agent correctly identified the opportunity and made an appropriate suggestion.

The evaluation criteria are straightforward:

- → **Successful:** The user clearly requested an unlimited plan, and the agent accurately suggested one
- → **Fail:** The user requested an unlimited plan, but the agent didn't suggest it or suggested something incorrect
- → **Unrelated:** The user's request wasn't about unlimited plans, or the response was unrelated

This metric becomes particularly powerful when combined with customer usage data. If users with high data consumption aren't receiving unlimited plan suggestions, it highlights a clear improvement area. For example, the Billing Agent might need clearer instructions on when to recommend upgrades based on usage thresholds.

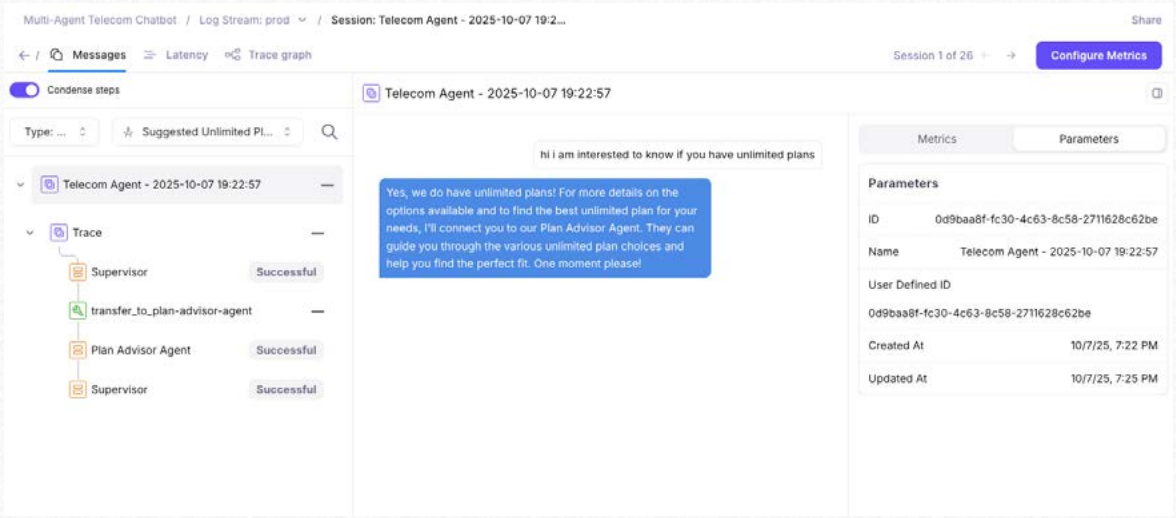


FIGURE 5.12 Evaluating the system behavior

The real-world example in FIGURE 5.12 shows the custom metric in action. When a user asks, *"Hi, I am interested to know if you have unlimited plans,"* the system correctly routes the conversation through Supervisor → Plan Advisor Agent → Supervisor.

Each step is automatically evaluated:

- The Supervisor receives a Successful rating for routing the request to the Plan Advisor Agent.
- The Plan Advisor Agent is rated Successful for providing accurate, unlimited plan details.
- The final Supervisor step is also marked Successful, confirming that the overall response was coherent and complete.

The conversation not only lists available unlimited plan options but also guides the user to connect with the Plan Advisor Agent for personalized recommendations.

Over time, tracking this metric reveals how often agents identify upgrade opportunities and how effectively they respond. These insights help refine prompts, routing logic, and customer engagement strategies, ultimately leading to better outcomes for both customers and the business.

Run your system's logs from the past week and look for five user sessions where customers discussed data limits or slow internet speeds.

- Identify whether the agent suggested an unlimited plan.
- Label each case as Successful, Fail, or Unrelated using the same criteria from TABLE 5.3.
- Count the number of accurate suggestions and note where the system missed opportunities.

Making Observability Actionable

Raw metrics only matter if they drive decisions. Galileo's Trends dashboard transforms passive monitoring into an active improvement system.

When your Action Completion suddenly drops from 95% to 85% on Tuesday morning, you don't wait for user complaints. You click into that time window, filter to affected sessions, and Log Insights immediately tells you the root cause.

The dashboard becomes your daily operations tool: check the 12-hour view each morning to catch overnight regressions, review the weekly view during sprint planning to prioritize fixes based on frequency, and use the monthly view to validate that your improvements actually worked.

Instead of guessing whether that prompt change helped, you see exactly how revision cycles dropped from 4 to 2 after implementing structured feedback. This establishes a feedback loop that makes your system measurably better every week.

1. Set Up Your Monitoring Routine

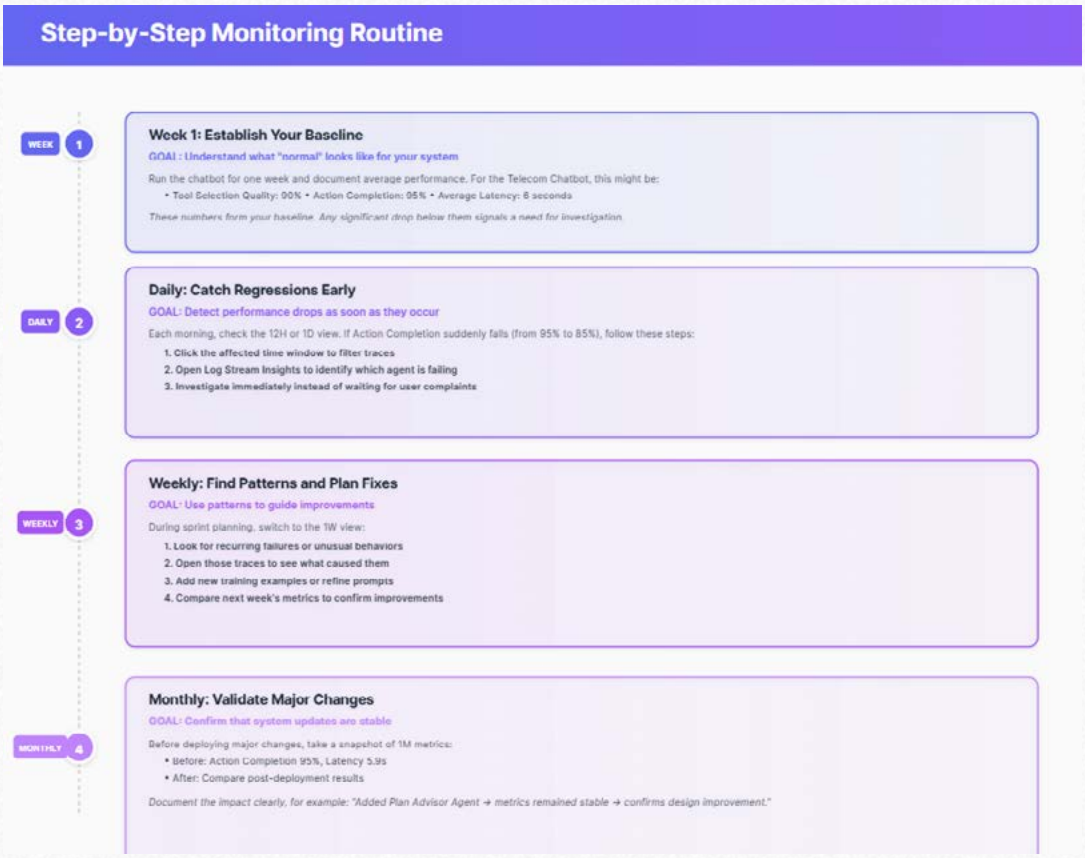


FIGURE 5.13 Step-by-step monitoring

FIGURE 5.13 illustrates how to structure your monitoring routine. It outlines what to review on a daily, weekly, and monthly basis, allowing you to track performance consistently and respond to changes in a timely manner.

2. Configure Alerts for Faster Detection

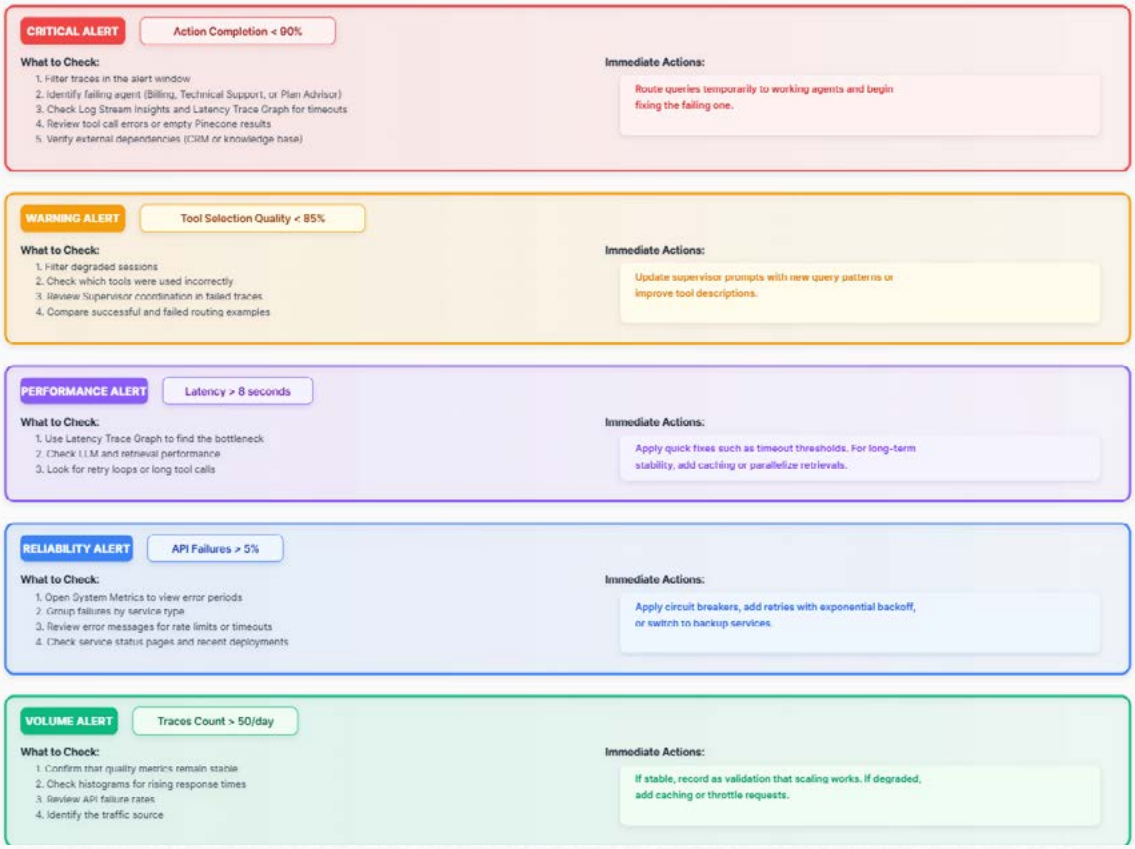


FIGURE 5.14 Common alert types and corresponding investigation steps

FIGURE 5.14 helps you understand the main alert categories. It highlights the conditions that trigger each alert, what you should check when it occurs, and the actions you can take to address it effectively.

After you set up these alerts, the final step is to close the loop once an issue has been resolved. Observability is most useful when it confirms that a fix has worked and the system continues to perform as expected.

3. Close the Loop After Each Fix

Once you have applied a fix, continue monitoring the same metric for the next 24 to 48 hours. This helps verify that the issue has been fully resolved and that no new side effects have appeared. If performance improves, update your alert thresholds to reflect the new baseline.

Document each incident clearly:

- → Symptom: What alert was triggered
- → Root cause: What investigation revealed
- → Resolution: What changes were made

Sharing this record with your team ensures that similar issues can be identified and resolved faster in the future. Over time, these notes become a valuable reference for troubleshooting and training new members.

EXERCISE

Take 10 minutes to build a concrete monitoring plan:

1. Assess your current state:
   - How often do you check metrics now?
   - Which metrics do you track consistently?
   - How quickly do you notice performance drops?
2. Identify your gaps:
   - Compare your routine to Figure 5.13
   - List two metrics you should track but don't
   - Which alert from Figure 5.14 would've caught your last issue?
3. Start with one frequency:
   - Pick daily, weekly, or monthly monitoring
   - Choose 2-3 metrics to track
   - Set up one alert based on your baseline
   - Block 15 minutes for your first review
4. Document your baseline:
   - Record current Action Completion, Tool Selection Quality, and Latency
   - Note what "normal" performance looks like
   - Write down any existing patterns (traffic spikes, slow periods)

Start small with daily monitoring. Once that's automatic, add weekly reviews. Consistent monitoring catches problems early.

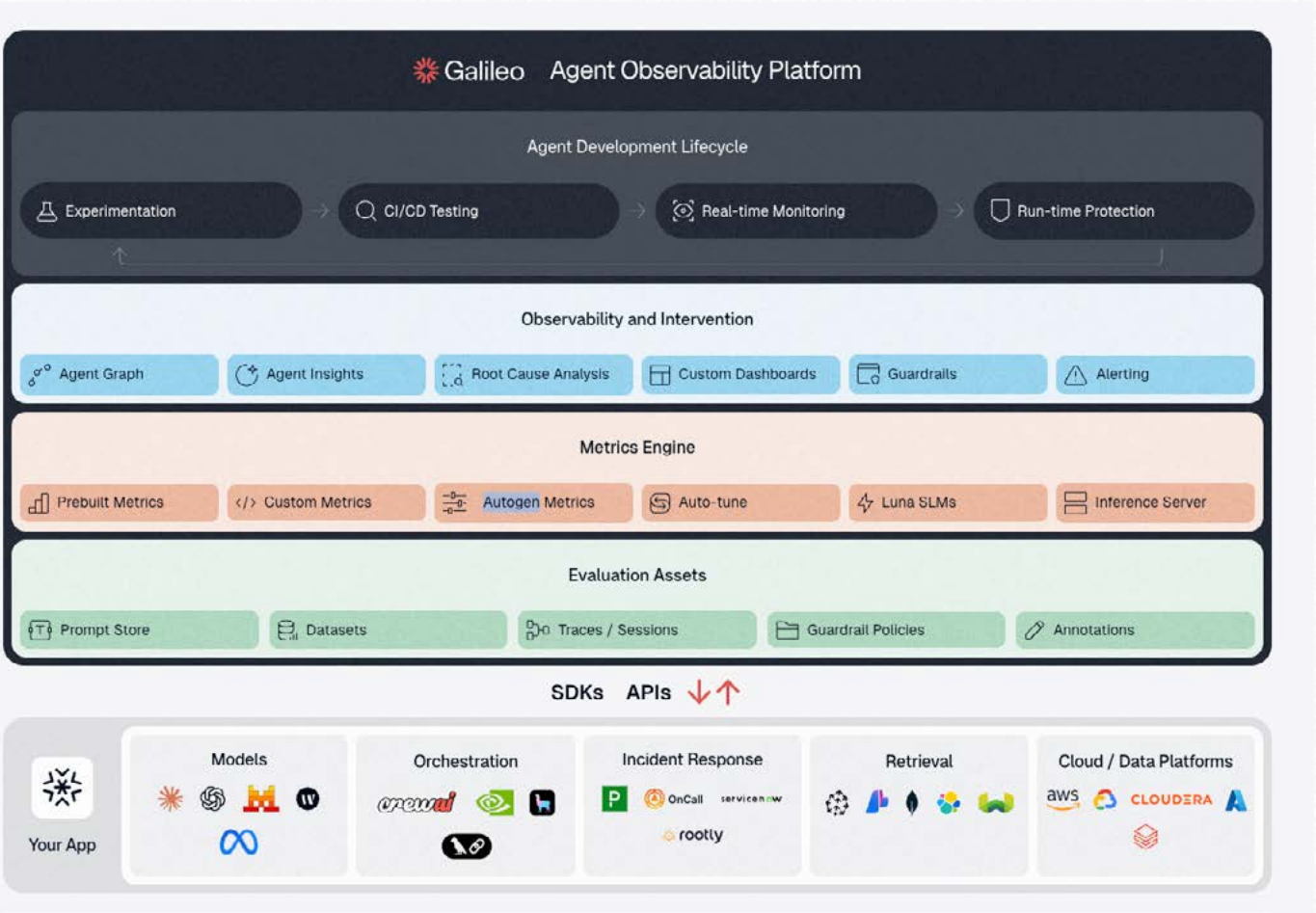
What You've Learned

In this chapter, you learned how to build a multi-agent system through LangGraph, provide them with the right tools, and utilize Galileo to understand and improve it effectively.

You now know how to:

- Define clear roles for each agent so the queries are routed correctly and the conversations stay on track.
- Write detailed tool descriptions that tell agents exactly when and how to use them.
- Track Action Completion and Tool Selection Quality to understand and monitor how effectively agents adhere to users' requests.
- Use both predefined and custom metrics along with the Insights Engine to find areas where the performance drops or context is lost.
- Prepare and maintain a consistent monitoring routine to verify your improvements and keep your system reliable over time.

From Chapter 1 to Production



You started this book questioning whether multi-agent systems were worth the complexity. Five chapters later, you have the tools to decide when they make sense and the mental frameworks to build them correctly.

You know when specialization justifies coordination overhead. You can choose architectures that match your constraints. You understand how to manage context, prevent failures, and monitor systems that actually work in production.

Remember that the difference between theory and reliable systems comes down to deliberate practice. You've built the foundation. Now apply it.

Curious about RAG architectures or how to evaluate agent quality with LLMs? Explore our popular ebooks that dive deep into RAG, agent engineering, evaluation strategies, and production system design. Each follows the same practical, hands-on approach you've seen here.

Try Galileo for free to put these patterns into practice. Use the same process to track your agents, identify issues early, and iterate quickly.

We hope you liked the book and will help us educate more developers by sharing it with your friends who are on the same journey.

01 Glossary

Term	Description
Agent Observability	Ability to monitor, trace, and interpret how agents make decisions and coordinate in multi-agent systems.
Caching	Reusing processed context to lower latency and reduce token cost during multi-agent exchanges.
Centralized Multi-Agent Architecture	Design where a single orchestrator manages agent interactions and consolidates outputs.
Compaction	Summarizing less relevant parts of context to maintain focus within token limits.
Context Adherence	Metric measuring how accurately an agent's output reflects the provided context.
Context Clash	Failure mode where contradictory information in context causes decision-making errors.
Context Confusion	Failure mode that occurs when too many tools create ambiguity in tool selection.
Context Distraction	When accumulated history distracts an agent from reasoning about the current task.
Context Isolation	Dividing workloads so each agent operates with an independent context window.
Context Poisoning	When a false or hallucinated detail persists in context, corrupting all subsequent reasoning.

02 Glossary

Term	Description
Coordination Overhead	Extra computation and communication required for agents to coordinate effectively.
Decentralized Architecture	Architecture that allows agents to communicate directly, improving resilience but reducing control.
Dynamic Routing	Process of directing queries to the most appropriate agent based on task complexity and confidence.
Dynamic Tool Retrieval	Loading only the most relevant tools at runtime to reduce confusion and improve efficiency.
Fault Tolerance	The capability of a multi-agent system to keep functioning when individual agents fail.
Graph View	Galileo visualization that maps multi-agent workflows, showing coordination patterns across sessions.
Hierarchical Architecture	A layered structure where supervisor agents manage specialized sub-teams for coordination.
History Context	Past interactions or learned patterns retained to preserve continuity in reasoning.
Hybrid Architecture	System combining centralized orchestration with decentralized execution for flexibility.
Instructions Context	System prompts and behavioral rules defining how agents should act.

03 Glossary

Term	Description
Knowledge Context	Factual or retrieved data informing agent reasoning during a task.
Log Stream Insights	Galileo's feature that detects patterns in performance anomalies across agent executions.
Multi-Agent System	Architecture where specialized agents collaborate to complete complex or distributed tasks.
Context offloading	Moving data out of context into external storage while retaining summaries.
Orchestrator	Central coordinating agent that manages routing, state, and aggregation of results.
Parallel Processing	Running multiple agents simultaneously to accelerate processing and improve throughput.
Peer-to-Peer Coordination	Architecture enabling agents to exchange information directly without an orchestrator.
Retrieval	Fetching only relevant external data into context to enhance decision accuracy.
Session-Level Tracking	Tracking whether multi-agent sessions successfully achieve user objectives.
Single-Agent System	Setup where a single model handles all tasks without inter-agent coordination.

04 Glossary

Term	Description
State Machine	Model representing defined agent states and transitions that control system behavior.
Step-Level Tracking	Monitoring each agent's decision and tool calls within a conversation for debugging.
Supervisor Agent	Agent responsible for routing tasks to specialists and coordinating their responses.
System-Level Tracking	Aggregate measurement of latency, cost, and success across all agent interactions.
Timeline View	Galileo's temporal breakdown tool shows the time spent in each phase of agent execution (communication, retrieval, and generation) to identify inefficiencies.
Tools Context	Tool definitions, outputs, and error logs agents use to perform external actions.
Trace View	Displays complete agent interactions in a collapsible tree structure, showing nested calls and decision points at each step.
Validation Gates/Guardrails	Checkpoints where agents sequentially verify or validate outputs before passing them forward.