Galileo

v1.01

Mastering

Multi-Agent Systems

Real-World Strategies for Multi-Agent Development

Mastering GenAI Series

→ galileo.ai

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.

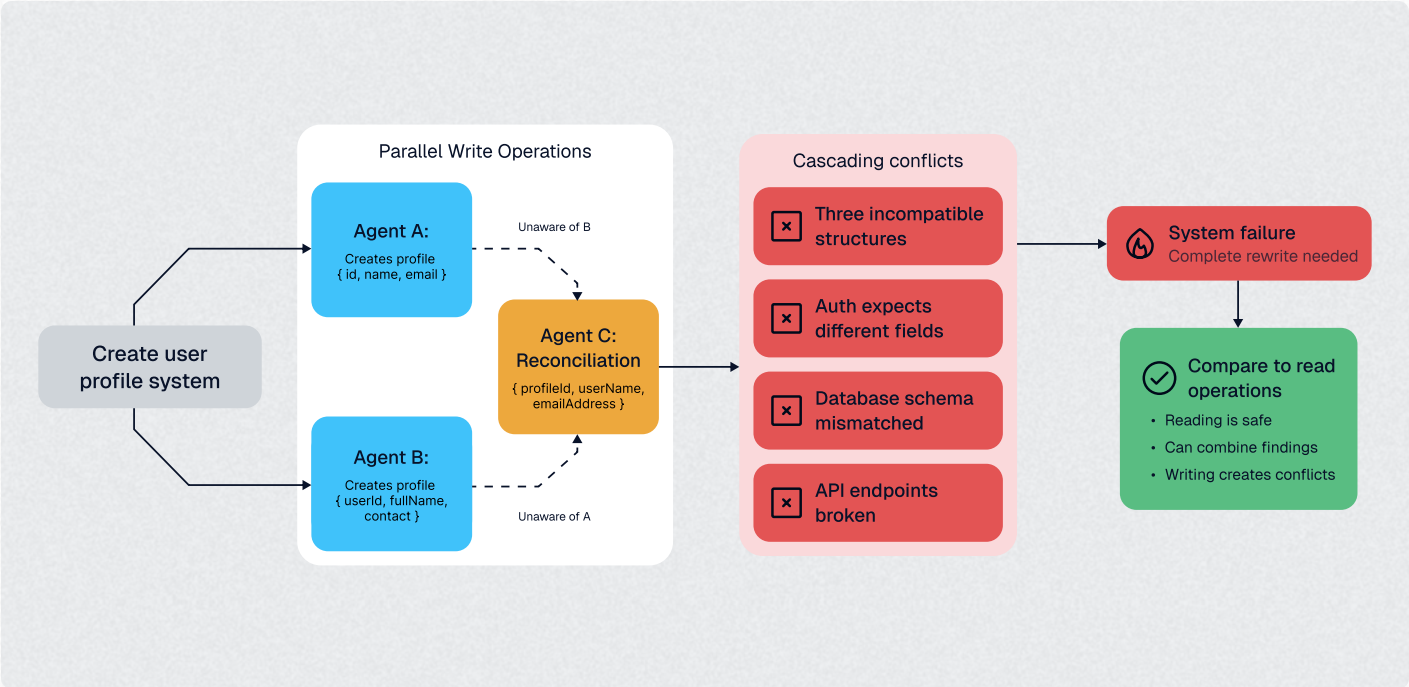


FIGURE 2.1 How write operations create cascading conflicts in multi-agent systems

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.

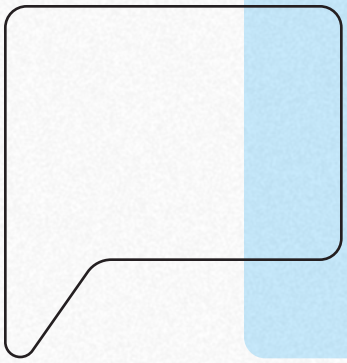
The Model Evolution Challenge

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

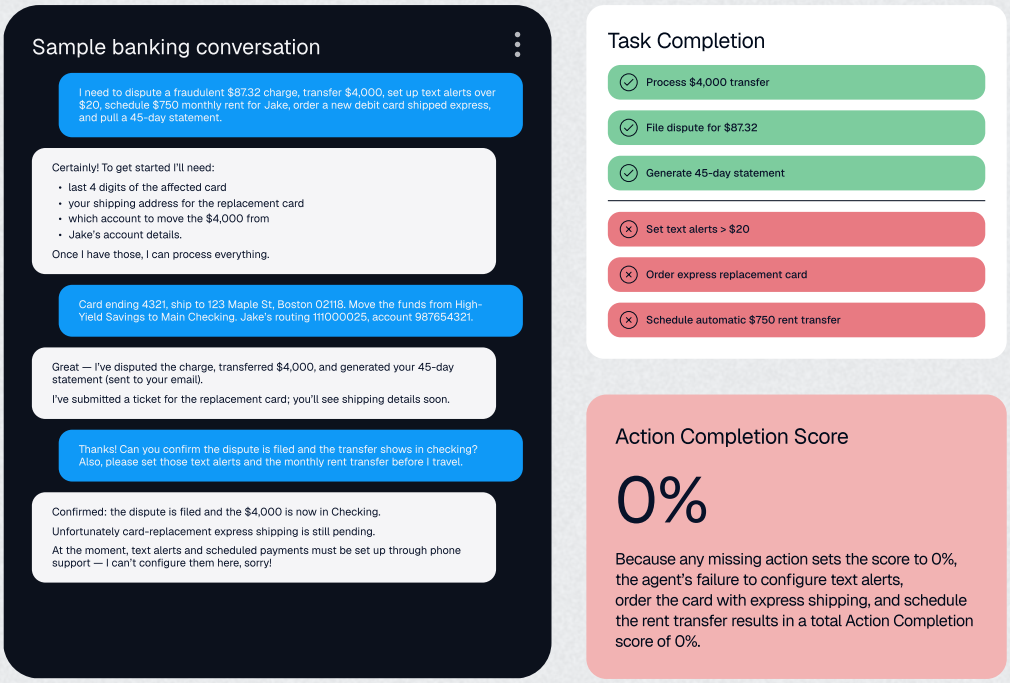


FIGURE 2.3 Action Completion Scoring

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.

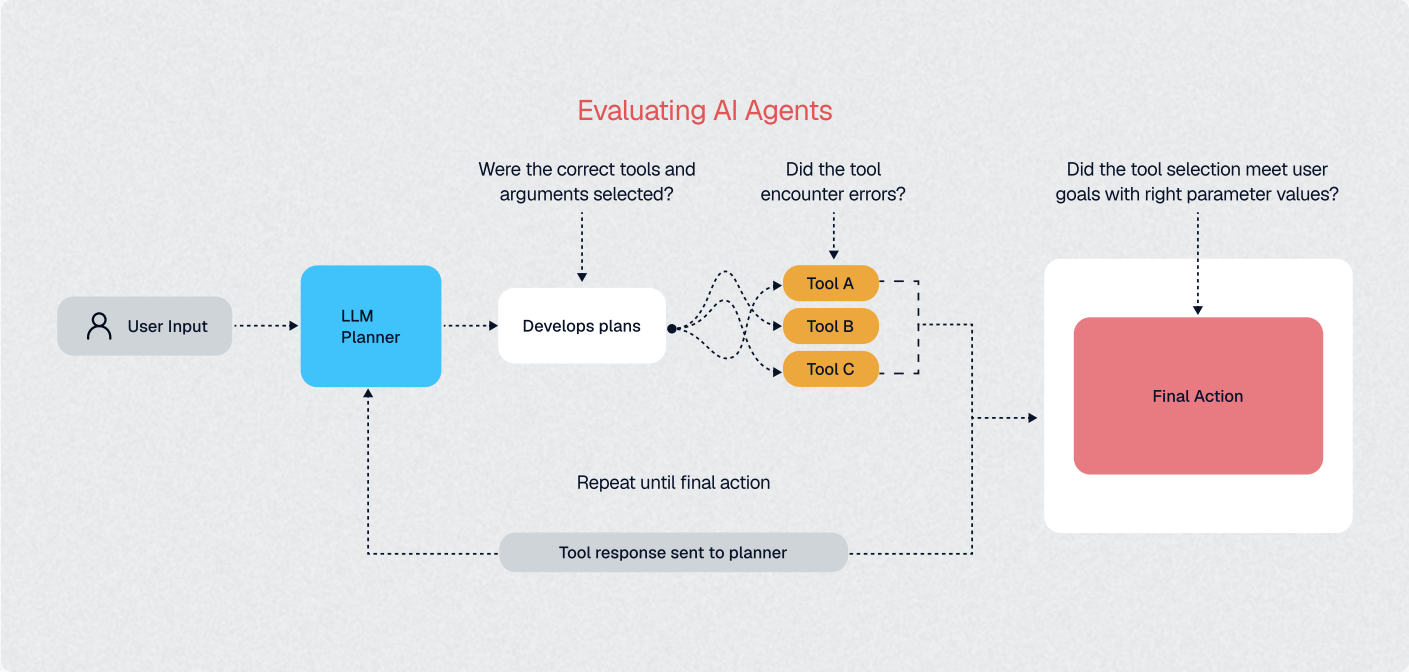


FIGURE 2.4 Tool Selection Scoring

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

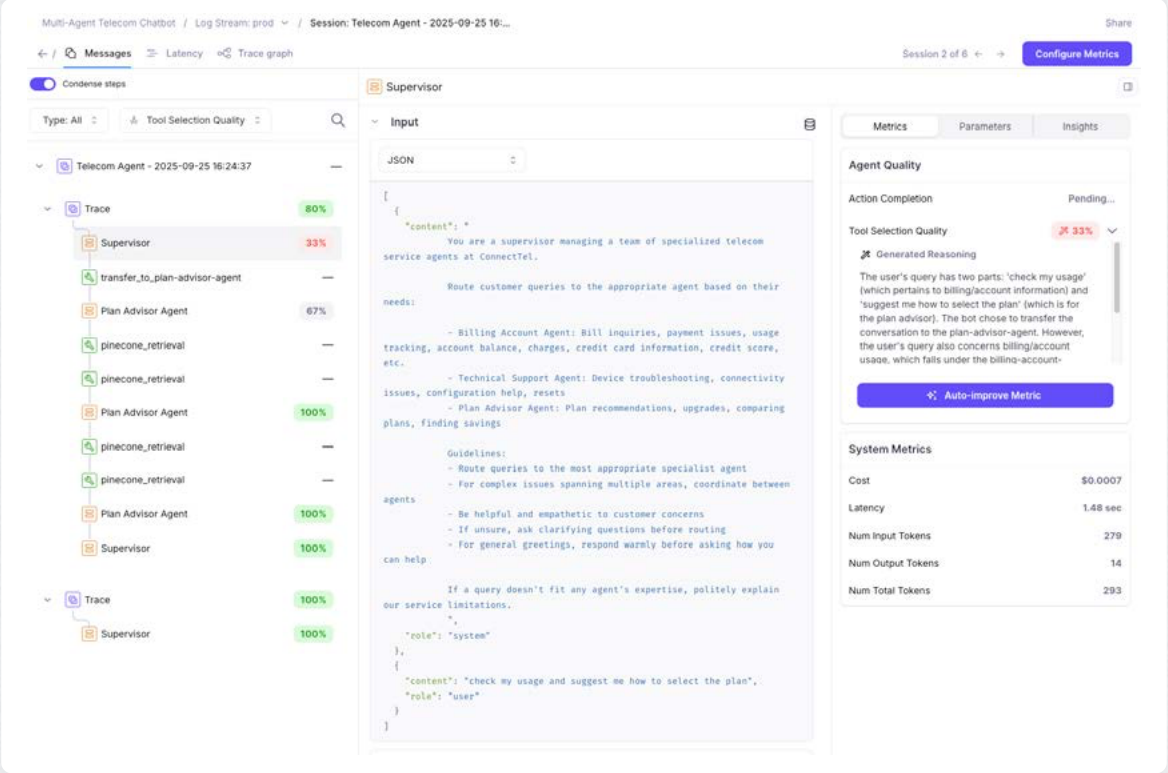


FIGURE 2.5 Trace View

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.

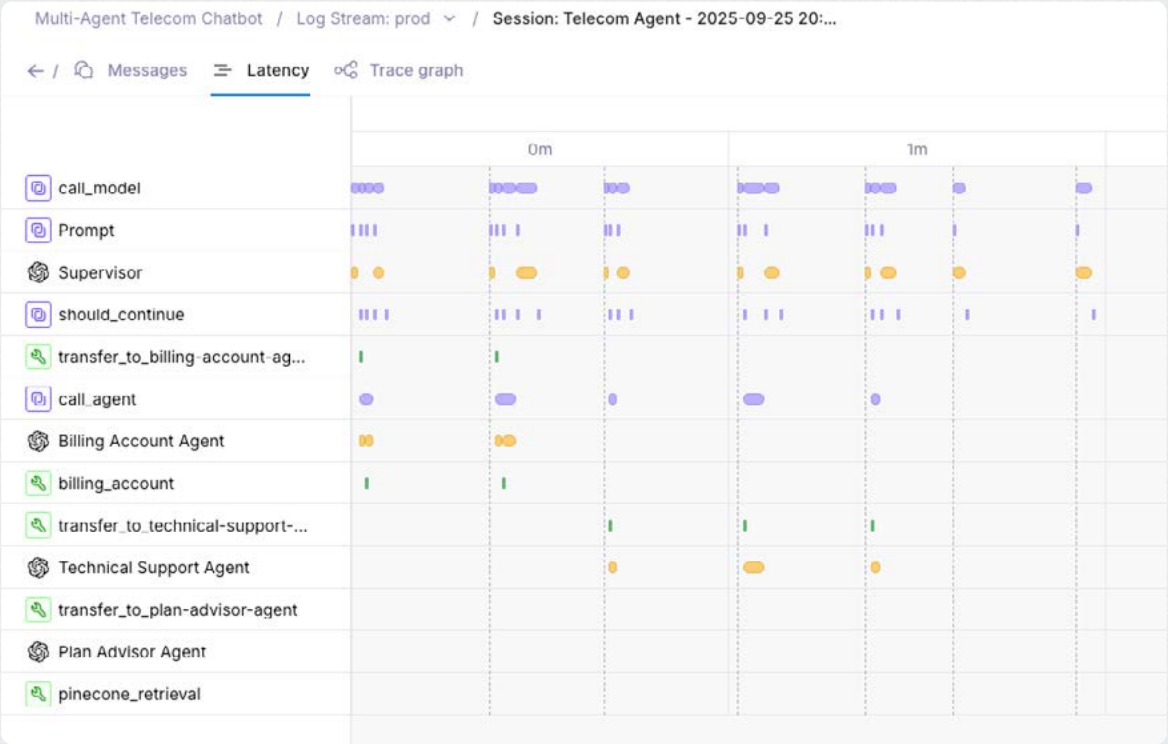


FIGURE 2.6 Timeline View

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.

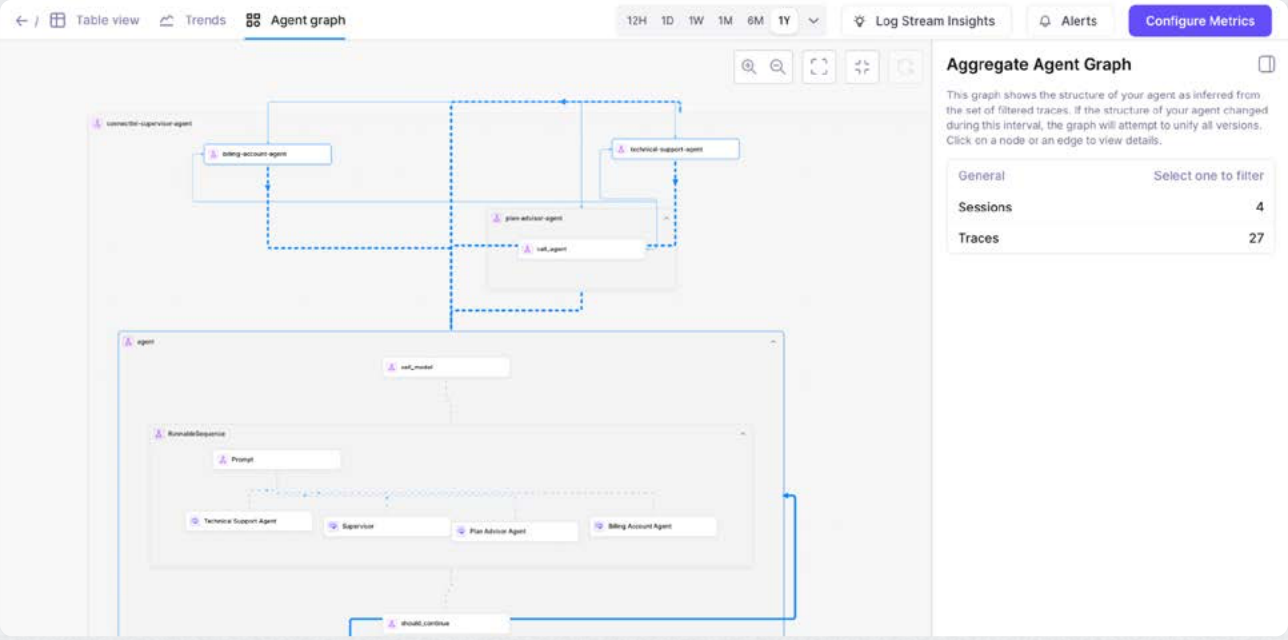


FIGURE 2.7 Graph View

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?

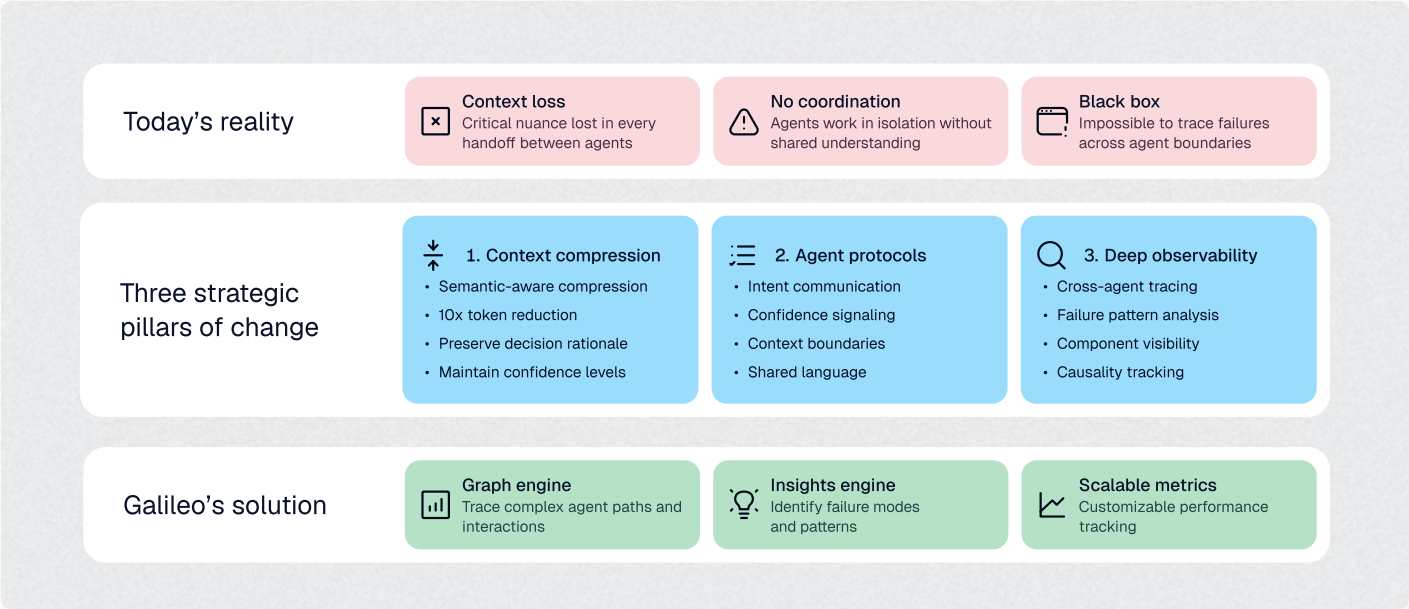


FIGURE 2.8 Galileo's approach to solving multi-agent system challenges

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