Sep 18, 2025 · 14 min read

Architectures for Multi-Agent Systems

Pratik Bhavsar
Galileo Labs

You ask your AI assistant to plan your daughter's 8th birthday party.

Theme: unicorns. Budget: $1000. Guests: 20 kids.

The single agent starts its sequential process: researching venues... then decorations... then cake options... then entertainment. Ten minutes later, it's still comparing balloon suppliers, hasn't even touched food options, and the venue it originally suggested is now fully booked.

Now imagine multiple specialized agents attacking this problem simultaneously. The venue agent is calling three local party spaces checking availability and pricing. The decoration agent is assembling a unicorn theme package from party suppliers. The food agent is getting quotes from bakeries and calculating pizza quantities. The entertainment agent is checking which magicians and face painters are free that Saturday.

Five minutes later, you have three complete party plans to choose from, each with vendor contacts, itemized budgets, and backup options.

CONTENTS

Why Architecture Shapes Everything
The Four Primary Architectures
Frameworks
Choosing Your Framework
Making the Architecture Decision
The Final Verdict

SHARE

But here's where multi-agent gets interesting. When the venue agent discovers that the perfect location has a unicorn mural, it immediately notifies the decoration agent to adjust the budget. When the cake agent finds a bakery offering a discount for large orders, the food agent pivots to include cupcakes. The agents adapt and coordinate in real-time, creating solutions a single agent would never discover.

This coordination capability reveals why architecture matters: it determines not just speed but also what problems you can solve.

By the way, we are constantly sharing our insights on agents. Don't miss out on other pieces in the Mastering Agent series.

- Benefits of Multi-Agent Systems
- Why Multi-Agent Systems Fail
- Learn to fix agents
- Top agent benchmarks
- Metrics for chatbot agents
- Types of agents

Why Architecture Shapes Everything

The way agents are organized fundamentally changes system behavior. Three factors explain 95% of performance variance: token usage accounts for 80% by itself, with tool calls and model choice making up the rest. Your architecture determines how efficiently you distribute that token usage across parallel work streams.

Information flow: In centralized systems, all data flows through one hub, creating a bottleneck but ensuring consistency. Decentralized systems allow direct peer communication, enabling faster local decisions but risking global inconsistency.

Failure modes: A centralized orchestrator creates a single point of failure. Take it down, and the entire system stops. Decentralized architectures continue operating even when multiple agents fail, but coordination becomes exponentially harder.

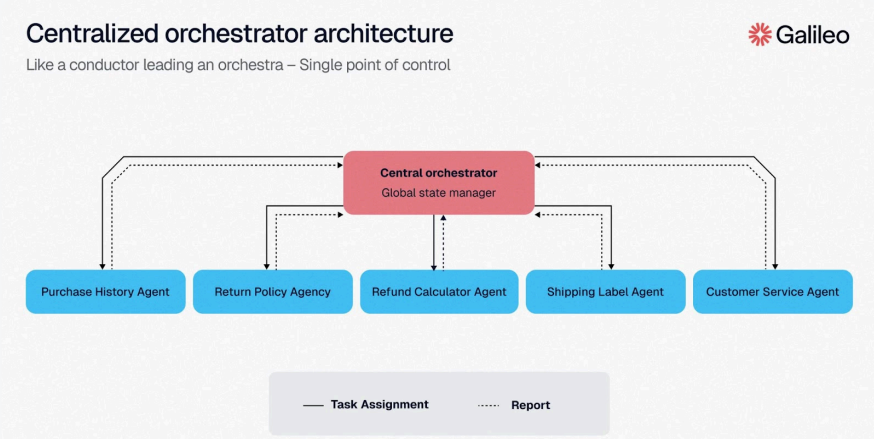
Scaling patterns: Single-agent architectures scale poorly with increasing tool count and context size. Performance decreases significantly even when

the context is irrelevant to the target task. Multi-agent architectures solve this by distributing context across agents with separate windows.

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.

1. Centralized: The Orchestrator Pattern



Think of a conductor leading an orchestra. A single powerful agent acts as the brain, coordinating all other agents. This central agent allocates tasks, monitors progress, and synthesizes results.

The orchestrator maintains global state and makes all routing decisions. Every action traces back to central decision-making, creating predictable, debuggable behavior. You always know why something happened and which agent made the call. This scales well by leveraging the map-reduce pattern.

Consider Anthropic's Research agent handling a complex query like "emerging AI startups in healthcare." The lead agent analyzes the request, develops a strategy, then spawns specialized subagents. One explores funding data, another investigates clinical applications, a third examines the regulatory landscape, and a fourth analyzes competitive dynamics. Each reports back to the orchestrator, which synthesizes findings into a coherent response.

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 in the central agent

The mental model is simple. Debugging is straightforward. You get guaranteed consistency and clear accountability. But the orchestrator becomes a bottleneck at 10-20 agents. Scale further, and coordination overhead overwhelms the central agent.

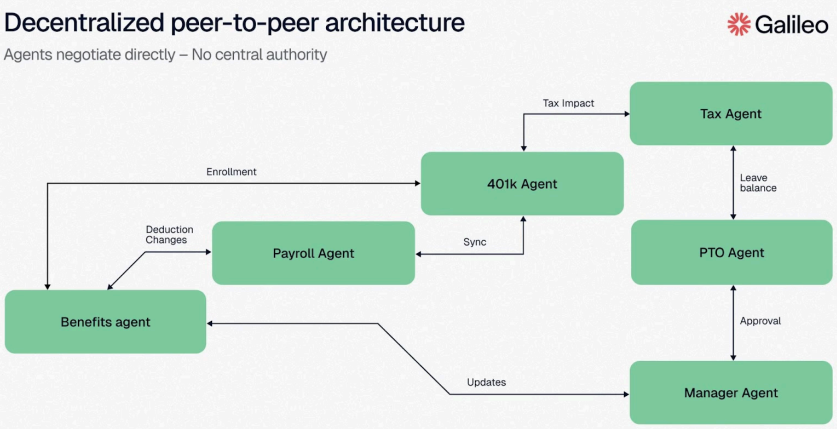
Take a customer service system handling product returns. The orchestrator receives the return request, then delegates: one agent checks purchase history, another reviews return policy eligibility, a third calculates refund amounts, and a fourth arranges shipping labels. Each reports back to the hub,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

The weakness? If the orchestrator fails, all return processing stops. You've traded resilience for simplicity.

Here is a reference implementation from Langgraph.

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

model = ChatOpenAI()

def supervisor(state: MessagesState) -> Command[Literal["ager
    # you can pass relevant parts of the state to the LLM (e.
    # to determine which agent to call next. a common patterr
    # with a structured output (e.g. force it to return an ou
    response = model.invoke( ... )
    # route to one of the agents or exit based on the supervi
    # if the supervisor returns "__end__", the graph will fir
    return Command(goto=response["next_agent"])

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

def agent_2(state: MessagesState) -> Command[Literal["supervi
    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



In this design, agents communicate directly with neighbors, making local decisions without central coordination. Picture a marketplace where vendors negotiate directly rather than through an auctioneer.

Intelligence emerges from local interactions. 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 401k 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. Each agent responds to requests and coordinates with peers as needed. When the benefits agent goes down for maintenance, other agents continue operating.

Performance profile shifts dramatically:

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

You gain resilience—one agent failing doesn't crash the system. You can scale to hundreds of agents without architectural changes. But coordinating global behavior becomes challenging. Maintaining consistency or enforcing system-wide priorities gets difficult without central oversight.

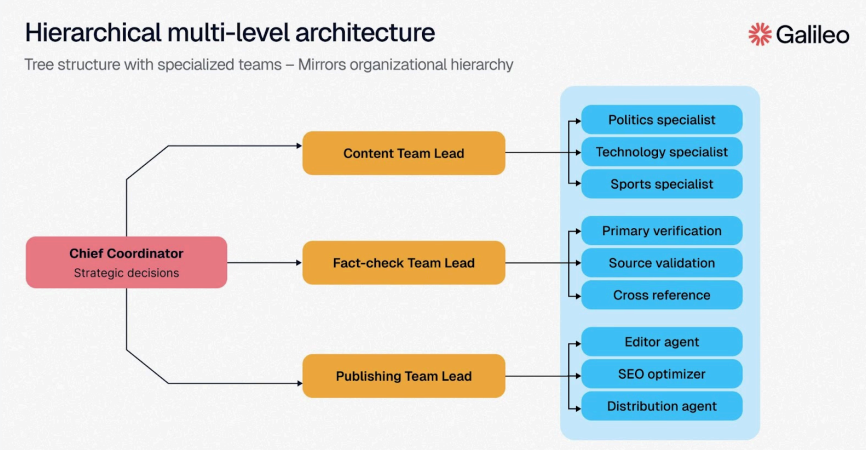
The decentralized approach excels when agents need autonomy and the system must survive partial failures. It struggles when you need guaranteed consistency or centralized decision-making.

Here is a reference implementation from Langgraph.

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

```python
model = ChatOpenAI()

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

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

def agent_3(state: MessagesState) -> Command[Literal["agent_1
    ...
    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



Multiple layers of supervision create a tree structure. Specialized teams work under team leaders who report to higher-level coordinators. It mirrors human organizations.

Decisions cascade down the hierarchy while information bubbles up. Each level abstracts complexity for the level above. A supervisor agent performs task planning, breaks down work, assigns sub-tasks, and facilitates communication between specialists.

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.

Take a news aggregation platform. The top supervisor coordinates between content, fact-checking, and publishing teams. The content supervisor manages agents for different beats: politics, technology, sports. Each beat agent oversees specialized scrapers for different sources. The fact-checking supervisor manages verification agents using different methods. Information flows up through summaries while specific tasks flow down through assignments.

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

This architecture handles complex, multi-domain problems elegantly. The organizational structure feels natural to human teams. But coordination overhead between levels adds complexity. Too many agents can overwhelm supervisor management capacity. Middle management isn't just a human problem—it affects AI systems too.

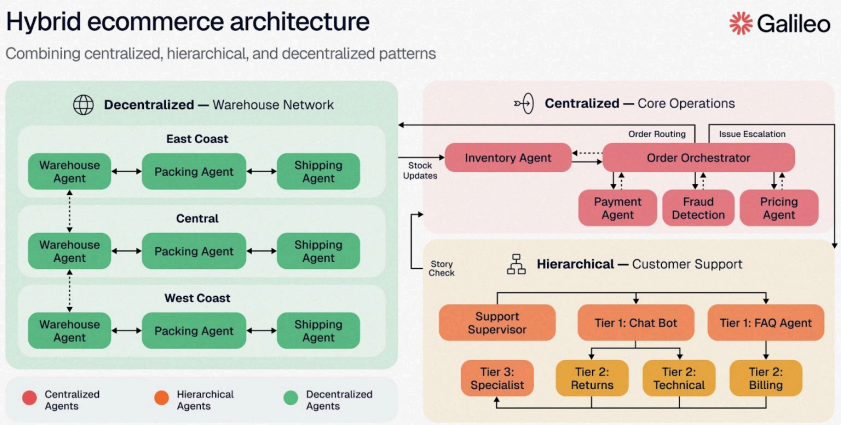
The hierarchical approach works best for problems with natural decomposition into sub-problems. Each level should add meaningful abstraction, not just bureaucracy.

Here is a reference implementation from Langgraph.

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

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

def team_1_supervisor(state: MessagesState) -> Command[Litera
    response = model.invoke(...)
    return Command(goto=response["next_agent"])

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

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

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()
```

```python
# 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[Lit
    # you can pass relevant parts of the state to the LLM (e.
    # to determine which team to call next. a common pattern
    # with a structured output (e.g. force it to return an ou
    response = model.invoke(...)
    # route to one of the teams or exit based on the supervis
    # if the supervisor returns "__end__", the graph will fir
    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



Hybrid designs combine centralized strategic coordination with decentralized tactical execution. Different parts of the system use different organizational patterns based on their requirements.

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.

Consider a food delivery platform. The central orchestrator handles tasks requiring consistency and transaction integrity like order placement and payment processing. Once confirmed, regional agent clusters take over. Restaurant agents coordinate directly with nearby driver agents for pickup timing. Driver agents negotiate with each other for efficient route sharing. Customer notification agents operate independently based on local delivery status.

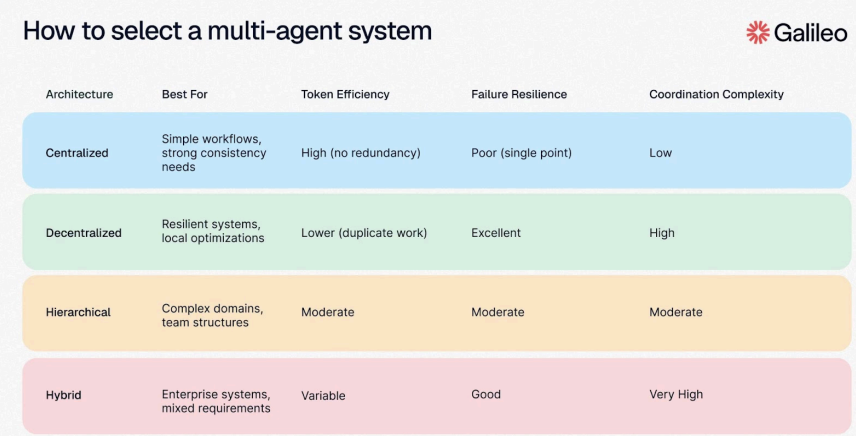
The center maintains what must be centralized: payments, order integrity, customer data. The edges optimize what benefits from local knowledge: routing, timing, real-time adjustments.

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

You balance control and resilience. You adapt architecture to different problem domains within the same system. This flexibility makes hybrid architectures popular for large-scale applications.

The complexity multiplies, though. Implementation and debugging become harder. You need to define the boundary between centralized and decentralized zones carefully. Agents must understand when to escalate to central coordination versus handle locally.



Frameworks

The explosion of agent frameworks reflects different philosophical approaches to agent coordination. Each framework makes architectural choices that favor certain patterns. Lets look at some of the popular ones.

LangGraph: Graph-Based State Management

LangGraph models multi-agent workflows as directed graphs, making it natural for complex state management and conditional flows. Agents are nodes, communications are edges, and state flows through the graph.

Architectural strength: Hierarchical and hybrid patterns through graph-based workflows. The graph structure naturally represents reporting relationships and peer connections.

Key capability: Persistent memory and stateful interactions across long-running processes. State machines handle complex conversation flows where context must persist across multiple turns.

The framework excels when you need explicit control over agent interactions. Define your graph, set your edges, and watch information flow predictably through the system. Debugging becomes visual, and you can literally see where messages get stuck.

Agno: High-Performance Multi-Agent Systems

Agno provides an industry-leading multi-agent architecture emphasizing performance. The framework claims agent creation at 2μs per agent—orders of magnitude faster than alternatives.

Architectural strength: Supports all patterns but optimized for high-performance scenarios where agent spawn time matters.

Key capability: Native multi-modal support with minimal memory footprint (~3.75 KiB per agent). This efficiency enables scenarios with hundreds or thousands of agents.

The performance focus makes Agno ideal for real-time systems. Consider a gaming AI where agents control individual units. Spawning agents needs to happen within frame budgets. Traditional frameworks would cause visible lag; Agno maintains smooth performance.

Mastra: TypeScript-First Workflows

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

Architectural strength: Workflow-centric hybrid architectures where business logic matters more than agent autonomy.

Key capability: Graph-based state machines orchestrating complex sequences of AI operations. Integrations with web services come built-in.

Mastra fits naturally into existing web architectures. Your agents become part of your application stack, not a separate system. This integration makes it easier to maintain consistency between AI and traditional components.

CrewAI: Role-Based Collaboration

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