Agent Orchestration Patterns and Control-Plane Design

Executive Summary

The rapid maturation of large language models from single-turn conversational interfaces into autonomous reasoning agents has fundamentally altered the landscape of distributed systems engineering. As agents acquire the ability to formulate plans, invoke external APIs, and recursively spawn subagents, the primary operational challenge is no longer model optimization, but rather systemic coordination. The objective is to manage the lifecycle of these probabilistic entities—starting, scheduling, routing, pausing, resuming, and recovering them across independent fleets—without relying on a single, fragile "super-agent" that collapses under the weight of unbounded context windows and non-deterministic logic.
This report conducts a technically rigorous investigation into multi-agent orchestration, testing the central hypothesis that the most reliable architecture for enterprise deployment is a hierarchical hybrid system. In this proposed architecture, a strictly deterministic control plane owns workflow state, scheduling, token budgets, and security policies; fleet-level supervisors coordinate localized work; probabilistic agents perform bounded reasoning tasks; and all consequential actions pass through explicit verification and human approval checkpoints. Analysis of production deployments, formal resource governance frameworks, and disaster recovery playbooks strongly supports this hypothesis.[1] Delegating orchestration logic entirely to language models invariably leads to runaway costs, unpredictable failure modes, and un-auditable state transitions.[4] Conversely, fully decentralized peer-to-peer swarms lack the deterministic governance required for secure execution. By establishing an architecture where deterministic infrastructure bounds probabilistic reasoning, organisations can scale multi-fleet agentic systems securely and predictably.

Key Conclusions

The investigation into agent orchestration architectures yields several definitive conclusions that must govern system design. First, durable execution is an absolute prerequisite for agentic workflows. Because multi-step reasoning loops and tool invocations span extended temporal windows—ranging from minutes to weeks—they are exceptionally vulnerable to network partitions, pod preemptions, and provider latency.[2] Relying on simple in-memory retries causes duplicated non-idempotent tool executions (e.g., double-charging a credit card) and state corruption. The infrastructure must persist every state transition before acknowledging success, enabling interrupted workflows to resume exactly where they halted.[2]
Second, the structural separation of the control plane from the execution plane is paramount.[3] The control plane must remain strictly deterministic, authored in standard software, and entirely independent of language model reasoning. It dictates access control, tracks execution

state, enforces token budgets, and logs events. The execution plane—where agents formulate plans and invoke tools—must operate as a cluster of ephemeral, stateless workers bounded by the policies distributed by the control plane. Mixing these responsibilities inside a single language model context guarantees systemic instability.

Third, formal resource governance—often formalized as Agent Contracts—must bound multi-agent systems.[1] Agents capable of dynamically spawning subagents will inevitably enter infinite loops of failure and retry without deterministic circuit breakers, recursion depth limits, and semantic similarity checks.[4] Every agent must operate under explicit conservation laws governing temporal limits, computational depth, and financial token consumption.

Finally, event-driven architectures significantly outperform continuous polling loops. Human-in-the-loop interventions should be designed as durable workflow pauses. Rather than keeping agent processes active while awaiting human authorization, the system must durably hibernate the state, emit an event, and awaken upon the receipt of an external cryptographic signal, conserving compute and enabling indefinite wait times.[9]

Terminology and Conceptual Model

To establish a precise architectural vocabulary, the following definitions delineate the components of multi-agent orchestration.

Term	Definition	Architecture Location	Implementation	Practical Example	Common Misunderstanding
Orchestrator	Central engine managing workflow steps, state transitions, and execution order.	Control Plane	Deterministic	A Temporal cluster tracking a research workflow.[6]	Believing the orchestrator is an AI model making routing decisions.
Control plane	Centralized layer managing policies, state, registries, scheduling, and	Global Infrastructure	Deterministic	A central policy server defining fleet database access.	Confusing it with the models that actually generate content.

	governance.[3]				
Execution plane	Decentralized layer where models process data, run reasoning loops, and invoke tools.[3]	Worker Nodes	AI/Probabilistic	A Python container running an Anthropic model.[10]	Thinking it maintains the authoritative state of the workflow.
Data plane	Pathways carrying telemetry, artefacts, and task data between components.[3]	Worker Infrastructure	Deterministic	The secure channel transmitting tool outputs back to the orchestrator.	Assuming it analyzes the data it carries.
Scheduler	Evaluates when tasks become eligible for execution based on time, dependencies, and resources.	Control Plane	Deterministic	A cron-based trigger initiating competitor monitoring daily.	Thinking the scheduler decides *how* a task is done.
Dispatcher	Assigns eligible tasks to specific queues or workers based on	Control Plane	Deterministic	Sending a python-formatting task to a code-execution queue.	Confusing dispatching with AI-based semantic routing.

	capability matching.				
Router	Evaluates incoming unstructured inputs and directs them to specialized pathways.[11]	Boundary/Gateway	Hybrid (AI/Rules)	Using embedding similarity search to route a query to a support agent.	Believing routers always require large language models.
Coordinator	Manages the breakdown and distribution of a larger goal into parallel or sequential subtasks.[12]	Fleet level	Hybrid	An agent decomposing a feature request into three coding tasks.[11]	Confusing it with the global orchestrator.
Supervisor	Managerial component overseeing a specific fleet, monitoring worker health, and aggregating results.	Fleet level	Deterministic/AI	A quality-control script checking if worker agents produced valid JSON.	Assuming supervisors are always LLMs.
Worker agent	Base entity executing a defined task, generating content, or invoking	Execution Plane	AI	A model instance writing a unit test.	Thinking workers have system-wide context.

	tools.				
Planner agent	Dedicated to decomposing complex goals into structured task graphs.	Execution Plane	AI	A model returning a JSON array of step-by-step instructions.	Assuming the planner also executes the steps.
Executor agent	Carries out one step of a plan, restricted from broader reasoning.	Execution Plane	AI	An agent granted only file-write permissions to save a script.	Believing executors have autonomy to change the overall plan.
Specialist agent	Worker highly fine-tuned or prompt-engineered for one narrow domain.	Execution Plane	AI	An agent optimized solely to translate English into French.	Thinking specialists can handle general conversational queries.
Evaluator agent	Operates in an optimization loop, tasked with critiquing the output of another agent.[13]	Execution Plane	AI	A model scoring a generated article against a rubric.	Assuming evaluators are deterministic test scripts.
Critic	Focused on finding	Execution	AI	An agent reviewing	Confusing it with a

agent	flaws, security risks, or logic errors in plans.	Plane		code for SQL injection vulnerabilities.	guardrail; critics analyze deep logic.
Verifier agent	Checks outputs against objective, deterministic truth constraints before completion.	Execution Plane	Hybrid	An agent running a Python compiler to ensure code executes.	Believing verifiers rely on subjective LLM judgement.
Guardrail agent	Lightweight, fast agent running in parallel to screen inputs/outputs for policy violations.[11]	Execution Plane	AI	A small model checking if a prompt contains malicious instructions.	Thinking guardrails are slow, deep-reasoning safety checks.
Relay agent	Passive component translating protocols, moving data, or bridging fleets without reasoning.	Data Plane	Deterministic	A script moving a finished file from S3 to a Git repository.	Assuming all agents contain an LLM prompt.
Passive agent	Acts only when explicitly invoked,	Execution Plane	AI	A summarization model waiting for	Believing agents must run continuousl

	holding no autonomous internal loops.			webhooks.	y to be "agentic."
Subagent	Temporary worker spawned dynamically by a parent agent to handle a decomposed sub-task.[4]	Execution Plane	AI	An agent instantiated just to read a single webpage and return data.	Thinking subagents have different permissions than their parents.
Temporary agent	Ephemeral runtime configuration existing only for the duration of a single task.	Execution Plane	AI	A containerized agent destroyed after completing a query.	Confusing it with a persistent bot identity.
Persistent agent	Long-lived identity with enduring episodic memory and consistent access policies.	Control Plane/Memory	Hybrid	A personal assistant agent remembering preferences over years.	Assuming it runs continuously in a while loop.
Agent fleet	Logically grouped namespace of agents sharing supervisors,	Control Plane	Deterministic	A "Software Engineering" fleet containing coders, testers, and	Confusing a fleet with a decentralized swarm.

	policies, and capabilities.			reviewers.	
Agent swarm	Decentralized, peer-to-peer network lacking a rigid control plane, relying on emergent coordination.	Execution Plane	AI	Multiple instances communicating directly to solve a puzzle.	Believing swarms are auditable and safe for enterprise use.
Agent hierarchy	Structured tree where global orchestrators manage fleet supervisors, which manage workers.[14]	System Architecture	Hybrid	A CEO agent routing to a CTO agent, routing to a Dev agent.	Assuming hierarchies are rigid and cannot scale dynamically.
Workflow	Predefined or dynamically generated structure of tasks, edges, and dependencies.[11]	Control Plane	Deterministic	A standard operating procedure for onboarding a new client.	Thinking workflows are just simple linear scripts.
Run	Single, unique execution instance of	Control Plane	Deterministic	The onboarding process for "Client A"	Confusing a run with the abstract workflow

	a workflow.			initiated today.	definition.
Task	Smallest atomic unit of work assigned to an agent or system.	Execution Plane	Deterministic	The command to "fetch user details from the database."	Believing tasks are open-ended goals.
Job	Often synonymous with task or workflow; a discrete unit of scheduled computation.	Control Plane	Deterministic	A nightly batch process triggering a fleet.	Confusing it with a continuously running service.
Step	Sequential progression within a workflow or an agent's reasoning loop.[5]	Execution Plane	AI	The agent's decision to use the Search tool.	Thinking steps are always tracked by the control plane.
Node	Vertex in a DAG representing a task, function, or agent state.[9]	Control Plane	Deterministic	The "Review Code" step in a graph.	Assuming nodes contain infrastructure logic.
Edge	Directed connection between nodes	Control Plane	Deterministic	The transition indicating "Proceed to	Thinking edges are physical network

	defining control flow or state transitions.[1][5]			deploy if review passes."	connections.
Dependency	Prerequisite condition that must be met before a node becomes eligible for execution.	Control Plane	Deterministic	Task B cannot start until Task A writes a file.	Assuming dependencies are resolved by the agents themselves.
Directed acyclic graph	Workflow structure with nodes and edges containing no loops, ensuring definitive completion.	Control Plane	Deterministic	A data pipeline orchestrator like Apache Airflow.	Believing DAGs can natively handle endless conversational loops.
State machine	Mathematical model where the system is exactly in one of a finite number of states.[9]	Control Plane	Deterministic	Tracking an agent as "Idle," "Running," or "Blocked."	Confusing it with LLM context memory.
Event	Immutable record that something has happened	Data Plane	Deterministic	A JSON payload stating "Task Failed at 10:00	Thinking events are commands. Events report the

	within the system.[16]			AM."	past.
Command	Directive instructing the system or an agent to perform a specific action.[16]	Control Plane	Deterministic	An API call to "Terminate Run 123."	Confusing a command with an event. Commands attempt to change the future.
Signal	Asynchronous input sent to a running, durable workflow to update its state or unblock it.[2]	Control Plane	Deterministic	A user clicking "Approve" on a web dashboard, signaling the workflow.	Believing signals require the agent to constantly poll for them.
Message	Data passed between agents or components, often over queues or buses.	Data Plane	Deterministic	A payload containing a newly drafted article.	Thinking messages are always natural language text.
Queue	Buffer holding tasks or messages awaiting processing, enabling asynchronous load	Control Plane	Deterministic	A Kafka topic holding 500 pending translation requests.	Assuming queues execute the code themselves.

	balancing.				
Mailbox	Agent-specific queue holding messages targeted exclusively at that persistent identity.	Control Plane	Deterministic	A storage bucket for an agent's incoming asynchronous communications.	Confusing it with a shared global queue.
Thread	Sequence of conversational turns representing temporary context.	Execution Plane	AI	A chat history between a user and an agent.	Believing threads represent durable operational state.
Session	Logical grouping of interactions and state bounded by a specific time or user login.	Execution Plane	Hybrid	A browser session opened for a web-navigation agent.	Thinking sessions survive hardware reboots automatically.
Checkpoint	Durably saved state of a workflow, allowing resumption without repeating past work.[2]	Control Plane	Deterministic	Saving the exact token output of a completed step to Postgres.[2]	Assuming checkpoints save the internal neural weights of the model.
Durable	Infrastructu	Control	Deterministi	Temporal	Believing

execution	re paradigm where function execution state is persisted, allowing seamless recovery.[6]	Plane	c	automatically resuming a workflow after a node failure.[17]	durable execution is just a try/catch block with retries.
Idempotency	Property that an operation can be applied multiple times without changing the result beyond the initial application.	Execution Plane	Deterministic	An API returning the same success code if asked to charge an invoice twice.[9]	Thinking idempotency prevents failures. It prevents duplicate side effects.
Retry	Automated re-execution of a failed task according to a defined policy.	Control Plane	Deterministic	Attempting an LLM call 3 times on a 502 error.	Assuming retries solve semantic hallucination errors.
Timeout	Hard limit on how long a task is permitted to run before being forcefully marked as	Control Plane	Deterministic	Terminating an agent generating for 10 minutes.	Believing agents will naturally stop themselves.

	failed.[6]				
Lease	Time-bound lock held by a worker on a task, indicating it is currently processing it.	Control Plane	Deterministic	A worker claiming a task for 60 seconds; if it doesn't heartbeat, the lock expires.	Confusing a lease with permanent assignment.
Heartbeat	Periodic signal emitted by a worker proving it is alive and making progress.	Execution Plane	Deterministic	An agent pinging the orchestrator every 10 seconds.	Thinking heartbeats prove the agent is producing high-quality work.
Dead-letter queue	Storage location for messages or tasks that have repeatedly failed and exhausted retries.	Control Plane	Deterministic	A table of malformed LLM JSON outputs that broke the parser.	Assuming items in the DLQ are automatically deleted.
Backpressure	Mechanism to slow down producers when consumers are overwhelmed.	Control Plane	Deterministic	Halting prompt injection into a queue when API rate limits near 90%.	Confusing it with a simple rate limit; backpressure is dynamic feedback.
Concurren	Maximum	Control	Deterministi	Allowing	Believing

cy limit	number of tasks, agents, or API calls allowed to run simultaneously.	Plane	c	only 5 browser-use agents to run at once.	concurrency is purely about CPU cores.
Rate limit	Maximum allowed frequency of operations over a given time window.	Control Plane	Deterministic	Throttling requests to 10,000 tokens per minute.[18]	Confusing rate limits with financial budgets.
Circuit breaker	Pattern that stops executions to a failing subsystem to prevent cascading failures.[5]	Control Plane	Deterministic	Blocking calls to an LLM provider if latency exceeds 10s repeatedly.	Thinking circuit breakers retry the failing task.
Compensating action	Task designed to logically undo the work of a previously successful task when a subsequent task fails.	Control Plane	Deterministic	An agent issuing a refund because a downstream booking failed.[19]	Assuming database rollbacks work for external APIs.
Saga pattern	Distributed transaction architectur	Control Plane	Deterministic	Booking a flight, hotel, and car, and	Believing sagas provide

	e utilizing a sequence of local transactions and compensating actions.[19]			cancelling all if the car fails.	ACID properties across independent microservices.
Human-in-the-loop	Workflows that explicitly pause and require human intervention to proceed.[13]	Control Plane	Hybrid	A human reviewing a generated deployment script before execution.	Thinking the human must sit and watch the agent type in real-time.
Human-on-the-loop	Workflows that execute autonomously but allow a human to monitor and interrupt.	Control Plane	Hybrid	A human watching an agent play a game and pressing "stop" if it errs.	Confusing it with mandatory human approval.
Policy engine	Deterministic rule evaluator checking if an action violates governance constraints.	Control Plane	Deterministic	Rejecting a file deletion because the agent lacks write permissions.	Believing the policy engine uses an LLM to interpret rules.
Model gateway	Proxy component	Data Plane	Deterministic	A server routing	Thinking the gateway

	handling routing, logging, and failovers between agents and models.[18]			Claude queries to AWS Bedrock or Vertex AI based on latency.	alters the prompt semantically.
Agent registry	Database cataloging available agent identities, configurations, and state histories.	Control Plane	Deterministic	A table listing all active AI assistants in a company.	Confusing it with the capability registry.
Capability registry	Directory mapping specific tools or skills to the agents or fleets that can perform them.[20]	Control Plane	Deterministic	A manifest showing which agents know how to run Python code.	Assuming agents automatically know what tools exist globally.
Service discovery	Mechanisms allowing agents to locate other agents, fleets, or tools dynamically.	Control Plane	Deterministic	An agent looking up the address of the translation fleet.	Believing service discovery relies on LLM web searches.
Run ledger	Append-only,	Control Plane	Deterministic	An immutable	Thinking the ledger can

	cryptographically secure log of all actions taken during a workflow run.			audit trail of who approved a financial transaction.	be edited if the agent makes a mistake.
Event log	Time-series database recording state transitions for observability and replay.	Control Plane	Deterministic	Temporal's history of all events in a workflow.[16]	Confusing it with application error logs.
Audit trail	Comprehensive, human-readable record of system and agent behaviors for compliance.	Control Plane	Deterministic	A PDF export of an agent's reasoning steps and tool calls.	Assuming audit trails capture every internal neural activation.
Shared memory	Centralized database accessible by multiple agents, often used in blackboard patterns.	Data Plane	Deterministic	A vector database holding corporate knowledge.	Believing shared memory resolves conflicting data writes automatically.

Local memory	Storage scoped exclusively to a single agent instance or run, inaccessible to others.	Execution Plane	Deterministic	A temporary /tmp directory inside a worker container.	Thinking local memory survives after the task finishes.
Working memory	Short-term context loaded into the LLM prompt window (the current conversational state).	Execution Plane	AI	The text of the last 5 turns of conversation.	Confusing it with durable storage limits.
Episodic memory	Logs of past events, interactions, and workflow runs retrieved via similarity search.	Data Plane	Hybrid	An agent recalling how it solved a similar bug three weeks ago.	Assuming episodic memory is perfectly accurate.
Semantic memory	Generalized facts, rules, and knowledge distilled from episodic memory over time.	Data Plane	Hybrid	An agent's learned knowledge that "Server A always requires SSH key B."	Thinking semantic memory is hardcoded by developers.

Artefact store	Durable storage for the tangible outputs produced by agents (code files, PDFs, images).	Data Plane	Deterministic	An S3 bucket storing the final generated marketing videos.	Confusing it with the conversational event log.

Centralised Orchestration Versus Decentralised Swarms

Architectural coordination typically falls into centralised, decentralised, or hybrid paradigms. The selection of coordination model dictates the system's scalability, observability, and susceptibility to catastrophic failure.

A. Centralised Orchestration

A central component (e.g., Temporal, AWS Step Functions) assigns work, tracks execution, enforces policy, and coordinates agents.[2]

- **Advantages**: Centralization offers unparalleled operational simplicity, total visibility into state transitions, strict governance, and guaranteed failure recovery.[16] Security boundaries are easily enforced because all inter-agent communication passes through a single policy engine.
- **Disadvantages**: The orchestrator can become a systemic bottleneck and a concentrated point of failure.
- **Avoiding the Super-Agent Anti-Pattern**: To prevent the orchestrator from suffering context overload and scaling limitations, it must remain lightweight and deterministic. The orchestrator tracks task envelopes (metadata, status, budgets) rather than the semantic content of the model conversations. It behaves as a state machine routing opaque payloads, completely devoid of language model reasoning.[9]

B. Decentralised Swarm Coordination

Agents communicate or negotiate directly without a central reasoning authority, utilizing peer-to-peer communication, market-based task allocation, stigmergic coordination (leaving markers in the environment), and gossip protocols.

- **Advantages**: High resilience to local node failure, organic scalability, and the potential for emergent problem-solving behaviors.
- **Risks**: Decentralized swarms are perilous for enterprise deployment. They exhibit high risks of conflicting goals, duplicated work, and message storms. Without central governance, unclear ownership leads to deadlocks or livelocks. Most critically,

unbounded swarms are highly susceptible to runaway token consumption and inconsistent state mutation.[1] Debugging a distributed swarm requires complex distributed tracing, and weak governance makes compliance audits nearly impossible.

C. Hybrid Architecture

The hybrid model synthesizes the governance of centralization with the cognitive scalability of distributed swarms.

- **Architecture**: The control plane is centralized and strictly deterministic. Execution is distributed. Planning may be delegated to specialized LLMs, but their plans are validated deterministically.[3] Agents communicate exclusively through bounded, orchestrator-managed channels. Local fleet supervisors manage the intricacies of individual fleets, while a higher-level global fleet manager coordinates only cross-fleet concerns and dependency resolutions.
- **Decision Framework**: Choose centralised orchestration for deterministic data pipelines and financial transactions. Choose decentralised swarms only in research environments or highly constrained edge networks where connectivity to a control plane is impossible. Choose the hybrid architecture for all production enterprise multi-agent applications, as it provides the necessary balance of autonomy and auditable governance.

Event-Driven Orchestration Versus Continuous Agent Loops

The execution model dictates how agents consume compute resources and respond to stimuli.

Event-Driven Orchestration

Agents are activated strictly when a defined event, command, timer, state transition, or dependency occurs. The infrastructure relies on event buses, message brokers, queue consumers, and webhook triggers.

- **Mechanisms**: Handlers must be idempotent. The orchestrator guarantees event ordering and manages duplicate delivery and eventual consistency.[9]
- **Advantages**: Highly cost-effective (compute scales to zero when idle), perfectly auditable, predictable, and resilient to infrastructure failure.

Continuous Agent Loops

Agents remain persistently active, utilizing while True: polling loops or continuous reflection loops to inspect state and autonomously discover tasks.

- **Risks**: Continuous loops suffer from severe model drift, where slight hallucinations compound over time. Uncontrolled repetition leads to runaway token burn (the "infinite loop of death") and massive context growth.[4] Shutdown is difficult because the loop hides internal state in memory rather than persisting it to a durable ledger.

Comparison and Scheduled Wake-Ups

Event-driven orchestration dominates continuous loops in cost, predictability, failure recovery, observability, scalability, safety, and determinism. Continuous loops are genuinely necessary only in high-frequency, ultra-low-latency environments (e.g., real-time voice synthesis agents or physical robotics). For all other workflows, event-driven architecture is superior. The pattern of **scheduled wake-ups** bridges this gap: an agent is not continuously running, but the deterministic control plane suspends the workflow durably and reactivates the agent via an event trigger when a timer expires or new data arrives.[2] This simulates persistent autonomy while maintaining zero idle cost.

DAGs, State Machines, and Durable Workflows

Agentic behavior must be mapped into execution structures that survive real-world infrastructure volatility.

Sequential, Parallel, and Dependency-Based Execution

- **Sequential execution**: Tasks flow one after another with explicit state hand-offs and deterministic ordering. Early termination is possible if a checkpoint fails.
- **Parallel execution**: Work is fanned out to independent subproblems and fanned back in for result aggregation.[11] Crucially, the system must deterministically analyze data dependencies to confirm tasks are truly independent. *Incorrect parallelization* occurs when multiple coding agents edit the same Git repository simultaneously without isolation, resulting in catastrophic merge conflicts and lost work. True parallel execution requires strict concurrency limits and isolated state boundaries.
- **Dependency-based execution**: Tasks sit dormant until prerequisites complete. Dependencies can be hard (Task A must succeed), soft (Task A must finish, regardless of success), conditional, or based on human approval.

DAG-Based Workflows

Directed Acyclic Graph (DAG) orchestration ensures tasks (nodes) and dependencies (edges) resolve in a topological order. Modern orchestration allows an AI planner to propose a runtime-generated DAG. However, deterministic software must immediately take ownership of this DAG: validating it, rejecting cycles, checking permissions, assigning task identifiers, and freezing it into an executable, immutable version.[9]

DAGs are insufficient for highly interactive workflows, indefinite monitoring, stateful conversations, or processes where the next step is completely unknown until the current step finishes (e.g., exploratory debugging). In these cases, State Machines and event-sourced workflows (like LangGraph executing over Temporal) provide the necessary cyclical flexibility.[9]

Long-Running and Resumable Execution

Agent workflows span hours, days, or weeks (e.g., a research fleet compiling a longitudinal study). Relying on one long model conversation thread or one continuous process is fatal; if the container restarts, all context is lost.[2] **Durable execution** abstracts this risk. The orchestrator

persists the workflow state after every meaningful transition to an event-sourced ledger.[6] Upon a worker crash, orchestrator restart, or model provider outage, the workflow runtime performs a deterministic replay. It rebuilds the state from the ledger, skipping already-completed LLM calls and tool executions, and resumes execution seamlessly.[2] Because true exactly-once execution is impossible in distributed systems, idempotency keys and deduplication ensure that side effects (like sending an email) occur at-most-once, while state transitions are processed effectively exactly-once.[2]

Control Plane Versus Execution Plane Design

A precise separation of responsibilities guarantees that an orchestration system remains secure, performant, and immune to prompt injection altering core logic.[3]

Control-Plane Responsibilities

The control plane must exclusively own system governance. It manages the agent registry, fleet registry, workflow definitions, and run state. It handles infrastructure scheduling, queue management, priority sorting, dependency resolution, concurrency control, and rate limiting. It enforces token budgets, model/provider routing policies, and security credentials.[1] Furthermore, it maintains the authoritative audit log, handles retries and escalations, and manages human approval status.

Execution-Plane Responsibilities

The execution plane (worker nodes) owns the probabilistic generation. Workers execute model calls, tool calls, code, and browser interactions. They manage local scratch state and task-specific context. Workers produce result payloads and structured status updates, emitting heartbeats to the control plane.

The Boundary and Interface Contract

The cardinal rule is: *Data reported by an agent must never be trusted merely because the agent reported it.* An agent may *request* an action (e.g., AgentSpawnRequested), but the deterministic control plane *authorises* it. An execution environment *performs* the action, a verifier *confirms* the output against schema, and the run ledger *records* the state transition. This explicit contract prevents a hallucinating LLM from updating the official run state to "Complete" while leaving tasks unfinished.

Queue, Scheduling, Priority, and Budget Design

Fleet management requires production-grade task queues to decouple task generation from execution capacity.

Queue Design and Priorities

Tasks should be segregated. Global queues handle cross-fleet routing, while fleet-specific and capability-specific queues route work to specialized environments (e.g., a GPU-enabled queue

for visual production). Dead-letter queues (DLQs) capture persistently failing tasks, and manual-review queues hold tasks requiring human intervention.
Priority strategies must balance urgency with fairness. Strict fixed priorities cause *priority inversion* and *starvation*, where low-priority tasks hold database locks or simply never execute because high-priority tasks flood the system.[21] This is prevented via aging mechanisms (increasing priority over time), weighted fair queuing, and tenant quotas.

Resource Governance

Autonomous agents require strict resource constraints.[1] The control plane enforces maximum parallel agents per fleet, per-model concurrency, and API rate limits. Financial budgets are critical; a scheduling strategy must evaluate the estimated token cost of a task against the fleet's remaining budget before dispatch. If a task requires an expensive model (e.g., Claude 3.5 Sonnet) but the budget is depleted, the scheduler delays the task or routes it to a cheaper fallback model.

Dynamic Subagent Lifecycle

Systems where agents can dynamically create subagents enable powerful task delegation but introduce severe operational risks, notably the "infinite loop of death" characterized by runaway recursive spawning and budget exhaustion.[4]

Lifecycle and Authorization

An agent never forks a process directly. It submits an AgentSpawnRequested command to the control plane.[4] The policy engine determines authorization based on recursion depth limits (e.g., blocking spawning past 3 levels deep) and parent-child budget propagation. The conservation law of agent contracts dictates that a parent agent's token/time budget must be mathematically subdivided and passed to the child; a child cannot be allocated more resources than the parent possesses.[1]

Registration and Expiration

Once authorized, the control plane instantiates the temporary subagent instance from a reusable worker template, assigning it heavily scoped credentials and a restricted tool allowlist (often explicitly removing the subagent's ability to spawn further agents).[4] To detect orphaned subagents and prevent uncontrolled multiplication, every subagent is assigned a hard cryptographic lifespan (lease). Upon forced expiration, the control plane terminates the container, cleans up temporary local memory, and reports a timeout failure to the parent.

Human Approval Checkpoints

Irreversible actions—such as merging code to production, executing financial transactions, modifying public databases, or interacting with external clients—mandate human-in-the-loop (HITL) checkpoints.[13]

Design and Implementation

A robust workflow cannot simply halt a script and block a thread while waiting for standard input; humans may take hours or days to respond, violating network timeouts and wasting compute. Instead, the workflow must pause durably.

1. The execution plane submits an ApprovalRequired event.
2. The orchestrator records the state, terminates the active worker process, and releases compute resources.[2]
3. The control plane notifies the designated reviewer, presenting the exact proposed action (e.g., a specific SQL diff) and supporting evidence.
4. The system waits indefinitely. Upon human action, the control plane injects a deterministic Signal back into the workflow, fetching the dormant state and safely resuming execution based on the approval or rejection.[2]
5. To mitigate approval fatigue, routing is risk-based: low-risk actions utilize optional sampled review or exception-based approval, while high-risk model outputs trigger mandatory, sometimes dual, approval.

Deterministic Workflows Versus Autonomous Agents

System architects must navigate a fundamental decision framework to prevent over-engineering: determining when to use ordinary deterministic code versus when to employ autonomous language-model loops.

The Decision Framework

The guiding principle is: *Models may recommend actions, but deterministic systems should authorise, schedule, execute, and record consequential state transitions.*

- **Strictly Deterministic**: Permission checks, state machine transitions, queue operations, financial limits, retries, timeouts, audit logging, schema validation, and credential handling. Replacing any of these with an LLM introduces catastrophic security and reliability vulnerabilities.
- **Model Reasoning Valuable**: Interpreting ambiguous requests, decomposing novel goals, synthesizing unstructured research, generating hypotheses, and adapting explanations. Here, the probabilistic nature of LLMs is a feature, not a bug.

Edge Case Qualification: When parsing highly unstructured, noisy log data to determine if a state transition *should* occur, deterministic rules may fail. Here, an LLM acts as an evaluator, but its output must conform to a strict JSON schema (e.g., {"status": "failed", "confidence": 0.9}) which the deterministic rules engine then processes to execute the actual transition.

Avoiding the Fragile “Super-Agent”

A single highly capable orchestrator agent—acting as a monolithic prompt that plans, routes, remembers, and executes everything—is a severe anti-pattern.[23]

Risks of the Super-Agent

As the super-agent operates, its context window fills with tool outputs, conversational tangents, and complex state histories. This leads to excessive prompt complexity, catastrophic forgetting, hallucinated task statuses, and an inability to isolate faults. Testing becomes impossible because a failure in execution corrupts the reasoning required for routing. Token usage scales exponentially, and centralized failure means one malformed API response brings down the entire system.

Architectural Alternatives and Hierarchical Compression

The solution requires architectural decoupling: a deterministic orchestration core, separate LLM planners, local fleet supervisors, and capability-specific workers. The top-level control plane coordinates multiple fleets while remaining entirely unaware of task-level semantic details through **hierarchical compression**. Workers execute tasks and return rich, structured JSON results to supervisors. Supervisors compress these results into fleet-level summaries and metrics. The global control plane receives only the highest-level operational metadata: status, risk, dependencies, budget use, and escalation flags.[12] This compression prevents context growth while retaining absolute operational control.

Multi-Fleet Orchestration

Enterprise deployment requires managing several heterogeneous fleets (e.g., software-development, deep-research, creative-production) simultaneously.

Fleet Isolation

Fleets must be strictly isolated to prevent security breaches and context contamination. Isolation occurs via namespaces, dedicated queues, and IAM permissions. A software agent operates in an environment lacking the network routes, credentials, and tool allowlists to interact with the legal fleet's document store. Memory stores are segregated; a visual-production agent cannot query the episodic memory of a competitor-monitoring agent.

Cross-Fleet Collaboration

Fleets collaborate not through unrestricted shared memory, but via explicit, schema-validated contracts.[24] If a strategy fleet requires research, it submits a TaskRequested event payload to the central control plane. The control plane routes this to the research fleet's queue. Upon completion, the artefact is placed in a secure artefact store, and a DependencySatisfied event alerts the strategy fleet. The global orchestrator communicates exclusively with fleet supervisors via these events, never directly manipulating individual workers.

Reliability and Failure Management

AI agents fail in fundamentally different ways than deterministic software, necessitating a specialized failure taxonomy and recovery strategies.

Failure Taxonomy and Containment

- **Infrastructure Failures**: Worker crashes, network partitions, unavailable providers. Handled deterministically via durable execution retries, exponential backoff, and circuit breakers.[2]
- **Semantic Failures**: Malformed model outputs, hallucinated completions, infinite loops, and runaway agent spawning.[4] Handled via strict schema validation, step-count circuit breakers, semantic convergence detection, and dead-letter queues.
- **State Failures**: Conflicting writes, partial completions, stale task leases. Handled via idempotency keys, deduplication, and optimistic concurrency locks.

Compensation and the Saga Pattern

Retrying an AI agent differs from retrying a deterministic function because the agent may generate entirely different logic on the second attempt. If a multi-step workflow fails midway after side-effects have occurred (e.g., an email was sent), true rollback is impossible. The system must utilize the Saga pattern, invoking predefined Compensating Actions (e.g., an agent drafting and sending a follow-up correction email) to restore operational consistency.[19]

State, Messages, and Communication Contracts

Free-form natural-language messages cannot serve as the sole source of operational truth. A structured communication model is mandatory for reliable state management.

Structured Communication Model

Communication between orchestrator, supervisors, and workers utilizes strongly typed JSON event envelopes containing correlation IDs, causation IDs, run IDs, task IDs, and idempotency keys.[2] This cryptographic lineage enables perfect trace reconstruction.

Standard Message Types

- TaskRequested: Contains capability requirements, budgets, and input artefacts.
- TaskStarted: Issued when a worker claims a lease.
- Heartbeat: Emitted periodically to maintain the lease.
- ApprovalRequired: Triggers durable pausing for human intervention.
- TaskCompleted: Contains output schemas and references to the artefact store.
- AgentSpawnRequested: A request to delegate work to a subagent, pending budget validation.

Agent Lifecycle Management

Defining a complete lifecycle for agents clarifies the distinction between persistent identities and temporary compute.

The Lifecycle

1. **Template Definition**: An immutable blueprint of capabilities, instructions, and limits.
2. **Provisioning/Activation**: The control plane provisions a temporary runtime environment (e.g., isolated container, Git worktree) and injects scoped credentials.
3. **Execution/Heartbeat**: The agent executes, emitting health verifications.
4. **Suspension/Termination**: Upon task completion or budget exhaustion, the runtime is destroyed.
5. **Archival**: Ephemeral scratch data is wiped, while meaningful outputs update the agent's persistent episodic memory.

A "persistent agent" in production does not mean a continuously running while loop burning compute. It means a persistent database identity, with durable memory and state records, that is activated inside an ephemeral worker container only when a relevant event requires attention.

Scheduling and Supervision

Operational supervision of agent fleets requires mechanisms to differentiate between a live process and actual semantic progress.

Supervision Mechanisms

A process being alive (emitting OS-level metrics) does not mean an agent is making progress. An agent claiming progress (via a status string) does not mean it is producing verifiable outputs. Therefore, supervision relies on explicit task claiming (leases), PING/PONG liveness checks, and semantic health verification.[5]

Detecting Stalled or Runaway Agents

Detecting agents that are stuck or repeating themselves requires tracking expected-progress windows and implementing step-count circuit breakers. If an agent loops 15 times without generating a verifiable output artefact, a semantic convergence detector (comparing the cosine similarity of recent thought traces) flags the loop, triggers a dead-letter escalation, and revokes the task lease, killing the worker.[5]

Observability and Fleet Operations

Operating concurrent fleets requires extensive observability to reconstruct execution paths and audit financial burn.

Required Observability

A fleet dashboard must track queue depth, active/idle workers, retry counts, token use, provider use, estimated vs. actual cost, and human-approval delays. To reconstruct *why* an action occurred, traces must link the initial workflow trigger to the routing decision, the specific prompt injected, the model version used, the tools called, and the raw API responses.[6]

Status Model

Tasks transition through a strict deterministic state machine: queued -> eligible -> assigned -> starting -> running -> waiting (for dependencies) -> blocked (for human approval) -> retry scheduled -> completed / failed / cancelled / expired.

Security and Governance

Security boundaries must restrict the inherent unpredictability of language models.

Boundaries and Validation

Agent orchestration relies on the principle of least privilege. Fleets operate under strict IAM role segregation. Agents utilize credential brokering—swapping temporary, scoped references for short-lived tokens precisely at the moment of tool execution—rather than holding raw secrets in memory. Environments are sandboxed with network egress restrictions and explicit tool allowlists to contain prompt-injection attacks. Critically, an AI planner's recommendation for a task must not automatically grant the executor new permissions; the deterministic policy engine validates the generated workflow against pre-approved capability mappings before execution.

Model-Independent and Provider-Independent Orchestration

The control plane must remain agnostic to underlying language models to prevent vendor lock-in and optimize latency/cost.

Abstraction via Model Gateway

A Model Gateway acts as the abstraction layer, intercepting structured request contracts and normalizing them into provider-specific formats (e.g., translating OpenAI tool-calling schemas to Anthropic formats).[18] The gateway handles capability-based routing, failing over automatically if a provider experiences an outage or rate-limit exhaustion. Thus, an agent is defined abstractly by its role, capabilities, instructions, output schema, and budget. The identical agent persona can be executed by Claude 3.5 Sonnet during complex tasks and Claude 3.5 Haiku during simple tasks, dynamically selected by the gateway without altering the surrounding orchestration logic.

Comparison of Implementation Technologies

Not all frameworks serve the same architectural role. Selecting the correct layer is critical.

Technology Category	Examples	Solves	Fails To Solve / Limitations	Suitability for Control Plane
Durable	Temporal, AWS	Exactly-once	Does not	**Ideal**. Acts as

Workflow Engines	Step Functions [2]	execution, state persistence, automatic retries, durable human-pauses.	natively provide LLM integration or prompt management out-of-the-box.	the ultimate deterministic core.
Data & DAG Orchestration	Apache Airflow, Dagster [9]	Static scheduled pipelines, tracking data lineage.	Inflexible for highly dynamic, cyclical agent reasoning loops requiring sub-second routing.	Poor for dynamic multi-agent loops.
Distributed Compute	Celery, Ray, K8s Jobs	Infrastructure-level task distribution, horizontal scaling.	Does not inherently track complex workflow state transitions or manage conversational memory.	Insufficient without custom state management.
Agent Frameworks	LangGraph, AutoGen, CrewAI [9]	Defining internal execution state machines, cyclic reasoning loops, tool integration.	Lack infrastructure-level durability without external databases; process crash loses state.	Serves the Execution Plane, not the Global Control Plane.
Messaging Infrastructure	Kafka, RabbitMQ	High-throughput asynchronous data planes, event buses.	Requires building a custom policy engine and state machine	Serves the Data Plane.

			from scratch.	

Proposed Multi-Fleet Agent Control Plane

The reference architecture relies on a strict separation of concerns, utilizing a durable workflow engine (like Temporal) as the Control Plane and a graph-based agent framework (like LangGraph) as the Execution Plane.[2]

Component Specification

1. **Control-Plane API**: (Deterministic) Ingress for all requests. Scales horizontally.
2. **Registries (Fleet/Template/Capability)**: (Deterministic) SQL databases tracking configurations.
3. **Scheduler & Dispatcher**: (Deterministic) Evaluates dependencies, enforces rate limits, pushes tasks to Temporal queues.
4. **Policy & Budget Manager**: (Deterministic) Enforces token limits and recursion depths.[1]
5. **Model Gateway**: (Deterministic) Routes normalized LLM requests to providers, handling failovers.[18]
6. **Worker Runtime**: (Probabilistic/Hybrid) Ephemeral containers running LangGraph state machines.
7. **State & Event Log**: (Deterministic) Append-only ledger of all execution histories.[16]
8. **Artefact Store**: (Deterministic) S3/Blob storage for durable agent outputs.
9. **Human-Approval Service**: (Deterministic) Intercepts ApprovalRequired events, pauses Temporal workflows, and injects resuming signals.[25]
10. **Evaluator Layer**: (Hybrid) Specialized models grading outputs against rubrics.[13]
11. **Secrets Broker**: (Deterministic) Swaps credential references for short-lived tokens.
12. **Dead-Letter Service**: (Deterministic) Traps malformed outputs for manual review.

Architecture Diagrams

Diagram 1: System-Level Architecture

Code snippet

```
graph TD
  API[Control-Plane API] --> Orchestrator
  Orchestrator --> Policy
  Orchestrator --> EventLog[(Event Log)]
  Orchestrator --> Q1
  Orchestrator --> Q2

  Q1 --> S1
```

```
  S1 --> W1
  S1 --> W2

  Q2 --> S2
  S2 --> W3

  W1 -.-> Gateway[Model Gateway]
  W2 -.-> Gateway
  W3 -.-> Gateway

  Gateway --> OpenAI
  Gateway --> Anthropic

  W1 --> Artefact
```

Diagram 2: Control-Plane Versus Execution-Plane Boundary

```
Code snippet
graph LR
  subgraph Control Plane
    Scheduler
    Dispatcher
    PolicyEngine
    EventLog
  end

  subgraph Execution Plane
    WorkerAgent
    LangGraphLoop
    ToolExecutor
  end

  Dispatcher -- "Task Envelope (Lease granted)" --> WorkerAgent
  WorkerAgent -- "Structured JSON Result" --> PolicyEngine
  PolicyEngine -- "Commit State" --> EventLog
```

Diagram 3: Task Lifecycle Sequence

```
Code snippet
```

```
sequenceDiagram
    participant Orchestrator
    participant Queue
    participant Worker
    participant ModelGateway
    participant Human

    Orchestrator->>Queue: Enqueue Task
    Worker->>Queue: Claim Task (Lease Start)
    Worker->>ModelGateway: LLM Reasoning Loop
    Worker-->>Orchestrator: Heartbeat
    Worker->>Orchestrator: ApprovalRequired Event
    Orchestrator->>Orchestrator: Hibernate Workflow State
    Orchestrator->>Human: Notify Reviewer
    Human->>Orchestrator: Signal Approval
    Orchestrator->>Worker: Resume Task
    Worker->>Orchestrator: TaskCompleted (Result)
```

Diagram 4: Dynamic Subagent Creation

```
Code snippet
sequenceDiagram
    participant ParentWorker
    participant ControlPlane
    participant PolicyEngine
    participant SubagentWorker

    ParentWorker->>ControlPlane: AgentSpawnRequested (Task X, Est. Cost Y)
    ControlPlane->>PolicyEngine: Verify Depth Limit & Budget Conservation
    PolicyEngine-->>ControlPlane: Authorized
    ControlPlane->>ControlPlane: Instantiate Ephemeral Identity
    ControlPlane->>SubagentWorker: Dispatch Task
    SubagentWorker->>ControlPlane: TaskCompleted
    ControlPlane->>ControlPlane: Terminate Subagent
    ControlPlane->>ParentWorker: DependencySatisfied
```

Diagram 5: Multi-Fleet Collaboration

```
Code snippet
```

```
sequenceDiagram
    participant StrategyFleet
    participant Orchestrator
    participant ResearchFleet
    participant ArtefactStore

    StrategyFleet->>Orchestrator: TaskRequested (Research Market)
    Orchestrator->>ResearchFleet: Dispatch Task
    ResearchFleet->>ArtefactStore: Save PDF Report
    ResearchFleet->>Orchestrator: TaskCompleted (Artefact URI)
    Orchestrator->>StrategyFleet: DependencySatisfied (URI)
```

Diagram 6: Failure and Resumption (Durable Execution)

```
Code snippet
sequenceDiagram
    participant WorkerNode1
    participant TemporalOrchestrator
    participant EventHistory
    participant WorkerNode2

    WorkerNode1->>TemporalOrchestrator: Complete Step 1
    TemporalOrchestrator->>EventHistory: Persist Step 1 Result
    Note over WorkerNode1: Hardware Crash
    TemporalOrchestrator->>TemporalOrchestrator: Lease Timeout
    TemporalOrchestrator->>WorkerNode2: Reassign Task
    WorkerNode2->>EventHistory: Replay History
    EventHistory-->>WorkerNode2: Return Step 1 Result (No Re-execution)
    WorkerNode2->>TemporalOrchestrator: Complete Step 2
```

Data-Model Requirements

Schema Entity	Key Fields	Purpose
Fleet	fleet_id, name, supervisor_id, allowed_capabilities, allowed_models, budgets, concurrency_limits	Namespaces agents, enforces isolation, and sets aggregate limits.

Agent Template	template_id, role, instructions_ref, tool_permissions, model_reqs, output_schema, timeout	Immutable blueprint defining an agent's authorized capabilities.
Agent Instance	instance_id, template_id, fleet_id, parent_agent, status, assigned_task, expiration	A live, bounded execution identity spawned from a template.
Workflow Run	run_id, workflow_version, fleet, objective, status, checkpoint, budget_allocated, trace_id	The overarching lifecycle and state of a client request.
Task	task_id, run_id, dependencies, capability_reqs, priority, lease_expires, input/output_artefacts	The atomic unit of execution routed to queues and workers.
Event	event_id, event_type, run_id, causation_id, timestamp, schema_version, payload	Immutable ledger of state transitions powering observability.

Algorithms and Pseudocode

The control plane must execute core algorithms deterministically.

Python
```
# DETERMINISTIC ORCHESTRATION LOGIC

def request_temporary_subagent(parent_task_id, capability_req, est_cost):
    """Handles controlled dynamic agent spawning to prevent runaway recursion."""
    parent = db.get_agent_instance_by_task(parent_task_id)
    fleet_policy = db.get_fleet_policy(parent.fleet_id)
```

```
    # Check recursion depth limit
    if parent.depth >= fleet_policy.max_agent_depth:
        return raise_error("Max recursive depth exceeded")

    # Check conservation laws (Budgets)
    if parent.remaining_budget < est_cost:
        return raise_error("Insufficient parent budget for delegation")

    # Authorize and Provision
    sub_id = generate_instance_id()
    deduct_budget(parent.agent_id, est_cost)
    allocate_budget(sub_id, est_cost)

    # Dispatch Task
    task = create_task(sub_id, capability_req, parent_task_id)
    enqueue_task(task)
    return success(task.task_id)

def evaluate_task_eligibility(task_id):
    """Dependency resolution algorithm."""
    task = db.get_task(task_id)
    for dep in task.dependencies:
        dep_task = db.get_task(dep)
        if dep_task.status!= "COMPLETED":
            return False
    return True

def handle_approval_checkpoint(workflow_run_id, proposed_action):
    """Durable pausing for human-in-the-loop."""
    workflow = db.get_workflow(workflow_run_id)
    workflow.status = "BLOCKED_PENDING_APPROVAL"
    workflow.save_checkpoint()

    notify_human_reviewer(proposed_action)

    # Suspend execution context entirely to save compute
    suspend_process(workflow_run_id)

def resume_from_signal(workflow_run_id, human_decision):
    """Resumes durably paused workflow."""
    workflow = restore_from_checkpoint(workflow_run_id)
    if human_decision == "APPROVED":
```

```
    workflow.status = "RUNNING"
    enqueue_next_step(workflow)
  else:
    workflow.status = "CANCELLED"
    execute_compensating_actions(workflow)
```

Practical Scenarios

Scenario A: Software-Development Fleet A Planner agent dynamically divides a feature request into three independent coding tasks. A deterministic dependency resolver ensures tasks are parallelized. Three Coding agents execute concurrently inside ephemeral, isolated Git worktrees to prevent shared-file corruption. A Testing agent verifies the merged codebase. Upon test success, the workflow enters a durable pause. A human architect receives a webhook, reviews the PR diff, and signals approval, triggering the deterministic merge and deploy scripts.

Scenario B: Deep-Research Fleet A Router evaluates a complex query and triggers a map-reduce DAG. Ten specialized research agents investigate independent subtopics in parallel, bounded by strict 5-minute timeouts. A deterministic Supervisor ensures all ten tasks resolve (handling failures via retries to backup models). Once aggregated, a Synthesis agent combines the findings, and a deterministic Citation Verifier script validates all URLs before returning the final artefact.

Scenario C: Competitor-Monitoring Fleet Continuous polling is avoided. A deterministic cron scheduler wakes the system daily, triggering simple Python scripts to inspect public RSS feeds. Only if the deterministic script detects a meaningful string difference does it emit an event that instantiates an Analytical agent. The agent synthesizes the change, saves a report to the artefact store, and immediately terminates, yielding zero idle compute cost.

Scenario D: Creative-Production Fleet Strategy, Copy, and Image-Prompt agents collaborate sequentially. The DAG explicitly enforces that the Copy agent's output must pass an Evaluator agent's strict brand-guideline rubric. If it fails three times, it routes to a human-review DLQ. Final image generation passes through a mandatory human approval checkpoint before being published to external social media platforms.

Scenario E: Cross-Fleet Workflow The Strategy Fleet requires deep market data. It submits a standardized TaskRequested event payload to the Research Fleet's queue. The Strategy workflow durably hibernates. The Research Fleet orchestrates its internal workers, stores the final PDF in S3, and emits a DependencySatisfied event. The control plane wakes the Strategy workflow, passing it the secure S3 URI to resume execution.

Anti-Patterns

Anti-Pattern	Why it seems attractive	How it fails / Symptoms	Safer Replacement

One super-agent	Easiest to prototype via a single massive prompt.	Context window collapse; catastrophic forgetting; system halts on one API error.	Hierarchical deterministic orchestration of specialized micro-agents.[23]
Unlimited agent spawning	Solves complex problems through rapid, emergent delegation.	"Infinite loop of death"; massive API bills; 50+ levels of runaway recursion.[4]	Strict depth limits and budget conservation constraints.[1]
Natural-language routing	Avoids defining strict schemas.	Upstream hallucinations break downstream parsing, causing systemic cascades.	Strongly typed JSON schemas and formal event envelopes.
No durable state	Simplifies local development architecture.	Process crashes wipe out hours of agent progress; no capability to wait for humans.[2]	Event-sourced durable execution via Temporal or Step Functions.[6]
Shared memory stores	Agents can easily read each other's context.	Cross-tenant data leakage; agents overwrite or hallucinate based on irrelevant data.	Strict fleet namespaces and explicit payload passing.
Monolithic model usage	Simplifies API keys (e.g., using GPT-4 for everything).	Burns vast financial budgets on trivial formatting or routing tasks.	Model gateway with capability-based routing to smaller/cheaper models.[18]
Trusting self-reported state	LLMs naturally output "Task Complete."	Agent hallucinates completion without actually invoking the necessary tools.	Deterministic verifiers inspecting the actual state of the environment.

Missing trace IDs	Faster initial setup without tracking metadata.	Impossible to debug why an agent made a decision three steps ago.	Enforced correlation IDs on all events.
No idempotency keys	Assumes APIs handle duplicates safely.	Agent retries cause double-charging of credit cards or duplicate emails.[9]	Deterministic deduplication at the orchestrator boundary.
Infinite reflection loops	Allows agents to autonomously fix their own errors.	Agent gets stuck oscillating between two failed states permanently.[5]	Semantic convergence detection and step-count circuit breakers.
Unrestricted tool access	Agents figure out what they need dynamically.	Prompt injection leads to catastrophic system compromise and data exfiltration.	Ephemeral credentials and explicit, scoped tool allowlists.
Shared working directory	Easy to pass files between agents.	Concurrent agents overwrite the same files, causing massive state corruption.	Ephemeral, isolated containers and explicit artefact hand-offs.
Autonomous irreversible actions	Fully automates workflows end-to-end.	Agent sends hallucinated emails to key clients or drops production databases.	Mandatory human approval checkpoints.[13]
Unvalidated AI plans	Leverages model creativity for scheduling.	Planner hallucinates circular dependencies, locking up the	Deterministic DAG validation before execution.[9]

		queue.	
Orchestration in prompts	"If X fails, try Y" written in the system prompt.	The model forgets the instruction; cannot enforce timeouts or backoffs.	Deterministic state-machine logic in the control plane.
Over-using agents	Sounds cutting-edge to have "Agentic AI."	Wastes $0.50 and 30 seconds on a task a 5ms Python script could do perfectly.	Default to deterministic scripts; use LLMs only for required probabilistic reasoning.

Testing Strategy

Testing the orchestration system must be entirely decoupled from testing the quality of the underlying LLM models. The orchestrator must be validated for its structural resilience.

- **Deterministic Fake Agents**: Utilize mock worker containers that do not call external APIs. Instead, they return predefined JSON payloads (including simulated errors) to test the control plane's routing, queue management, and retry logic without incurring token costs.
- **Workflow Replay Tests**: Force a worker node to crash mid-execution. Assert that the durable orchestrator reassigns the task, replays the event history, and resumes execution seamlessly without duplicating idempotency-protected database writes.[2]
- **Agent-Spawning-Limit Tests**: Deploy a malicious mock agent programmed to continually request subagents. Assert that the control plane's policy engine severs the loop exactly at the configured depth or token limit.[4]
- **Chaos Engineering**: Introduce random network partitions, out-of-order events, and delayed heartbeats to validate lease expirations, dead-letter routing, and graceful degradation.

Evaluation Scorecard

Criteria	Single LLM Super-Agent	Decentralised Swarm	Central Deterministic Orchestrator	Hierarchical Hybrid (Proposed)
Reliability/Res	Very Poor	Poor	Excellent	Excellent

umability				
Scalability (Fleet size)	Low	High	Medium	Very High
Security & Isolation	Poor	Poor	Excellent	Excellent
Auditability/Observability	Low	Low	Very High	Very High
Model Independence	Low	Low	High	High
Cost Control / Budgets	Very Poor	Very Poor	High	Excellent
Human Oversight (HITL)	Difficult	Difficult	Built-in	Built-in
Implementation Complexity	Low	High	High	Very High
Prevents Runaway Autonomy	No	No	Yes	Yes

Recommended Implementation Roadmap

Building a multi-fleet agentic system requires a staged progression, avoiding premature abstraction.

- **Stage 1: Deterministic Single-Fleet Foundation**: Establish the core task schema, queues, and worker runtimes. Do not implement models yet. Build standard logs, retries, and timeouts. *Exit criteria*: Ability to reliably execute and monitor standard Python scripts.
- **Stage 2: Durable Workflows**: Integrate Temporal or AWS Step Functions. Implement persistent state, event sourcing, dependency resolution, and idempotency keys. *Exit criteria*: Workflows survive forced worker crashes and resume perfectly.

- **Stage 3: Agent Specialisation**: Introduce LangGraph for localized worker execution. Build the capability registry, prompt templates, and the model gateway. *Exit criteria*: Isolated agents can execute distinct reasoning loops and interact with basic tools.
- **Stage 4: Human Governance**: Implement durable pausing mechanisms, human approval webhooks, strict permissions, token budgets, and the audit trail.
- **Stage 5: Multi-Fleet Control**: Establish namespace boundaries. Deploy fleet supervisors and implement cross-fleet communication via explicit event contracts. Establish the central observability dashboard.
- **Stage 6: Dynamic Subagents**: Implement the AgentSpawnRequested command. Enforce recursion depth limits, budget conservation laws, parent-child tracing, and automatic cryptographic expiry.[1]
- **Stage 7: High Availability and Scale**: Deploy replicated control-plane services, distributed Kafka queues for the data plane, multi-region failover, and advanced priority scheduling to prevent starvation.

Final Design Decision

For a small team or advanced builder managing several agent fleets, the optimal, production-ready architecture is the **Hierarchical Hybrid Control Plane**, leveraging a durable engine like Temporal for the deterministic core and a graph framework like LangGraph for localized agent execution.[9]

Resolution of Core Questions:

- *Should the global orchestrator use an AI model?* **No.** It must remain a purely deterministic state machine (Go/Java/Python) tracking envelopes, budgets, and state transitions to prevent context collapse and hallucinated scheduling.[9]
- *Which functions remain deterministic?* Workflow state, permissions, queues, token budgets, retries, audit logging, and dependency resolution.
- *Should supervisors be persistent processes?* **No.** Supervisors should be persistent *identities* (database records with episodic memory) activated on-demand via event-driven triggers to conserve compute.
- *How should agents communicate?* Exclusively through schema-validated JSON event contracts routed by the control plane, not unrestricted natural-language chats.[24]
- *How should dynamic subagents be controlled?* Agents submit spawn requests to the control plane, which enforces strict recursion depth limits and budget conservation before provisioning an ephemeral container.[1]
- *How should long-running workflows be resumed?* Via event-sourced durable execution. The state is recorded at every transition, allowing deterministic replay after a crash.[2]
- *How should human approvals be implemented?* The workflow durably hibernates, releasing compute resources, and resumes only upon receiving an external cryptographic signal.[2]
- *Which parts to build in-house vs outsource?* Do not build the durable workflow engine or queues; utilize Temporal, Step Functions, or existing robust infrastructure. Build the domain-specific policy engines, model gateways, and agent prompts in-house.

Final Recommended Principles:

- Deterministic control, probabilistic reasoning.
- Persistent state, ephemeral execution.
- Explicit contracts, not unrestricted conversation.
- Bounded autonomy, not unlimited loops.
- Local fleet supervision, global policy control.
- Event-driven activation, not continuous polling.
- Model-independent agent definitions.
- Artefacts and events as the operational source of truth.
- Verification before completion.
- Human approval before consequential actions.

Open Questions and Unresolved Trade-Offs

While the hierarchical hybrid model maximizes reliability, significant unresolved trade-offs persist in the 2026 landscape. First, managing semantic memory across deeply isolated fleets remains a friction point; overly strict namespaces hinder organizational knowledge sharing, while centralized vector databases risk cross-tenant context leakage and prompt injection. Second, establishing accurate pricing and scheduling topologies for dynamically spawned subagents is difficult when utilizing external models with highly variable latency and token generation rates. Finally, defining the optimal boundary between a complex deterministic DAG defined by a developer and a generalized plan generated by a Planner agent remains a fluid engineering challenge, heavily dependent on the continued maturation of model reasoning capabilities and adherence to structured JSON schemas.

Works cited

1. Agent Contracts: A Formal Framework for Resource-Bounded Autonomous AI Systems (Full), accessed July 25, 2026, https://arxiv.org/html/2601.08815v1
2. Durable Execution for LLM Agents 2026: Temporal + LangGraph - AppScale Blog, accessed July 25, 2026, https://appscale.blog/en/blog/durable-execution-llm-agents-temporal-langgraph-checkpointing-2026
3. What is agent orchestration? Meaning, Architecture, Examples, Use ..., accessed July 25, 2026, https://aiopsschool.com/blog/agent-orchestration/
4. General-purpose sub-agents recursively spawn unbounded child agents, causing exponential fan-out and massive token burn · Issue #68110 · anthropics/claude-code - GitHub, accessed July 25, 2026, https://github.com/anthropics/claude-code/issues/68110
5. The $50,000 Loop: How to Stop Runaway AI Agent Costs | CloudAtler Blog, accessed July 25, 2026, https://cloudatler.com/blog/the-50-000-loop-how-to-stop-runaway-ai-agent-costs
6. Agentic AI Workflows: Why Orchestration with Temporal is Key | IntuitionLabs, accessed July 25, 2026,

https://intuitionlabs.ai/articles/agentic-ai-temporal-orchestration
7. Who are Workato? - Viewpoint Analysis, accessed July 25, 2026, https://www.viewpointanalysis.com/post/who-are-workato
8. Agent Contracts: A Formal Framework for Resource-Bounded Autonomous AI Systems - arXiv, accessed July 25, 2026, https://arxiv.org/pdf/2601.08815
9. Kinde Orchestrating Multi-Step Agents: Temporal/Dagster ..., accessed July 25, 2026, https://kinde.com/learn/ai-for-software-engineering/ai-devops/orchestrating-multi-step-agents-temporal-dagster-langgraph-patterns-for-long-running-work/
10. Anthropic: Building Effective Agents: Practical Framework and Design Principles - ZenML, accessed July 25, 2026, https://www.zenml.io/llmops-database/building-effective-agents-practical-framework-and-design-principles
11. Building Effective AI Agents \ Anthropic, accessed July 25, 2026, https://www.anthropic.com/engineering/building-effective-agents
12. Agent Looping 2026: Prompts → Loops → Orchestrated Teams - AppScale Blog, accessed July 25, 2026, https://appscale.blog/en/blog/agent-looping-prompt-engineering-loop-engineering-orchestrated-agents-2026
13. https://www.anthropic.com/engineering/building-effective-agents - NAITIVE AI Consutling Agency Blog, accessed July 25, 2026, https://blog.naitive.cloud/building-effective-agents/
14. Contract Net Protocol - Wikipedia, accessed July 25, 2026, https://en.wikipedia.org/wiki/Contract_Net_Protocol
15. AI agent frameworks: A developer's guide to production, accessed July 25, 2026, https://tyk.io/learning-center/ai-agent-frameworks-a-developers-guide-to-production/
16. Temporal with AI: Durable Orchestration for Intelligent Systems - Medium, accessed July 25, 2026, https://rzaeeff.medium.com/temporal-with-ai-cfb7bb1ae0ed
17. Temporal: Durable Execution Solutions, accessed July 25, 2026, https://temporal.io/
18. What is prompt engineering? Meaning, Architecture, Examples, Use, accessed July 25, 2026, https://aiopsschool.com/blog/prompt-engineering/
19. Saga Pattern - Upsonic, accessed July 25, 2026, https://upsonic.ai/lexicon/saga-pattern
20. AI Agent Orchestration Framework - Suprmind, accessed July 25, 2026, https://suprmind.ai/hub/insights/ai-agent-orchestration-framework/
21. The Priority Inversion Problem: Ensuring Urgent Agent Work, accessed July 25, 2026, https://tfsfventures.com/blog/the-priority-inversion-problem-ensuring-urgent-agent-work-preempts-routine-tasks
22. priority queue design for agent email processing — LobsterMail, accessed July 25, 2026, https://lobstermail.ai/blog/priority-queue-design-for-agent-email-processing

23. Building effective agents - Simon Willison's Weblog, accessed July 25, 2026, https://simonwillison.net/2024/Dec/20/building-effective-agents/
24. Agent Contracts: A Formal Framework for Resource-Bounded Autonomous AI Systems (Full)Accepted for oral presentation at COINE 2026 (16th International Workshop on Coordination, Organizations, Institutions, Norms and Ethics for Governance of Multi-Agent Systems), co-located with AAMAS 2026, Paphos, Cyprus. - arXiv, accessed July 25, 2026, https://arxiv.org/html/2601.08815v3
25. Build an AI Agent with Temporal - YouTube, accessed July 25, 2026, https://www.youtube.com/watch?v=GEXIlEH2XiQ