Cross-Vendor Memory, Task Continuity, and Agent Session Recovery

1. Executive Summary

As artificial intelligence agents transition from human-supervised, interactive copilots to fully autonomous infrastructure entities, the operational paradigm governing their execution must undergo a fundamental architectural shift. The contemporary reliance on provider-specific session states—such as an OpenAI thread, a Claude conversation, or a localized model context window—introduces severe reliability bottlenecks.[1] In an enterprise platform where multi-agent swarms orchestrate long-running workflows spanning weeks or months, binding project memory and task state to a transient, vendor-controlled compute layer guarantees catastrophic context loss upon process crashes, API timeouts, or deliberate model reassignments.[3]

This research report presents a comprehensive, provider-neutral architecture designed to completely decouple authoritative task state, reasoning provenance, and execution continuity from any individual language model provider. By establishing a universal agent harness built upon distributed-systems principles, the architecture treats large language models strictly as stateless compute workers. The platform assumes absolute authority over memory governance, task orchestration, Git-native workspace isolation, and tool execution ledgers. Through the implementation of a hybrid canonical state model, strict idempotency controls via the Transactional Outbox pattern, and the Agent Execution Record (AER) standard, this architecture ensures that one agent can reliably and deterministically continue another agent's work.[4] The resulting framework allows seamless failover between frontier cloud models and local specialized models, ensuring that project state, decisions, and operational memory remain durable, verifiable properties of the platform itself.

2. Definitions and Terminology

To establish a rigorous technical foundation for cross-vendor task continuity, it is necessary to formalize the terminology that governs the platform's state management and execution lifecycle. These definitions explicitly reject the conversational chatbot paradigms that currently dominate agent interactions.

Term	Architectural Definition
Provider Session	A transient, vendor-specific representation of context (e.g., an OpenAI thread or Gemini session). It serves only as a

	temporary compute layer and is explicitly rejected as the authoritative source of truth for task progress.[1]
Project Memory	Durable, long-term semantic knowledge regarding a specific project, including architectural decisions, directory structures, and operational constraints, maintained entirely independently of transient agent context.[3]
Task State	The authoritative, strongly consistent, relational record of a single unit of work. It defines the objective, acceptance criteria, dependencies, and the current execution phase within the orchestrator database.
Agent Run	A bounded execution attempt by a specific agent identity utilizing a specific model provider. A single long-running task may encompass multiple sequential or concurrent agent runs across diverse providers.[6]
Checkpoint	A validated, point-in-time snapshot of task progress comprising the Git commit, worktree state, execution ledger offset, and active advisory leases, from which an agent can deterministically resume.[2]
Run Ledger	Standardized as the Agent Execution Record (AER), this is a structured, queryable provenance primitive capturing intent, observations, inferences, and tool executions at every discrete step.[2]
Handoff Packet	A formal, provider-neutral data structure generated prior to an agent terminating or pausing. It details completed objectives, unresolved blockers, and recommended next actions to brief the subsequent worker.[1]

Event	An immutable operational record (e.g., CHECKPOINT_VALIDATED, TOOL_FAILED) stored in a write-ahead log, utilized to reconstruct state transitions and forensically audit agent behavior.[6]
Artefact	Persistent work products such as implementation plans, generated code patches, or binary files, maintained via content-hashed object storage rather than within conversational context.[1]
Context Snapshot	A point-in-time, compiled prompt payload generated by the Context Compiler, accompanied by a manifest detailing exactly which memories and files were included or excluded.[2]
Replay	The deterministic reconstruction of an agent run using recorded evidence. It enables simulated execution against mocked tool outputs or live continuation for debugging and cross-model comparison.[2]
Recovery	The forensic and systematic protocol by which a replacement agent rebuilds state comprehension from the canonical database, Git repository, and Run Ledger following a crash or interruption.

3. Why Provider Transcripts Are Insufficient

A prevalent architectural anti-pattern in contemporary multi-agent orchestration is the reliance on conversational transcripts as the primary mechanism for state transfer. When an agent handoff occurs, existing systems typically transfer either the full raw message history or a compressed natural-language summary to the receiving agent.[1] Both methodologies fundamentally fail in autonomous, high-reliability enterprise environments.

The transmission of raw conversational history induces the Context Dump Fallacy. Large, unstructured context transfers exponentially increase noise, degrade model reasoning, and cause downstream agents to lose the logical thread behind earlier decisions.[1] When critical operational signals, precise tool-call parameters, and verified architectural constraints are buried within massive, unstructured context windows, model performance degrades sharply

due to the well-documented "Lost in the Middle" effect.[1] Furthermore, full transcripts inextricably bind the task state to a specific provider's tokenization idiosyncrasies and tool-calling schemas, rendering cross-model continuation nearly impossible without significant semantic distortion.

Conversely, reliance on summarization induces severe information loss. Summaries inherently strip away the reasoning chain, the exact tool parameters attempted, and the cryptographic evidence supporting intermediate conclusions.[1] This summarization process routinely yields "plausible incorrectness," a dangerous failure mode where a downstream agent produces an output based on a hallucinated or oversimplified summary without any uncertainty signals, blinding human reviewers to the deterioration of the state.[3] True reasoning provenance cannot be faithfully reconstructed from compressed state checkpoints or conversational summaries alone.[8] Therefore, the architecture dictates that provider transcripts must be treated merely as ephemeral scratchpads. They are fundamentally insufficient recovery mechanisms and hold absolutely no authoritative weight in the system.

4. Requirements and Invariants

The foundational architectural principle of this universal agent harness is that the platform must remain fully recoverable and fully operational without relying on provider-specific APIs, proprietary session stores, or historical chat threads. To achieve this resilience, several strict invariants must hold true during normal execution, infrastructure failures, and model transitions.

The most critical invariant is that a provider session must never equate to project memory, task state, or execution history. If an agent process crashes mid-execution, particularly during a non-idempotent action such as a production database migration or an external payment dispatch, the system must definitively resolve the external state before dispatching a replacement worker.[9] When a replacement model with entirely different capabilities—such as a local, specialized coding model replacing a frontier cloud reasoning model—assumes a task, the platform must guarantee that the incoming model operates strictly on verified, schema-bound facts rather than hallucinatory conversational summaries. Furthermore, the system must maintain stable identity for all external assets. Git branches, file paths, and database records must be tracked via invariant identifiers and content hashes that survive system reboots, network partitions, and aggressive garbage collection cycles.

5. Memory-Layer Architecture

To prevent context contamination and ensure that agents receive precise, high-signal information at the exact moment of need, the architecture separates memory into six distinct, strictly managed layers. This stratification allows the Context Compiler to selectively inject required knowledge without overflowing token budgets.

5.1 Organisation Memory

Organisation memory dictates the global, long-lived rules spanning all projects across the

enterprise fleet. This repository includes strict security policies, budget controls, infrastructure credentials, data-retention requirements, and provider-routing restrictions. Because these policies fundamentally alter agent behavior across all tenants, modifications to this layer require explicit, role-based human operator approval. Agents may query this memory to understand environment constraints but possess no permissions to alter it. Crucially, historical runs retain a cryptographic foreign-key linkage to the specific policy version under which they were executed. This guarantees that compliance audits accurately reflect the rules present at execution time, preventing retroactive policy changes from invalidating historical execution audits.

5.2 Project Memory

Project memory houses long-term, semantic knowledge concerning a specific initiative or repository. It differs fundamentally from standard repository documentation by operating as a highly normalized, machine-readable semantic index.[3] While a standard repository README.md serves human developers with unstructured prose, project memory explicitly maps system boundaries, specific operational constraints, dependency graphs, and historical incidents into a queryable format. To prevent knowledge staleness, this layer does not duplicate source code; instead, it maintains rigid pointers to architecture decision records (ADRs) and directory structures, acting as an index that agents query dynamically during planning phases.

5.3 Task Memory

Task memory tracks the highly volatile lifecycle of a specific unit of work. The architecture represents task memory as an event-sourced projection, combining immutable domain events to derive a current, mutable projection of the state. This memory layer tracks the precise objective, acceptance criteria, scope, discovered blockers, and expected file modifications. By projecting immutable events into a queryable current state, the system isolates the current task status from the noise of the underlying execution process, providing a clean, strongly consistent abstraction for the central orchestration and scheduling components.

5.4 Agent-Run Memory

Agent-run memory is tightly scoped to a single execution attempt by a specific assigned model. It encapsulates the operational metadata of the run, tracking the exact prompt versions, harness versions, context snapshot manifests, tool execution results, token usage, and latency metrics.[6] This layer explicitly distinguishes between the task lifecycle and the agent-run lifecycle. A single complex task may orchestrate dozens of distinct agent runs across multiple providers, either sequentially due to crashes or concurrently due to subtask delegation. The agent-run memory ensures that the specific idiosyncrasies of an OpenAI or local model execution are contained, preventing provider-specific artifacts from leaking into the universal task memory.

5.5 Artefact Memory

Artefact memory manages persistent, bulky work products such as complex implementation

plans, generated code patches, unstructured reports, exported datasets, and test logs.[1] To optimize database performance, artefacts are content-hashed (e.g., using SHA-256) and stored entirely within object storage. They are treated as strictly immutable entities; if an agent modifies a plan, the system generates a completely new artefact version rather than mutating the existing object. This layer acts as the definitive storage for operational deliverables, ensuring that large payloads are kept entirely out of the transactional database and conversational context windows, thereby reducing latency and controlling infrastructure costs.

5.6 Event Log

The event log constitutes an append-only, immutable sequence of operational events governing the entire platform. Utilizing a write-ahead log philosophy, events such as task creation, agent assignment, tool execution success, checkpoint validation, and provider failures are recorded sequentially with strict causal ordering.[6] This log utilizes a global sequence number and correlation IDs to trace an action through the distributed system. The event log acts as the ultimate, unforgeable source of truth, enabling the reconstruction of task memory projections from scratch and facilitating exact forensic session replay following a catastrophic orchestrator crash or database corruption event.

6. Canonical State Architecture

The system utilizes a hybrid canonical-state architecture, balancing the strict transactional ACID guarantees required for orchestrator concurrency with the high-throughput, unstructured storage requirements of agent telemetry and artefact generation.

The core relational state is maintained in PostgreSQL. PostgreSQL provides the essential transactional consistency for the event log, task state projections, active concurrency leases, and the execution ledger. The system strictly and explicitly rejects the use of vector databases as the primary or sole memory system. Semantic similarity retrieval is fundamentally incapable of representing authoritative system state. A vector search cannot reliably evaluate boolean permissions, assert the completion status of a task, or determine which agent currently holds a Git branch lock. Vector stores retrieve the "most semantically similar" text, which is actively dangerous when distinguishing between an approved architectural decision and a rejected, highly similar alternative proposal.

To prevent devastating race conditions when multiple autonomous agents operate concurrently on shared infrastructure, the architecture employs PostgreSQL advisory locks mapped to specific resource identifiers. This acts as a robust, highly performant lease mechanism.[10] By using pg_advisory_lock with explicit fencing tokens, the system strictly fences resources so that a delayed agent experiencing a severe garbage collection pause or network timeout cannot accidentally overwrite the work of a newly assigned replacement agent.[10]

Object storage handles large artefacts, model context snapshots, and raw provider request and response payloads, keeping the relational database lean. Git serves as the exclusive, cryptographic source of truth for source code, diffs, and worktrees, completely decoupled from the central database. Finally, a dedicated full-text search engine indexes project memory and architectural decisions to support rapid agent queries. By combining PostgreSQL for state,

object storage for bulk data, and Git for codebase integrity, the system achieves extreme durability and operational scale without introducing provider lock-in.

7. Canonical Data Model

The logical data model enforces strict referential integrity and defines the lifecycle of all autonomous operations. The schema design ensures that every action is traceable, auditable, and inherently recoverable.

The Organisation and Project schemas define the multi-tenant boundaries of the system, establishing foreign keys for all subordinate records to enforce strict row-level security. The Task schema, identified by a cryptographic UUID, contains essential fields including the objective, project identifier, assigned agent identifier, and priority. Its status transitions are tightly controlled, moving strictly through states such as queued, executing, paused, recovering, and completed.

The Agent_Run schema tracks the ephemeral compute layer. It records the specific provider, the exact model version, execution timestamps, and termination reasons ranging from graceful completion to catastrophic process crashes. Provider sessions are managed via a transient Provider_Session table, which links the internal agent run identifier to external tracking IDs, such as the Model Context Protocol session identifiers, allowing the system to sever the connection gracefully upon task reassignment.[11]

The Event schema forms the backbone of the system's observability. It captures the aggregate identifier, event type, a JSONB payload containing the event context, and a strictly monotonically increasing causal sequence number. The Checkpoint schema acts as a recovery anchor, storing the task identifier, the run identifier, the exact Git commit hash representing the code state, the precise execution ledger offset, and a cryptographic hash of the active context snapshot. Finally, the Tool_Execution schema is crucial for side-effect tracking and idempotency management, recording the idempotency key, canonical input hash, execution status, and the specific side-effect classification of the invoked tool. Statuses across the platform are managed as both event-derived projections for historical accuracy and synchronized mutable fields for high-performance orchestrator querying.

8. Run Ledger and Event Design

Current observability systems provide execution traces but do not natively provide structured reasoning provenance as a first-class, schema-level primitive.[4] While reasoning may exist within trace data as unstructured natural language, it cannot be reliably queried or forensically analyzed. This architecture introduces the Agent Execution Record (AER), serving as the formal Run Ledger standard.[2]

The Run Ledger standard dictates that every step the agent takes is recorded neutrally and structurally. It captures intent, observation, inference, and evidence chains. This ledger is exportable as a machine-readable JSON structure, designed to be injected into a replacement agent's prompt during recovery.

A standard Run Ledger entry includes the specific run identifier, the task identifier, the provider details, and timestamps. The core of the ledger is the actions array. Each action records a

discrete step identifier, the agent's stated intent, the corresponding tool execution reference, the empirical observation derived from the tool output, and the agent's logical inference.[2] A subsequent decisions array records approved actions alongside the specific cryptographic evidence, such as a commit hash or an architecture policy, that justified the decision.

To prevent the ledger from experiencing unbounded growth and overwhelming the replacement agent's token budget, large tool execution outputs are explicitly not embedded directly within the JSON payload. Instead, the ledger stores an object-storage reference or a content hash. The system applies aggressive log compaction, retaining full, exhaustive event sequences in cold object storage while keeping only the latest validated checkpoint and active step states in the hot relational database. This balance ensures comprehensive auditability without degrading runtime performance.

9. Checkpoint and Handoff Standards

A robust multi-agent architecture requires structured, deterministic transitions of control. Before an agent voluntarily stops, reaches a runtime limit, switches providers, or pauses a task, it must produce a standard handoff packet. Handoffs are not unstructured context dumps or lossy summaries; they are structured briefings where agents transfer only the precise information required for the next step.[1]

The handoff packet serves as the canonical transition document. It must pass strict programmatic validation against a JSON or YAML schema before the orchestrator accepts it. Mandatory fields include the overarching objective, a detailed list of completed subtasks, any unresolved blockers, and a precise map of the current Git state.[7]

A standard handoff packet specifies the handoff version, identifiers, and current status. It explicitly lists the acceptance criteria and maps completed work against those criteria. Crucially, it documents the current problem blocking progress and catalogs rejected alternatives, preventing the replacement agent from exploring known dead ends. The packet includes a highly specific Git block, detailing the repository, the base commit, the active branch, the isolated worktree path, and a boolean flag indicating whether the worktree contains dirty, uncommitted changes.

Incomplete handoffs—such as when an agent process crashes catastrophically before the packet generation subroutine can execute—trigger the Recovery Manager. The system forensically infers the packet contents by querying the latest validated checkpoint, extracting the current Git worktree status, and analyzing the event log to deduce which actions completed successfully prior to the crash.

10. Recovery and Resume Protocol

When a task halts due to a provider API timeout, an orchestrator process crash, or a context window overflow, the replacement agent must execute a highly deterministic recovery sequence. The architecture expressly forbids the replacement agent from simply receiving the prior conversational transcript and attempting to continue blindly.

The recovery sequence initiates by acquiring a strict PostgreSQL advisory lock on the task identifier to prevent duplicate recovery workers from executing concurrently and corrupting

the state.[10] Following successful lock acquisition, the orchestrator loads the authoritative task object alongside the latest validated checkpoint and the preceding handoff packet. Subsequently, the system initiates a Git validation phase, inspecting the designated worktree to compare the stored commit hash against the current file system reality.[13]

If the branch diverged or the worktree is dirty, the system safely applies the stored patch from the checkpoint or forces a hard reset to the checkpointed commit to ensure absolute state synchronization. The system then enters outcome reconciliation. If the crash occurred mid-tool-execution, the system executes an idempotency check against external APIs to determine if the action succeeded or failed.[5] Once the physical and external states are reconciled, the Context Compiler generates a fresh prompt tailored to the replacement model's specific capabilities. Finally, the replacement agent runs a low-cost validation routine, such as executing a test suite or a Git status check, records any discrepancies, and safely resumes execution under a newly instantiated agent run record.

11. Context Reconstruction

To eliminate the Context Dump Fallacy, the architecture introduces a dedicated Context Compiler that dynamically assembles a fresh, high-signal context session from the canonical state infrastructure.[1] The compiler never indiscriminately dumps the entire project history into the model's context window.

Context assembly follows strict precedence weighting and token-budget allocation. Mandatory imperatives, such as agent role instructions and organizational security policies, are injected first. Task imperatives, including the objective, acceptance criteria, and the handoff packet, follow. The environmental reality, consisting of the current Git status, uncommitted diffs, and test suite results, provides the necessary grounding. Finally, project memory, including relevant architectural decisions retrieved via semantic search, is woven into the prompt.[3]

The compiler adapts its assembly strategy based on the specific capabilities of the replacement model. For a frontier cloud model possessing a 128k context window, the compiler may include dense execution ledgers and extensive codebase references. Conversely, if handing off to a smaller, local coding specialist, the compiler aggressively summarizes past business decisions and entirely omits organizational policies unrelated to code generation, ensuring the model's attention mechanism remains focused on the immediate engineering task. Every compiled context is accompanied by a manifest documenting provenance. This manifest records every item included, its token cost, its cryptographic hash, and the explicit rationale for any items excluded due to budget constraints, ensuring complete auditability of the agent's informational diet.

12. Session Replay

Session replay represents the deterministic reconstruction of an agent run using recorded evidence. It is a critical capability for failure investigation, cross-model comparison, and compliance auditing.[2] Because the architecture records all state transitions and tool execution

outputs in the canonical database, an agent run is completely reconstructable.
The platform supports multiple distinct replay modes. Narrate replay provides a read-only, step-by-step walkthrough of the agent's intent, observations, and decisions, incurring zero execution cost and serving as a powerful tool for incident reviews. Live replay re-executes the original plan against live external systems, which is useful for verifying resolution after an infrastructure outage.[2]
Crucially, the architecture introduces Mock Replay, also known as Safe Replay. This mode re-runs the reasoning loop with a new model or an updated prompt version. During safe replay, the orchestrator intercepts all tool calls and instantly returns the recorded outputs from the execution ledger. This mathematically isolates the model's reasoning capabilities from environmental non-determinism. It guarantees deterministic, duplicate-free execution and allows engineering teams to perform counterfactual evaluations, determining exactly how a new, cheaper model would have handled a complex historical incident without risking real-world side effects.[2]

13. Memory Provenance and Conflicts

In a multi-agent swarm operating over long durations, two agents will inevitably record contradictory information. The system must not silently overwrite conflicting memories, nor can it trust an agent's self-reported confidence. Every durable memory entry is bound to a strict provenance schema requiring verifiable authority and evidence.
Confidence within this architecture is not represented as a mere model-generated decimal score; it is an evidence-based status. A memory is only elevated to an authoritative state if it points to cryptographic evidence, such as a merged Git commit hash, or possesses explicit human approval. The system actively detects contradictions using background semantic evaluations and deterministic rule engines.[14]
When a conflict is detected—for example, if Memory A claims a service supports exactly-once execution, while Memory B claims it provides at-least-once execution—the system initiates a formal conflict lifecycle. Conflicting entries are logically grouped, and the system evaluates their authority levels. A verified architecture memory strictly overrides an unverified worker assumption. If authority levels are tied, an agent is dynamically dispatched to execute a verification tool, such as running a grep search or executing a test suite, to determine empirical reality.[15] The losing entry is not deleted; it is explicitly marked as superseded, preserving the historical audit trail while preventing its inclusion in future context assemblies.

14. Governance and Access Control

To maintain stability and security across the platform, role-based access control (RBAC) is strictly enforced at the memory layer. Not every agent possesses the authority to modify long-term project constraints or architectural policies.
Worker agents operate with least-privilege access. They are permitted to write run logs, create temporary task notes, propose memory updates, and attach evidence. However, they are strictly restricted from directly altering organizational policies, overriding verified security findings, or marking business decisions as authoritative. Architecture agents are granted

elevated privileges, authorized to propose and approve project memory, maintain architecture decision records, and supersede historical policies based on new evidence.
Security agents hold specialized authority to mandate approval gates for sensitive API tools and update security-specific constraints. Ultimately, human operators retain absolute, overriding authority. Operators are required to approve high-impact governance decisions, resolve complex memory conflicts, override autonomous agent decisions, and execute secure memory deprecation in accordance with legal compliance requirements. All memory proposals from autonomous workers undergo an approval workflow, ensuring that temporary assumptions are quarantined from the global project memory pool until an authorized entity verifies them.

15. Git Integration

Multi-agent operations within shared repositories routinely suffer from catastrophic failure modes, including silent file overwrites, context contamination, and locking conflicts on shared build tools.[16] To resolve these concurrency issues, the architecture tightly integrates memory and recovery mechanisms with Git worktrees, abandoning standard branching workflows. When a task is initiated, the orchestrator issues a git worktree add command, provisioning a completely isolated filesystem directory and a dedicated branch for the specific agent session.[13] The physical worktree path, the branch name, and the base commit hash are persisted securely in the canonical task state. This guarantees that multiple agents can operate on the same repository simultaneously without cross-contamination.
If a process crashes and a replacement agent is assigned, it does not rely on hallucinatory transcripts to determine which files were modified. The orchestrator routes the new agent directly to the assigned worktree. If the repository was force-pushed, or the task branch diverged during a long operational pause, the system executes an automated reconciliation. It uses the recorded base commit and uncommitted diffs stored in the latest checkpoint to safely apply the changes to the updated tree. If the physical worktree directory was accidentally deleted during a host restart, the orchestrator simply reconstructs it by checking out the branch and applying the stored diff patch, rendering the agent completely immune to local filesystem corruption.

16. Security and Privacy

The introduction of highly durable, cross-session agent memory introduces severe data leakage and privacy risks. Storing raw reasoning traces and unstructured transcripts indefinitely risks persisting prompt injection payloads, personally identifiable information (PII), and infrastructure credentials in the clear, violating compliance frameworks.
The architecture addresses these threats through multi-layered cryptographic and filtering controls. Field-level encryption is enforced for all payload columns within the PostgreSQL database. A secret-redaction middleware intercepts all provider responses and tool outputs before they reach the event log, securely replacing API keys and credentials with deterministic cryptographic hashes. Strict tenant and project isolation are enforced via PostgreSQL row-level security (RLS), mathematically guaranteeing that a compromised agent in one project cannot

retrieve memory from another.
To comply with stringent data residency and legal deletion requirements, such as the GDPR right-to-be-forgotten, the immutable event log utilizes hash-chained segments mapped to specific user identities. When a legal deletion request is issued, the cryptographic key for that specific segment is destroyed. This effectively crypto-shreds the sensitive data, rendering it unreadable while preserving the structural sequence and causal integrity of the broader append-only log. Furthermore, prompt and completion contents are captured as OpenTelemetry span events rather than core attributes, allowing administrators to dynamically suppress payload logging in production to maintain strict privacy compliance without breaking the observability trace.[18]

17. Failure-Mode Analysis

The architecture is explicitly designed to guarantee resilience against complex, compounding distributed systems failures. The following structured failure matrix defines the system's deterministic response to critical anomalies.

Failure Scenario	Detection Mechanism	Recovery Procedure	Permitted Data Loss
Model API Timeout	Provider Adapter logs timeout; heartbeat fails.	Check idempotency key against downstream service. If uncommitted, retry with fallback model.[9]	Zero. Task state remains locked.
Context-Window Overflow	Token counter exceeds limits during context assembly.	Context Compiler aggressively drops low-priority history based on relevance weights; prompts agent to summarize.	Historical reasoning metadata (temporarily excluded).
Orchestrator Crash	PostgreSQL lease expires without renewal.[10]	Replacement worker acquires lease, loads latest checkpoint, applies Git patch,	Uncommitted transient memory since last valid checkpoint.

		synthesizes handoff packet.	
Missing Handoff Packet	Database query reveals null handoff payload upon resumption.	Recovery Manager initiates forensic reconstruction using event log, Git diffs, and tool ledger.	Agent's internal, unlogged chain of thought.
Branch Divergence	Git validation phase detects base commit mismatch upon wake.	Orchestrator applies stored patch to a fresh worktree and initiates interactive rebase protocol.	Zero. Conflicts escalated to agent or human.
Conflicting Memory	Background semantic evaluation flags contradiction.[14]	System groups entries, checks authority levels, dispatches agent to verify via grep, supersedes losing entry.	Zero. Losing entry preserved historically.

18. Cross-Provider Test Suite

To ensure continuous, mathematically verifiable provider neutrality, the architecture mandates a rigorous conformance and continuity test suite. This suite executes adversarial handoffs across diverse model classes, such as swapping execution from OpenAI to Anthropic, or failing over from a massive cloud model to a constrained local model.

The suite evaluates state reconstruction by verifying that the replacement agent accurately identifies the overarching objective and completed work without hallucinating file paths. Decision continuity tests ensure the replacement agent strictly follows approved architectural decisions and explicitly ignores superseded or unverified memory notes. Tool continuity tests inject deliberate network failures immediately following a non-idempotent tool call, measuring the system's ability to safely retry via idempotency keys without duplicating the external effect.[19] Context quality is continuously monitored to ensure the Context Compiler respects strict token budgets while maintaining high signal-to-noise ratios. The primary pass criteria for production deployment dictate a 0% duplicated-work rate, a 0% test regression rate during cross-vendor handoffs, and a measurable reduction in context-token expenditure.

19. Technology Comparison

The universal agent harness was designed following a rigorous evaluation of four distinct architectural patterns.

Option A, the Transcript-centric architecture, relies entirely on provider conversation history. While attractive for its implementation simplicity, it fails completely across model providers, suffers irreparably from the Context Dump Fallacy, and provides no recovery from non-idempotent tool failures. Option B, the Mutable task-record architecture, continuously updates a central JSON object. This provides operational simplicity but suffers from devastating race conditions during concurrent execution and lacks the auditability required for debugging complex model hallucinations.

Option C, a pure Event-sourced architecture, records all changes as immutable events. While providing flawless auditability, it introduces extreme operational complexity, painful schema migrations, and unacceptable latency during state reconstruction for fast-moving agent loops.

Option D, the Hybrid Canonical-State architecture, represents the optimal balance. It leverages the strict transactional ACID guarantees of PostgreSQL for relational state and locking, append-only logs for forensic events, object storage for massive artefact payloads, and Git worktrees for isolated code execution. When comparing durable execution engines, relying on raw PostgreSQL advisory locks combined with the Transactional Outbox pattern provides equivalent concurrency safety to specialized workflow engines like Temporal, but with significantly reduced infrastructure overhead and deeper integration with relational business data.[5]

20. Recommended Reference Architecture

The recommended reference architecture enforces a strict, unidirectional data flow that entirely isolates the language models from the canonical state.

The architecture originates with the Fleet Control Plane, managed by a Human Operator. The Task Service manages the overall task lifecycle and delegates assignments. The Agent Registry tracks available workers and their capabilities. The Policy Engine enforces RBAC and global constraints. The core intelligence resides in the Context Compiler, which assembles the provider-neutral prompt, and the Recovery Manager, which handles forensic reconstruction and lock management.

These control plane components interact directly with the Canonical State Infrastructure. PostgreSQL serves as the beating heart of the system, managing task state, the event log, active advisory leases, and the transactional outbox. Object storage houses the cryptographically hashed artefacts, while Git repositories manage branches and isolated worktrees. A search index handles the semantic retrieval of project memory.

Crucially, the Provider Adapters sit at the very edge of the architecture. They function purely as a stateless compute layer. When a normal task executes, the Git Manager checks out a worktree, the Context Compiler builds the prompt, and the Provider Adapter calls the model. The model's response is intercepted by the Tool Gateway, which generates idempotency keys and executes tools using the Outbox. The Memory Service logs the AER, and the Checkpoint

Manager saves the state. If the provider fails, the Adapter simply drops the connection. The Recovery Manager seamlessly re-assigns the task, queries PostgreSQL to verify tool idempotency, compiles a new prompt for a fallback model, and resumes execution without data loss.

21. Implementation Roadmap

The implementation is strategically staged to incrementally decouple execution state from the provider layer, suitable for deployment by an advanced engineering team.

Stage 1: Minimum Durable Task State. Establish the foundational PostgreSQL schemas for tasks, agent runs, and events. Implement the basic JSON handoff packet to facilitate rudimentary state transfer between agents.**Stage 2: Checkpointing and Recovery.** Deploy the Checkpoint Manager. Implement Git worktree isolation to prevent file collisions and establish the basic patch recovery protocol for dirty workspaces.**Stage 3: Memory Layers.** Deploy the stratified six-layer memory architecture. Implement the Agent Execution Record (AER) to meticulously track intent, observation, and inference, distinguishing between proposed and verified memory.**Stage 4: Provider-Neutral Context Compiler.** Build the Provider Session Adapters and the Context Compiler. Implement dynamic token budgeting and context manifest generation to ensure auditability of the prompt assembly process.**Stage 5: Replay and Conformance Testing.** Implement the Tool Gateway with strict idempotency keys. Enable Safe Replay functionality using stored tool outputs from the ledger. Construct the cross-provider continuity test suite.**Stage 6: Governance and Security.** Deploy the RBAC permissions matrix. Implement field-level encryption, OpenTelemetry GenAI observability conventions, memory quarantine workflows, and conflict resolution mechanisms.**Stage 7: Fleet-Scale Operations.** Deploy pg_advisory_lock leases for multi-agent concurrency prevention. Optimize database triggers, implement long-term object storage archival, and deploy comprehensive operational dashboards.

22. Open Questions

While this architecture definitively resolves the immediate challenges of cross-vendor continuity and crash recovery, several deep technical domains remain open for ongoing research and refinement.

The orchestration of distributed governance adjudication requires further study. When specialized agents—such as a Security agent and an Architecture agent—conflict across a massive, swarmed workflow, defining the precise programmatic, algorithmic thresholds for human escalation versus automated resolution remains an evolving science.

Additionally, managing semantic information loss during extreme capability mismatches requires empirical baselining. While the Context Compiler effectively mitigates token budget constraints, quantifying the exact threshold of semantic degradation when migrating complex reasoning traces from a 128k-context frontier model to an 8k-context local model is a non-trivial challenge. Finally, in deeply distributed, multi-region environments where agents operate across disparate physical hosts, synchronizing Git worktrees reliably without incurring massive network overhead presents a physical infrastructure challenge that extends beyond

the orchestrator's logical bounds. These areas represent the frontier of autonomous systems engineering, requiring continuous iteration as the underlying models evolve.

Works cited

1. AI Agent Handoff: Why Context Breaks & How to Fix It | XTrace, accessed July 25, 2026, https://xtrace.ai/blog/ai-agent-context-handoff
2. Reasoning Provenance for Autonomous AI Agents: Structured Behavioral Analytics Beyond State Checkpoints and Execution Traces - arXiv, accessed July 25, 2026, https://arxiv.org/html/2603.21692v1
3. Agent Handoff Patterns: Human-Agent Interface Guide | Augment ..., accessed July 25, 2026, https://www.augmentcode.com/guides/agent-handoff-patterns-human-agent-interface
4. [2603.21692] Reasoning Provenance for Autonomous AI Agents: Structured Behavioral Analytics Beyond State Checkpoints and Execution Traces - arXiv, accessed July 25, 2026, https://arxiv.org/abs/2603.21692
5. The Transactional Outbox Pattern: A Rigorous Examination for Distributed Systems Engineers | by Mubashar | Medium, accessed July 25, 2026, https://medium.com/@nustianrwp/the-transactional-outbox-pattern-a-rigorous-examination-for-distributed-systems-engineers-9c189836f470
6. Audit Logs & Activity – Nextra - CloudAgent Documentation, accessed July 25, 2026, https://docs.cloudagent.io/guide/reports-insights/audit-logs
7. What Is the Agent Handoff Pattern? How to Design AI Outputs for Downstream Use, accessed July 25, 2026, https://www.mindstudio.ai/blog/what-is-agent-handoff-pattern
8. Reasoning Provenance for Autonomous AI Agents: Structured Behavioral Analytics Beyond State Checkpoints and Execution Traces - arXiv, accessed July 25, 2026, https://arxiv.org/pdf/2603.21692
9. Idempotent Agent Tool Calls: Preventing... | Edilec Blog, accessed July 25, 2026, https://edilec.com/blog/ai-11008/idempotent-agent-tool-calls/
10. Distributed Locks Are a Code Smell - DEV Community, accessed July 25, 2026, https://dev.to/practiceoverflow/distributed-locks-are-a-code-smell-mnd
11. MCP protocol contract - Amazon Bedrock AgentCore - AWS Documentation, accessed July 25, 2026, https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/runtime-mcp-protocol-contract.html
12. Transports - Model Context Protocol, accessed July 25, 2026, https://modelcontextprotocol.io/specification/2025-06-18/basic/transports
13. Git Worktrees for Parallel Development: 3x Throughput with AI Agents - James Phoenix, accessed July 25, 2026, https://understandingdata.com/posts/git-worktrees-parallel-dev/
14. AMA: Adaptive Memory via Multi-Agent Collaboration - ACL Anthology, accessed July 25, 2026, https://aclanthology.org/2026.findings-acl.152.pdf
15. Agent Memory Engineering — Nicolas Bustamante, accessed July 25, 2026,

https://nicolasbustamante.com/blog/agent-memory-engineering
16. How to Use Git Worktrees for Parallel AI Agent Execution | Augment Code, accessed July 25, 2026, https://www.augmentcode.com/guides/git-worktrees-parallel-ai-agent-execution
17. Git Worktrees for AI Coding: How to Run Multiple Agents Without Conflicts | MindStudio, accessed July 25, 2026, https://www.mindstudio.ai/blog/git-worktrees-parallel-ai-coding-agents
18. OpenTelemetry for AI Agents: Observability, Tracing, and the GenAI Semantic Conventions | Zylos Research, accessed July 25, 2026, https://zylos.ai/research/2026-02-28-opentelemetry-ai-agent-observability/
19. Idempotent Tool Calls: Safe Retries for AI Agents — Agentspan blog, accessed July 25, 2026, https://agentspan.ai/blogs/idempotent-tool-calls-safe-retries-for-ai-agents/
20. Building Durable AI Agents with Temporal — Crash-Proof Long ..., accessed July 25, 2026, https://niteagent.com/blog/2026-06-29-durable-ai-agents-temporal-guide/