Interchange Is Not Storage

Persistence Roles and Runtime Design for Durable AI Coding Agents

Chaitanya Mishra
Independent Researcher

March 2026

Abstract

Long-running coding agents are often discussed as if they merely need more memory, larger context windows, or better prompts. The deeper systems problem is different. Once an agent becomes effectful, interruptible, resumable, concurrent, or auditable, persistence is no longer a matter of serializing a session object. It becomes a question of which state is authoritative, which state is only a checkpoint, which events record externally visible effects, and which substrate can recover these distinctions after failure. This paper argues that many current stacks collapse four different persistence roles – interchange, checkpointing, effect history, and primary system of record – into one mutable session artifact, often JSON-shaped. That collapse is convenient for demos and bounded experiments, but it is architecturally weak for durable software work. The paper develops a persistence-role framework and an operational-depth ladder for coding-agent systems, then uses them to analyze failure modes and compare JSON files, SQLite, and PostgreSQL. The main recommendations are deliberately narrow. Plain JSON remains useful for interchange, export, debug snapshots, and bounded checkpoints. SQLite is often the strongest default for local-first and single-node durable agents because it provides transactions, indexing, constraints, and crash recovery without a separate server. PostgreSQL is usually preferable once ownership, concurrency, audit, and operational governance become shared concerns. The paper then compares BEAM-based runtimes with Ruby, Python, TypeScript/Node.js, Go, Rust, and Java. The result is not a language ranking but a control-plane analysis: BEAMchanges the design space because supervision and failure isolation are runtime primitives, while other ecosystems remain competitive in narrower or differently structured envelopes. The paper concludes with a reference architecture and a proposed evaluation protocol for testing recovery claims in enterprise-grade coding agents.

Keywords. AI coding agents; durable execution; persistence roles; SQLite; PostgreSQL; BEAM; Elixir; Erlang; auditability; workflow systems

1 Introduction

Coding agents are moving away from the conditions under which casual persistence works. Repository-level tasks increasingly require multi-step interaction with source trees, shells, test runners, CI systems, human approvals, and remote services. SWE-bench evaluates issue-resolution tasks that require environment interaction rather than one-shot code generation [43]. SWE-agent goes further by exposing an explicit agent-computer interface for repository navigation, editing, and execution [42]. LangGraph and Temporal, from different traditions, both make durable execution, replay, and recovery explicit concerns rather than afterthoughts [15–17, 44].

This change in workload invalidates a common convenience assumption: that the primary durable state of the agent can be modeled as a mutable session document, often stored as a JSON blob. That assumption is attractive because much agent state is document-shaped. Messages, plans, tool traces, memory snippets, and configuration all serialize naturally. For prototypes, exports, and bounded snapshots, that is often enough. The problem begins when the same artifact is also treated as the system of record for crash recovery, side-effect tracking, human interrupt and resume, concurrent coordination, and audit.

The central claim of this paper is precise. The most important persistence mistake in current coding-agent stacks is not merely "using JSON." It is collapsing several distinct persistence roles into one artifact. Interchange format, checkpoint format, event or effect ledger, and primary system of record solve different problems. A design that conflates them may still look elegant in a demo because the same session blob seems to do everything. Under long-running conditions, that elegance becomes operational ambiguity.

The paper therefore reframes the problem around *persistence roles* and *operational depth*. Persistence roles identify what a durable artifact is for. Operational depth identifies how exposed the system is to resumability requirements, externally visible effects, concurrent actors, and governance demands. These two axes are enough to explain why a flat session file may be harmless in one setting and deeply unsound in another.

The thesis is intentionally bounded. This is not an anti-Python paper, not a general attack on agent frameworks, and not a claim that all serious systems must adopt one runtime or one database. It is a systems analysis paper with three narrower conclusions. First, JSON remains valuable at the edges: interchange, export, and bounded snapshots are appropriate uses. Second, local and single-node durable agents often benefit more from SQLite than from file-level mutation because SQLite supplies atomic commit, write-ahead logging, constraints, and queryability with minimal operational burden [38, 40, 41]. Third, shared and compliance-heavy agent systems tend to look like ordinary transactional systems, which makes PostgreSQL a stronger primary store because of MVCC, indexing, backup and recovery tooling, and shared operational discipline [22, 23, 25, 26].

A related but distinct question concerns the control-plane runtime. Durable storage does not remove the need to isolate failures, supervise long-lived workers, handle human pauses, and recover cleanly after crashes. The BEAMdeserves serious attention because lightweight isolated processes, monitors, and supervisors are normal programming tools rather than framework add-ons [1, 3, 5, 7]. But other ecosystems remain strong in different ways: Python dominates model and tooling integration, Go offers clean service construction, Java has materially improved its concurrency story through virtual threads, and Rust remains attractive for performance-sensitive or correctness-critical components [9, 13, 14, 21, 30, 35].

Contributions.

1. A persistence-role framework that distinguishes interchange artifacts, checkpoints, effect ledgers, and primary systems of record.
2. An operational-depth ladder for classifying when coding-agent persistence requirements become transactional, queryable, concurrent, and audit-sensitive.
3. A failure-mode analysis showing why mutable session blobs are often inadequate for long-running, effectful agent systems.
4. A comparative design analysis of JSON, SQLite, and PostgreSQL as persistence substrates, plus a bounded control-plane runtime comparison across BEAM, Python, Node.js, Go, Rust, Ruby, and Java.
5. A reference architecture and evaluation protocol for durable, enterprise-grade coding agents.

Methodological stance

The paper is a systems and architecture analysis rather than a benchmark paper. Where claims are supported directly by official documentation or established systems literature, the paper states them directly. Where the argument depends on design inference, the paper marks that inference explicitly. No experiments are fabricated, and no throughput or reliability numbers are invented. Instead, the paper separates empirical claims from architectural claims and closes by proposing an evaluation protocol for testing the latter.

2 Related Work

This paper sits between several literatures that are often cited separately but rarely synthesized at the systems-design level.

The first is the literature on tool-using and repository-level agents. ReAct established the now-familiar pattern of interleaving reasoning with actions in an external environment [32]. Voyager emphasized long-horizon accumulation of reusable skills and showed that agent state is not reducible to a short conversation history [45]. SWE-bench formalized issue-resolution tasks over real repositories, and SWE-agent pushed the discussion toward agent-computer interfaces for browsing, editing, and executing code [42, 43]. These works establish the workload: coding agents increasingly operate over long, effectful trajectories.

The second is the literature and practice of durable execution and workflow systems. LangGraph names threads, checkpoints, persistence, interrupts, and durable execution as core concepts [15–17]. Temporal makes the same class of commitments from a workflow-systems perspective by treating recovery and replay as foundational rather than optional [44]. Prefect, while oriented toward orchestration rather than interactive coding agents, is also relevant because it treats result persistence, orchestration metadata, and remote execution as durable system concerns [28, 29]. The present paper borrows the durable-execution lens but asks a different question: what should count as the authoritative durable record for agent state?

The third is the database and recovery literature. Transaction-processing work provides the classical language of atomicity, durability, logging, and recovery [11]. The saga literature and later operational writing on external effects explain why durable intent recording, idempotency, and compensating actions matter whenever state transitions cross system boundaries [8, 12]. The database mechanisms discussed in this paper are not speculative. They are standard properties of SQLite and PostgreSQL, including write-ahead logging, rollback, concurrency control, indexing, and recovery [22, 25, 26, 38, 40].

The fourth is the runtime literature on fault isolation, supervision, and structured concurrency. Armstrong's work and the OTP design principles remain the clearest statement of why hierarchical supervision changes the operational character of a long-lived service [1, 5, 7]. Mainstream ecosystems have moved closer to that terrain. Python's `TaskGroup` offers structured cancellation within a process [30]. Java now has finalized virtual threads and still-preview structured concurrency [13, 14]. Go continues to rely on explicit cancellation and request scoping [9, 10]. These changes matter, but they do not erase runtime-level differences in isolation and supervision.

The fifth is the author's prior manuscript, *Beyond Session JSON: Durable State and Runtime Design for Long-Running AI Coding Agents* [18]. That paper centered the *recoverable state envelope* as the minimal durable state needed for safe resumption. The present paper uses that earlier observation only as a boundary condition. Its thesis, framing, and contribution are different: the core question here is not the internal composition of recoverable state, but the misassignment of persistence roles and the consequences of that misassignment for operationally deep coding-agent systems.

3 Problem Formulation and Core Definitions

The argument of this paper is easiest to state once several overloaded terms are separated.

Definition 1 (Agent session)**.** An *agent session* is a long-lived execution context that coordinates model calls, repository state, tool invocations, human interactions, and references to external artifacts over time. A session may span many process lifetimes, machine failures, or human pauses.

Definition 2 (Persistence role)**.** A *persistence role* is the purpose for which durable state is retained. This paper distinguishes four roles: *interchange artifact*, *checkpoint*, *effect ledger*, and *primary system of record*. One physical substrate may support multiple roles, but the roles are not conceptually interchangeable.

Definition 3 (Interchange artifact)**.** An *interchange artifact* is a serialized representation used to move data across boundaries: API payloads, exported traces, snapshot bundles, or debug dumps. Its primary obligation is portability and readability, not authoritative recovery.

Definition 4 (Checkpoint)**.** A *checkpoint* is a bounded snapshot used to accelerate restart, debugging, replay, or human inspection. A checkpoint may be discardable if a more authoritative store exists elsewhere.

Definition 5 (Effect ledger)**.** An *effect ledger* is a durable record of externally visible actions and their lifecycle states, such as `planned`, `dispatched`, `acknowledged`, `failed`, `compensated`, or `superseded`. The ledger must be queryable enough to determine whether replay is safe.

Definition 6 (Primary system of record)**.** The *primary system of record* is the authoritative durable store from which the session's recoverable state can be reconstructed after interruption. A serialization format may appear inside it, but the system of record also includes transaction, recovery, locking, indexing, and schema-evolution semantics.

Definition 7 (Control-plane runtime)**.** The *control-plane runtime* is the runtime that coordinates session lifecycles, retries, worker supervision, interrupts, leases, and recovery decisions. It is distinct from the model runtime or the tool runtime, although they may coexist in one process in simple systems.

Definition 8 (Safe resumability)**.** A session is *safely resumable* if, after crash or restart, the system can determine a continuation that is observationally valid with respect to completed and pending external effects, modulo explicit idempotency or compensation rules.

These definitions yield a design constraint that is simple enough to state formally.

Proposition 1. If a system uses a single mutable session artifact as both checkpoint and authoritative effect history, then there exists a crash schedule in which the artifact alone cannot distinguish among at least two different external-effect states. Therefore, safe resumability for non-idempotent effects requires one of the following: a jointly committed intent-and-result record, a durable effect ledger with replay discriminators, or a compensation protocol.

Interpretation. The point is not that files are inherently unsafe or that only databases can recover. The point is narrower. Once the same session artifact is expected to answer all of the following questions – what was the plan, what actually ran, which effect committed, which retry is legal, and what can be audited later – the artifact has been assigned too many roles.

Table 1: Persistence roles and their primary obligations.

Role	Primary question	Typical contents	Dominant properties
Interchange artifact	How is data moved or exported?	API payloads, snapshot bundles, debug dumps, handoff files	Portability, readability, versioned schema, loose coupling
Checkpoint	How can execution restart quickly?	Cursor state, compact context, pending work summaries, cache keys	Fast write, bounded size, discardability, restart convenience
Effect ledger	What externally visible actions were intended, dispatched, or completed?	Tool intents, idempotency keys, acknowledgments, retries, compensation markers	Append discipline, queryability, ordering, replay safety, audit
Primary system of record	What state is authoritative after failure?	Session metadata, task tree, leases, memory references, effect links, policy decisions	Transactions, crash recovery, partial update support, indexing, migrations, access control

4 Persistence Roles and Operational Depth

The persistence-role distinction explains *what* durable state is for. A second distinction explains *when* different persistence designs become necessary.

Definition 9 (Operational depth)**.** *Operational depth* is the degree to which an agent system is exposed to long-lived execution, externally visible effects, human interrupts, concurrent actors, multi-session coordination, and governance requirements. This paper uses five levels, from shallow prototypes to enterprise control planes.

Figure 1 presents the conceptual model. The important observation is that one agent session may emit multiple durable artifacts, each with different semantics.

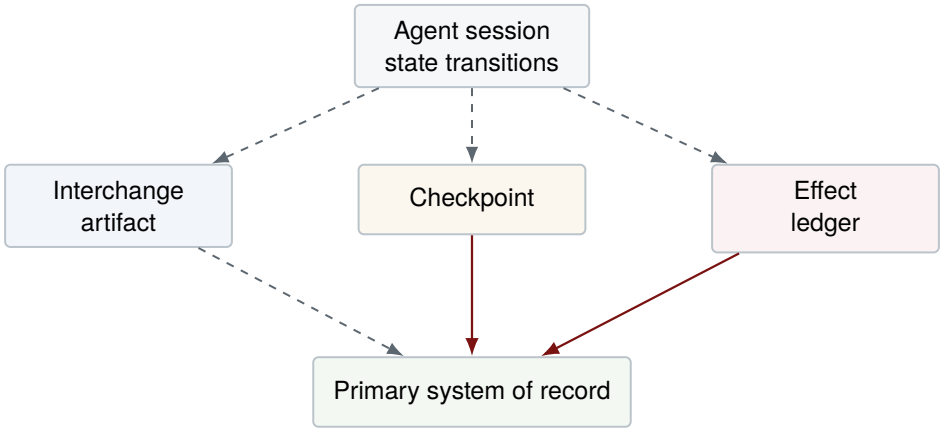

Figure 1: A durable agent system should not collapse every persistence role into one session blob. The same session may emit exportable artifacts, checkpoints, and effect records, while a distinct system of record remains authoritative.

The five operational-depth levels are intentionally simple.

Table 2: Operational depth ladder for coding-agent systems.

Level	Setting	Typical requirements	Likely persistence posture
D0	Notebook, demo, one-shot run	Minimal restart value, no hard audit, disposable state	Files and transient caches are usually acceptable
D1	Local-first single-user agent	Resume after editor restart, inspect history, avoid repeated local actions	JSON export plus SQLite or equivalent local durable store
D2	Single-node team service	Multiple sessions, queued work, human approvals, shared diagnostics	SQLite can still work if writes are mediated and workload is modest
D3	Shared multi-user platform	Concurrent actors, role separation, analytics, online maintenance	PostgreSQL or comparable shared transactional store becomes attractive
D4	Enterprise control plane	Auditability, policy gates, incident review, backup and recovery, governance	PostgreSQL-class system of record, explicit effect ledger, artifact store, operational controls

Operational depth matters because the same technical defect changes meaning across levels. If a local D1 agent loses a debug snapshot, the cost may be annoyance. If a D4 agent duplicates a pull-request comment, replays an irreversible deployment step, or cannot explain who approved a code-edit batch, the defect is not cosmetic. It is an operational failure.

Two consequences follow. First, the question "Can JSON work?" is underspecified. JSON can work for which role, at which depth, under which failure model? It works very well as an interchange format across all depth levels. It can also work as a checkpoint at D0 or D1. What becomes brittle is promoting the same file into the primary system of record as depth rises. Second, the question "Should I use SQLite or PostgreSQL?" is also underspecified until the roles are separated. SQLite is often enough for the system of record at D1 and D2, especially when there is a single machine and a controlled write path. PostgreSQL becomes preferable when the durable state must be shared among independent actors, operational teams, and services.

5 Failure Modes of Session-Blob Architectures

This section examines what goes wrong when a mutable session blob is treated as the durable heart of a coding-agent system. The term *session blob* refers to any design in which the agent's authoritative state is represented primarily as one document per session, regardless of whether that document is stored as a plain file or inside another service.

5.1 Why the design remains attractive

The session-blob design survives because it offers real short-term advantages. It minimizes schema design work. It makes debugging feel easy because one can open the file and inspect a human-readable object. It matches the intuition that agent state is "just context plus history." In small frameworks, it also minimizes the number of moving parts.

Those are not imaginary benefits. They explain why the design is rational for shallow systems. The problem is that the benefits scale poorly. Each additional operational requirement is handled by grafting more structure into the blob: nested task trees, tool traces, effect markers, interrupt metadata, human approvals, compacted history, cache references, and branch state. What begins as a convenient serialization format slowly turns into a database implemented by hand.

5.2 Full rewrites, append-only growth, and hot-cold coupling

Flat document mutation often forces one of two bad behaviors. The first is repeated full rewrites of a growing object. The second is uncontrolled append-only growth, followed by ad hoc compaction. Both couple hot state and cold history into one artifact. A small update – for example, advancing a session cursor or recording a single tool result – now implies rewriting or reparsing unrelated material.

At shallow depth, this is mostly an efficiency annoyance. At greater depth it becomes architectural friction. History retention, compaction policy, and interactive responsiveness are now intertwined. The system lacks a native way to say that one part of the state is hot and mutable while another is cold, immutable, and query-oriented.

5.3 Weak partial-update and concurrency semantics

A session blob also has weak native semantics for partial updates. When multiple logical actors need to touch different fields – a worker updating a tool result, a human approval service recording a decision, a scheduler renewing a lease, an auditor querying effect history – the system must either serialize all access through one writer or reinvent field-level concurrency in application code.

This matters even before true multi-node concurrency appears. Single-node services often have concurrent logical actors. Once the authoritative state is one document, isolation becomes all-or-nothing. Optimistic strategies devolve into compare-and-swap over large payloads. Pessimistic strategies devolve into coarse file or record locks. Neither resembles the fine-grained update semantics that transactional databases already provide.

5.4 Crash ambiguity around external effects

The most serious failure mode appears when the agent performs non-trivial external effects. Consider a run that logs a session intent, writes a file patch, posts a code review comment, or opens a pull request. If the effect commits externally but the session blob does not record the committed state durably before crash, the restart path is ambiguous. The system must choose among replaying, skipping, probing the remote system, or escalating to a human. The blob alone does not tell it which branch is valid.

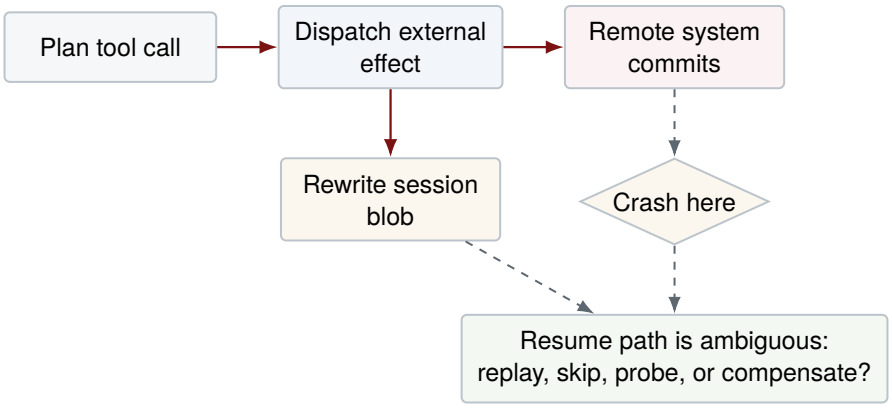

Figure 2: If a remote effect commits but the authoritative durable record is still a mutable session blob, then a crash can leave replay legality undecidable from the blob alone.

This is where the distinction between inconvenience and seriousness becomes sharp. Duplicate local cache rebuilds are often tolerable. Duplicate external effects are not. Once coding agents interact with CI, ticketing systems, pull requests, artifact registries, or deployment workflows, they need a durable effect protocol rather than only a session snapshot.

5.5 Schema drift, migration pain, and query blindness

Blob-centric systems also age badly. As the agent acquires new concepts – branch lineage, approval states, memory provenance, policy exceptions, compensation markers, task leases – the document schema drifts. Migrating active sessions becomes difficult because old and new states coexist inside partially evolved objects. Even if migrations are written carefully, the system still lacks native support for many ordinary questions: which runs failed during tool X, which sessions are waiting on human approval, which repositories have the most retries, which policy exemptions were granted last week, or which actions touched production branches.

Document search can answer some of these questions, but not with the same economy as indexed relational structures. The issue is not that every field must be normalized. The issue is that a primary system of record should support selective mutation, efficient predicates, and explainable history. Treating queryability as an afterthought often means that operators cannot see the system they are asked to trust.

5.6 Branching, merge state, and multi-agent coordination

Long-running coding agents increasingly branch. One branch explores a test failure. Another proposes a refactoring. A human may interrupt, redirect, or merge outcomes. Multi-agent systems complicate this further because several workers may share repository references, policy state, or effect history. A single session blob is poorly suited to represent branch lineage, merge intent, shared leases, or cross-session coordination. It can encode these structures, but it cannot manage them gracefully.

Table 3: Failure modes when the session blob is treated as authoritative durable state.

Failure mode	Mechanism	Mostly inconvenient at	Serious at
Full-object rewrites	Small logical changes require rewriting or reparsing large documents	D0, sometimes D1	D2+ when sessions and traces become large
Append-only growth and compaction pain	Hot state and cold history share one artifact	D0–D1	D2+ when history must be retained and queried
Replay ambiguity	External effect commits without authoritative durable intent or result record	Rarely acceptable beyond D0	D1+ whenever effects are user-visible or irreversible
Coarse concurrency control	Independent actors update the same document under one lock or compare-and-swap loop	D1	D2+ when schedulers, humans, and workers all touch state
Schema drift	New concepts accumulate inside loosely versioned nested objects	D1	D2+ when live migrations and compatibility matter
Poor queryability	Operational questions require scanning documents or ad hoc indexing	D1	D2+ when diagnosis, analytics, or audit matter
Weak audit trail	History is reconstructed from snapshots rather than durable event records	D0	D3+ where incident review and governance exist

6 Persistence Substrates: JSON, SQLite, and PostgreSQL

The previous section described the failure modes. This section asks what different substrates do well when the persistence roles are separated correctly.

6.1 JSON is valuable, but mostly at the edges

JSON remains one of the best interchange formats available. RFC 8259 succeeds precisely because JSON is simple, ubiquitous, and portable [2]. For coding agents, that makes it appropriate for API boundaries, exportable traces, checkpoint bundles, small configuration objects, and human-readable diagnostics. It is also a reasonable storage format for cold artifacts placed behind a more authoritative store.

The mistake is not to keep JSON in the architecture. The mistake is to let JSON define the architecture. An interchange format does not magically acquire transactional semantics, selective update support, or principled crash recovery merely because it is written to disk. Application code can emulate some of those properties, but the question then becomes why the system is rebuilding database semantics by hand.

6.2 When SQLite is the strong default

For many D1 and D2 systems, SQLite is the most disciplined answer. It is embedded, local, and operationally light, yet it provides exactly the properties that blob-centric designs usually miss: atomic commit, rollback, write-ahead logging, constraints, indexes, and standard query semantics [38, 40]. It also fits the local-first character of many coding agents. The agent often already runs near the working tree, caches, and tool artifacts. Adding a separate database server can be needless complexity.

SQLite is especially attractive when the write path can be mediated through one service instance or one machine. In that setting, the single-writer limitation of WAL mode is usually acceptable, while concurrent readers remain cheap [40]. The system gains durable partial updates without paying the deployment and operational costs of a client-server database. In practice, this often means using tables for hot and queryable state, with file paths or object references for large artifacts.

A further advantage is hybrid modeling. SQLite can store structured relational fields while still accommodating selective JSON content in columns when some payloads are naturally semi-structured [39]. That hybrid pattern is often more principled than storing the entire session as one nested document.

6.3 Where SQLite stops being enough

SQLite is not a universal answer. The official documentation is explicit about important boundaries. WAL mode permits one writer at a time, relies on same-machine shared memory, and is not appropriate on network filesystems [40]. Long-running readers can delay checkpoints. Write-heavy multi-process or multi-node coordination can therefore become awkward. These are not flaws in SQLite so much as signs that the workload is leaving the local-embedded regime.

The deeper signal is organizational rather than purely technical. Once multiple services or operators share responsibility for the same durable agent state, the system is no longer merely local software with persistence. It is becoming an operational platform. At that point, the primary store must support not just correctness but ownership boundaries, maintenance procedures, backup and restore, external analytics, and role-based access practices.

6.4 When PostgreSQL becomes preferable

PostgreSQL becomes attractive precisely where SQLite begins to feel socially or operationally narrow. It offers write-ahead logging, MVCC, rich indexing, shared access, mature backup and recovery tooling, and replication options that are normal expectations in managed multi-user systems [22, 24–27]. These are not abstract enterprise features. They map directly onto agent-platform problems: several coordinators touching session state, policy services recording approvals, audit queries over effect history, and offline analytics over run outcomes.

PostgreSQL also supports hybrid structures well. Its JSONB support is useful for payload fields whose internal schema changes faster than surrounding relational metadata. But the official documentation also warns indirectly against a common anti-pattern: updates to a row lock the whole row, so very large document-shaped rows can increase contention [23]. This is exactly why JSONB should complement a relational design rather than replace it indiscriminately.

A good rule is simple. Use PostgreSQL when the durable state is shared enough that one now cares about concurrent writers, differentiated access, online maintenance, audit queries, backup procedures, and team-scale operational visibility. In other words, use PostgreSQL when the agent platform has become recognizably similar to other serious transactional systems.

Table 4: Comparison of common persistence substrates by role.

Substrate	Best roles	Strengths	Limits	Typical use in agent systems
Plain JSON files	Interchange, export, bounded checkpoints	Portable, human-readable, easy to diff and ship	Weak partial updates, poor native concurrency control, weak queryability, replay ambiguity if treated as authority	API payloads, session export bundles, debug snapshots, small config
SQLite	Primary store at D1–D2; local effect ledger; local metadata	Transactions, WAL, indexes, constraints, embedded deployment, hybrid relational-plus-JSON modeling	Single-writer regime, same-machine assumptions, awkward for broad shared ownership	Local-first durable agent, desktop or single-node service, mediated write path
PostgreSQL	Primary store at D3–D4; shared effect ledger; analytics-facing metadata	MVCC, shared access, rich indexing, backup and recovery, replication, mature operational tooling	Higher operational burden, schema discipline needed, overkill for tiny local tools	Team and enterprise control planes, multi-actor coordination, policy and audit workloads

7 Runtime Comparison for the Control Plane

The runtime question should be framed narrowly. The issue is not which language can express agent logic at all. Any mainstream language can. The issue is which runtime properties matter for a long-lived control plane whose job is to coordinate sessions, workers, retries, interrupts, and recovery.

The relevant criteria are operational rather than ideological: concurrency model, fault isolation, supervision, crash handling, observability, deployment ergonomics, and how naturally the runtime integrates with durable state.

7.1 Elixir and Erlang

The strongest bounded case for BEAM-based systems is that supervision, monitoring, and lightweight process isolation are normal building blocks rather than optional framework features [1, 3–7]. For a control plane that must keep many sessions alive, isolate failures, restart components selectively, and reason about process lifecycles, that matters. A supervised process tree is not merely an implementation detail. It becomes part of the architecture.

This does not mean that the BEAMeliminates the durability problem. A supervised process still needs a correct system of record. But it does mean that the boundary between in-memory control state and durable recovery state is often easier to manage because failure is expected and structured. The BEAMis

therefore especially compelling for session coordinators, approval wait states, lease managers, and other orchestration-heavy components.

7.2 Python

Python remains the dominant worker-plane language for model integration, repository tooling, and ML-adjacent ecosystems. Its control-plane story has improved through `asyncio` and `TaskGroup`, which provide structured cancellation within a process [30]. Python is therefore far from unusable for durable agent systems.

The bounded criticism is different. Long-lived control planes in Python usually rely on framework discipline, process supervision outside the interpreter, or explicit queue and workflow systems rather than runtime-level isolation primitives. The default interpreter also remains GIL-constrained unless one adopts the newer free-threaded build, and the ecosystem transition to free-threading is still incomplete [21, 31]. Python therefore remains very strong for tool execution and model-facing workers, while its control-plane ergonomics depend more heavily on surrounding infrastructure.

7.3 TypeScript and Node.js

TypeScript and Node.js are attractive for developer tooling, web integration, and fast product iteration. The operational caution is explicit in Node's own documentation: applications should not block the event loop or the worker pool [19, 20]. That warning matters for coding agents because parsing large payloads, serializing large session documents, and performing CPU-heavy analysis in-process can all distort latency or stall progress. A Node-based control plane can be sound, but it benefits from strong boundaries around CPU-heavy or long-running work.

7.4 Go

Go remains one of the clearest choices for a service-oriented control plane. Goroutines, channels, and `context.Context` provide a practical model for request scoping, cancellation, and concurrent orchestration [9, 10]. Deployment is simple, runtime overhead is modest, and the service ecosystem is mature. The tradeoff is that supervision and failure trees are not language-level primitives in the OTP sense. They must be modeled through libraries, process structure, and operational conventions.

7.5 Rust

Rust is attractive where correctness under load, predictable resource behavior, or integration with performance-sensitive infrastructure matters. Its async ecosystem is powerful, and its type system makes many invalid states difficult to represent [35]. The caution for control-plane work is not that Rust cannot do it, but that crash semantics and recovery ergonomics are less centered on supervisor-style orchestration. Even panic handling is intentionally limited: `catch_unwind` catches only unwinding panics, and `UnwindSafe` is advisory rather than a general rollback mechanism [36, 37]. Rust is often strongest in critical subsystems, worker sandboxes, protocol layers, or embedded components around a broader orchestration architecture.

7.6 Java

Java has become much more interesting for this problem than older folklore suggests. Virtual threads are finalized and make thread-per-task designs practical at scale [13]. Structured concurrency continues to improve lifecycle management and observability, although it remains a preview API as of JDK 26 [14]. Combined with mature database tooling, operational visibility, and enterprise deployment practices, Java is a serious contender for large agent control planes. Its tradeoff is not capability but complexity budget and organizational fit.

7.7 Ruby

Ruby remains attractive where product velocity, DSL construction, and web integration dominate. Fiber schedulers and Ractors expand the concurrency story, but they do not recreate OTP-style supervision or erase the need for external operational discipline [33, 34]. Ruby is therefore competitive in bounded service envelopes and product-centric systems, though less obviously advantaged for heavily supervised, orchestration-dense control planes.

Table 5: Runtime comparison for long-lived coding-agent control planes.

Runtime	Concurrency model	Isolation and supervision	Strengths	Bounded cautions
Elixir / Erlang	Lightweight isolated processes on BEAM	Supervisors, monitors, restart strategies are first-class	Excellent fit for orchestration-heavy control planes, failure containment, long-lived coordination	Durable storage design is still required; smaller ecosystem for ML-adjacent worker code
Python	Asyncio, threads, processes; structured task groups in-process	External supervision and framework discipline usually do most of the work	Best ecosystem for model, tooling, and worker execution integration	Default GIL regime, mixed extension ecosystem, weaker runtime-level isolation story
TypeScript / Node.js	Event loop plus worker threads	Supervision is framework and process-manager oriented	Fast product iteration, good web and tooling integration	Event-loop sensitivity, CPU-heavy work and large payload processing need strong boundaries
Go	Goroutines plus contexts and channels	No OTP-style supervision, but excellent service ergonomics	Clean deployment, explicit cancellation, mature backend operations	Failure trees and restart policies are architectural, not runtime defaults
Rust	Async runtimes and threads; strong static guarantees	Isolation depends on process architecture, not supervisor primitives	Strong correctness and performance story for critical components	Higher development friction, less natural for rapidly changing orchestration logic
Java	Virtual threads plus structured concurrency trajectory	Strong operational tooling; structured concurrency still preview	Serious contender for large control planes, database-heavy platforms, observability	Complexity and platform weight may exceed needs of smaller systems
Ruby	Threads, fibers, fiber schedulers, Ractors	Limited built-in supervision model for service orchestration	Product-centric ergonomics, DSLs, web integration	Weaker default story for orchestration-heavy, highly supervised backends

8 Reference Architecture for Durable Coding Agents

A principled architecture follows directly from the previous sections. The basic design rule is to treat agent persistence as a layered system rather than a monolithic session blob.

8.1 Local-first durable agent

For D1 systems, a sound architecture is simple: a single control process, a SQLite database, a working-tree sandbox, and an artifact directory. The database stores session metadata, task state, leases, and effect records. The artifact directory holds large logs, diffs, and checkpoints. JSON remains present as export or import format and occasionally as a checkpoint format, but not as the only durable authority.

This architecture fits desktop agents, IDE-adjacent assistants, and single-user research tools. It preserves the simplicity that blob designs seek, but without sacrificing transactions or queryability.

8.2 Shared team service

At D2 and D3, the architecture usually separates the coordinator from worker execution. A scheduler assigns leases. Human approval states live in durable tables. The effect ledger becomes explicit. Checkpoints and artifacts move to object storage or a structured file store. If the service still lives on one machine and the write path is narrow, SQLite may remain adequate. Once multiple services or operators must touch the same state, PostgreSQL becomes more natural.

8.3 Enterprise control plane

At D4, the control plane should be designed as an ordinary, auditable stateful service. That means at minimum: a primary transactional store, an explicit effect ledger, durable references to artifacts, policy and approval tables, and clear separation between control-plane state and worker sandboxes.

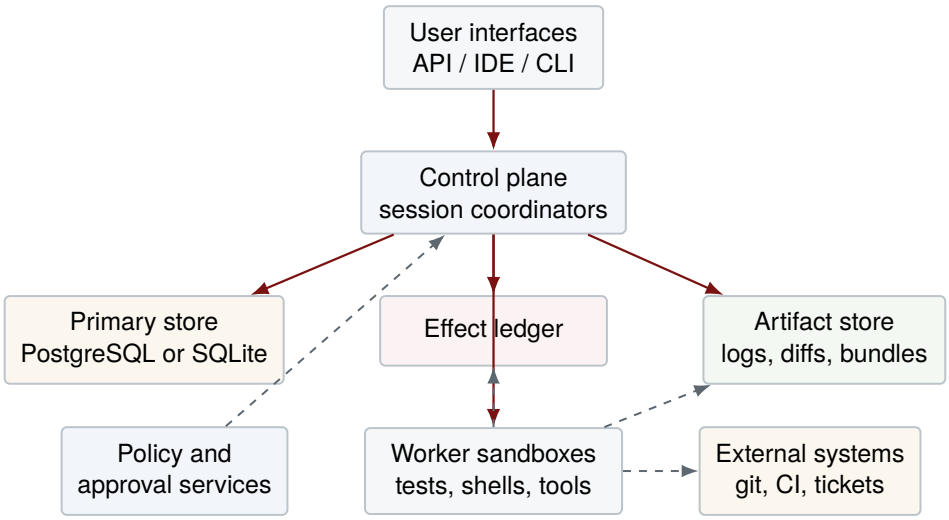

Figure 3: Reference architecture for durable coding agents. The control plane owns coordination and durable decisions; workers remain disposable. The ledger records externally visible actions, while the primary store remains authoritative for session state.

8.4 Supervision-oriented control plane

When the control plane is implemented on the BEAM, the process hierarchy itself can mirror the service architecture: one supervisor per subsystem, one coordinator per session, separate supervised pools for workers, approvals, and lease managers. The same logical architecture can be built in other runtimes, but more of the lifecycle discipline must then be supplied by frameworks, job systems, or surrounding infrastructure.

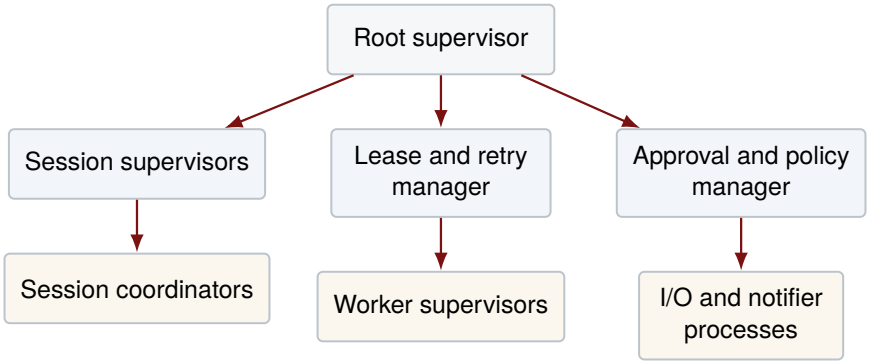

Figure 4: A supervision-oriented control plane. The main advantage is not raw performance but structured failure containment and explicit restart boundaries.

The architecture can be summarized in six rules: keep JSON as an interchange and snapshot format, not the sole durable authority; record externally visible actions in an effect ledger with replay discriminators; keep the primary store queryable and partially mutable; separate large artifacts from hot coordination state; treat workers as disposable and the control plane as durable; and choose the runtime for lifecycle management, not merely for language familiarity.

9 Evaluation Agenda and Limitations

Some claims in this paper are architectural and some are empirical. The distinction matters.

The architectural claims are these: conflating persistence roles produces replay ambiguity; partial-update and query requirements grow with operational depth; and supervision semantics materially shape control-plane design. These claims can be reasoned about from documented mechanisms and failure schedules.

The empirical claims require testing. A serious evaluation protocol for durable coding agents should therefore include at least the following workloads:

1. **Crash-injection around external effects:** force crashes before and after intent logging, dispatch, acknowledgment, and checkpoint writes.
2. **Concurrent logical actor workloads:** scheduler, worker, and approval service updating the same session family under realistic contention.
3. **Human interrupt and resume:** suspend sessions for hours or days, migrate schema, and verify safe continuation.
4. **Audit reconstruction:** answer operational and governance questions without scanning raw blobs.
5. **History growth and compaction:** compare cold-history retention, checkpoint size, and recovery latency across substrate choices.

These experiments would not prove one universal architecture, but they would expose whether a claimed durable design actually survives realistic failure schedules.

The paper also has clear limits. It does not benchmark runtimes. It does not compare distributed databases. It does not claim that every coding agent needs PostgreSQL or that BEAM-based control planes are mandatory. It does not argue against document storage in the abstract. The claim is narrower and, in the author's view, more robust: once coding-agent systems become operationally deep, the architecture must separate persistence roles and assign them to substrates with appropriate durability semantics.

10 Conclusion

The systems problem in modern coding agents is often misstated. The core issue is not merely that agent sessions grow large, or that prompts become long, or that tool traces become messy. The deeper issue is that many stacks promote a convenient session artifact into a role it cannot carry well. Interchange, checkpointing, effect history, and system of record are different jobs. When they are collapsed into one mutable session blob, the result is a design that works best in the very conditions under which enterprise coding agents are least interesting: short runs, weak effects, shallow recovery needs, and narrow ownership.

The alternative is not exotic. It is to apply familiar systems discipline. Keep interchange formats at the edges. Use a queryable transactional store as the authority. Record external effects explicitly. Separate hot coordination state from cold artifacts. Choose a control-plane runtime whose failure model matches the service you are actually building.

On that view, SQLite is often the correct default for local and single-node durable agents, while PostgreSQL becomes preferable when agent state becomes shared, concurrent, auditable, and operationally governed. BEAM-based runtimes deserve more attention than they typically receive because supervision and isolation are first-class assets for long-lived coordination, but other languages remain competitive within different architectural envelopes.

The general lesson is conservative. Durable coding agents should be designed less like chat transcripts with extra memory and more like stateful workflow systems with explicit recovery semantics. Once that shift is made, many fashionable implementation debates become easier to evaluate. The question stops being which serialization format feels convenient and becomes which roles the system must support, under which failures, for which operators. That is a better question for serious systems work.

A Illustrative schema slices by persistence role

The following sketch is not a complete schema. Its purpose is to show how the roles described in the paper can be represented separately without excessive complexity.

Table 6: Illustrative relational slices for a durable coding-agent system.

Table or store	Role	Representative fields
`sessions`	Primary store	session id, repository id, status, current cursor, owner, created at, updated at
`session_tasks`	Primary store	task id, session id, parent task id, state, priority, assigned worker, branch label
`effect_ledger`	Effect ledger	effect id, session id, task id, tool, intent hash, idempotency key, state, external reference, timestamps
`approvals`	Primary store plus audit	approval id, session id, policy gate, requester, approver, decision, rationale, timestamps
`artifacts`	Artifact index	artifact id, session id, type, URI or path, content hash, size, retention class
checkpoint blob	Checkpoint	compact execution summary, prompt cache keys, resumable cursor snapshot, version pins
export bundle	Interchange artifact	human-readable session summary, selected traces, selected diffs, machine-portable metadata

The virtue of this split is not theoretical elegance. It is operational clarity. Each store or table can now be assigned a retention policy, migration path, backup procedure, and access pattern appropriate to its role.

References

References

[1] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, 2003. https://erlang.org/download/armstrong_thesis_2003.pdf.

[2] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, Internet Engineering Task Force, 2017. https://www.rfc-editor.org/rfc/rfc8259.

[3] Elixir Documentation. Supervisor. Elixir documentation, accessed March 10, 2026. https://hexdocs.pm/elixir/main/Supervisor.html.

[4] Elixir Documentation. Task.Supervisor. Elixir documentation, accessed March 10, 2026. https://hexdocs.pm/elixir/1.18.4/Task.Supervisor.html.

[5] Erlang System Documentation. Processes. Erlang/OTP documentation, accessed March 10, 2026. https://www.erlang.org/doc/system/ref_man_processes.

[6] Erlang/OTP Documentation. `supervisor` behaviour. Erlang/OTP documentation, accessed March 10, 2026. https://www.erlang.org/doc/apps/stdlib/supervisor.html.

[7] Erlang System Documentation. Supervisor Behaviour. Erlang/OTP design principles, accessed March 10, 2026. https://www.erlang.org/doc/system/sup_princ.html.

[8] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[9] The Go Project. `context` package. Go documentation, accessed March 10, 2026. https://pkg.go.dev/context.

[10] Sameer Ajmani. Go Concurrency Patterns: Pipelines and cancellation. The Go Blog, March 13, 2014. https://go.dev/blog/pipelines.

[11] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[12] Pat Helland. Life Beyond Distributed Transactions: An Apostate's Opinion. In *CIDR*, 2007.

[13] OpenJDK. JEP 444: Virtual Threads. OpenJDK, accessed March 10, 2026. https://openjdk.org/jeps/444.

[14] OpenJDK. JEP 525: Structured Concurrency (Sixth Preview). OpenJDK, accessed March 10, 2026. https://openjdk.org/jeps/8366891.

[15] LangChain. Durable execution. LangGraph documentation, accessed March 10, 2026. https://docs.langchain.com/oss/python/langgraph/durable-execution.

[16] LangChain. Interrupts. LangGraph documentation, accessed March 10, 2026. https://docs.langchain.com/oss/javascript/langgraph/interrupts.

[17] LangChain. Persistence. LangGraph documentation, accessed March 10, 2026. https://docs.langchain.com/oss/python/langgraph/persistence.

[18] Chaitanya Mishra. *Beyond Session JSON: Durable State and Runtime Design for Long-Running AI Coding Agents*. Private manuscript, 2026.

[19] Node.js Documentation. Don't Block the Event Loop (or the Worker Pool). Node.js documentation, accessed March 10, 2026. https://nodejs.org/en/docs/guides/dont-block-the-event-loop/.

[20] Node.js Documentation. Worker threads. Node.js documentation, accessed March 10, 2026. https://nodejs.org/download/release/latest-v22.x/docs/api/worker_threads.html.

[21] Sam Gross. PEP 703 – Making the Global Interpreter Lock Optional in CPython. Python Enhancement Proposals, accessed March 10, 2026. https://peps.python.org/pep-0703/.

[22] PostgreSQL Global Development Group. PostgreSQL Documentation: Indexes. PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/indexes.html.

[23] PostgreSQL Global Development Group. PostgreSQL Documentation: JSON Types. PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/datatype-json.html.

[24] PostgreSQL Global Development Group. PostgreSQL Documentation: Logical Replication. PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/logical-replication.html.

[25] PostgreSQL Global Development Group. PostgreSQL Documentation: Concurrency Control. PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/mvcc.html.

[26] PostgreSQL Global Development Group. PostgreSQL Documentation: Write-Ahead Logging (WAL). PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/wal-intro.html.

[27] PostgreSQL Global Development Group. PostgreSQL Documentation: WAL Internals. PostgreSQL documentation, accessed March 10, 2026. https://www.postgresql.org/docs/current/wal-internals.html.

[28] Prefect. Deployments. Prefect documentation, accessed March 10, 2026. https://docs.prefect.io/v3/deploy/index.

[29] Prefect. Persist results. Prefect documentation, accessed March 10, 2026. https://docs.prefect.io/v3/develop/results.

[30] Python Software Foundation. Coroutines and Tasks. Python 3 documentation, accessed March 10, 2026. https://docs.python.org/3/library/asyncio-task.html.

[31] Python Software Foundation. Initialization, Finalization, and Threads. Python C API documentation, accessed March 10, 2026. https://docs.python.org/3/c-api/init.html.

[32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. https://arxiv.org/abs/2210.03629.

[33] Ruby Documentation. `Fiber`. Ruby documentation, accessed March 10, 2026. https://ruby-doc.org/3.1.4/Fiber.html.

[34] Ruby Documentation. `Ractor`. Ruby documentation, accessed March 10, 2026. https://docs.ruby-lang.org/en/3.4/ractor_md.html.

[35] Rust Async Working Group. *Asynchronous Programming in Rust*. Online book, accessed March 10, 2026. https://rust-lang.github.io/async-book/.

[36] The Rust Project Developers. `std::panic::catch_unwind`. Rust standard library documentation, accessed March 10, 2026. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html.

[37] The Rust Project Developers. `std::panic::UnwindSafe`. Rust standard library documentation, accessed March 10, 2026. https://doc.rust-lang.org/std/panic/trait.UnwindSafe.html.

[38] SQLite Documentation. Atomic Commit in SQLite. SQLite documentation, accessed March 10, 2026. https://sqlite.org/atomiccommit.html.

[39] SQLite Documentation. JSON Functions and Operators. SQLite documentation, accessed March 10, 2026. https://sqlite.org/json1.html.

[40] SQLite Documentation. Write-Ahead Logging. SQLite documentation, accessed March 10, 2026. https://sqlite.org/wal.html.

[41] SQLite Documentation. Appropriate Uses For SQLite. SQLite documentation, accessed March 10, 2026. https://sqlite.org/whentouse.html.

[42] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, 2024. https://arxiv.org/abs/2405.15793.

[43] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations*, 2024. https://arxiv.org/abs/2310.06770.

[44] Temporal. Temporal Platform Documentation. Temporal documentation, accessed March 10, 2026. https://docs.temporal.io/.

[45] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv preprint arXiv:2305.16291, 2023. https://arxiv.org/abs/2305.16291.