Beyond Session JSON

Durable State and Runtime Design for Long-Running AI Coding Agents

Chaitanya Mishra
Independent Researcher

March 2026

Abstract

Long-running AI coding agents are better modeled as stateful workflows than as chat sessions. They plan, invoke tools, wait for humans and external services, retry after failure, and resume across hours or days. Once an agent behaves this way, persistence stops being a serialization problem and becomes a systems problem. The central claim of this paper is that the key design object is the *recoverable state envelope*: the minimum durable state required to resume execution safely without duplicating externally visible effects. From that perspective, plain JSON remains useful for interchange, export, and bounded snapshots, but it is usually too weak to serve as the primary system of record once concurrency, crash recovery, schema evolution, queryable history, and audit become first-class concerns. The paper develops a four-plane model of agent state – conversational, execution, memory, and recovery – with audit treated as a cross-cutting concern, and uses that model to compare plain JSON files, SQLite, and PostgreSQL. The resulting recommendation is conservative. SQLite is often the best default for local or single-node agents because it provides transactions, write-ahead logging, constraints, and queryability without the operational burden of a client/server database. PostgreSQL becomes the stronger choice when state is shared, concurrent, auditable, or enterprise-managed. The paper then compares Elixir, Erlang, Ruby, Python, TypeScript/Node.js, Go, Rust, and Java as control-plane runtimes. BEAM-based systems have real operational advantages because supervision, isolation, and failure handling are runtime primitives, but those advantages do not eliminate the need for sound durable-state design.

Keywords. AI coding agents; durable execution; SQLite; PostgreSQL; BEAM; Elixir; Erlang; JSON; long-running workflows; agent infrastructure

1 Introduction

AI coding agents are increasingly asked to behave like long-running workflows rather than single-turn assistants. In realistic settings they inspect repositories, form plans, edit files, run tests, wait for CI, request human approval, and resume after interruption. Frameworks such as LangGraph make this shift explicit by treating threads, checkpoints, interrupts, and durable execution as core abstractions rather than optional conveniences [19–21]. The research literature points in the same direction. ReAct couples reasoning with tool use [33]; Voyager emphasizes reusable skills accumulated over long horizons [42]; recent work on agent memory likewise treats memory as a systems problem rather than as a prompting trick [22, 43].

Once agents persist across hours or days, the storage question changes. The issue is no longer how to save a prompt or keep the last few messages around. It is how to preserve enough state to replay safely, recover after failure, support human pauses, audit actions, and evolve the schema without corrupting active runs. That is a database problem, a runtime problem, and, ultimately, an operations problem.

This is where the familiar mutable JSON session file becomes brittle. Agent state often looks document-shaped: messages, plans, scratch work, tool outputs, and memory items. For local prototypes and debug snapshots, JSON is attractive because it is readable, ubiquitous, and easy to inspect [2]. The problem begins when the document stops being a convenience format and becomes the authoritative durable record.

The aim of this paper is not to turn JSON into a villain or to argue that one runtime solves every problem. The aim is narrower: to identify the persistence and runtime properties that matter once coding agents become long-lived, effectful systems, and to show where the usual defaults stop being adequate.

The central claim is that the relevant unit of durability is the *recoverable state envelope*. If a system cannot durably distinguish between a tool call that was merely planned, one that was dispatched, and one that already committed an external effect, then crash-safe resumption is impossible without additional protocol machinery. Viewed this way, JSON belongs at the edge – as an interchange, export, or snapshot format – while relational storage usually provides the safer core. SQLite is often the right default for local or single-node durable state. PostgreSQL is usually better once concurrency, shared ownership, audit, or enterprise operations become central.

The runtime question is related but distinct. Durable storage and workflow engines can absorb much of the recovery burden, but runtime semantics still shape the control plane. BEAM-based systems deserve attention because supervision and failure containment are runtime primitives [1, 3, 7, 9]. Mainstream languages have also improved: Python, Java, Go, Rust, Node.js, and Ruby all provide increasingly capable concurrency tools with different operational tradeoffs [12, 15, 17, 18, 23, 24, 31, 36]. The comparison that follows is therefore deliberately bounded and operational rather than tribal.

Contributions.

1. A four-plane model of agent state – conversational, execution, memory, and recovery – with audit treated as a cross-cutting concern.
2. A distinction between JSON as a flexible value format and JSON as a primary persistence architecture.
3. A deployment-oriented framework for choosing among plain JSON, SQLite, PostgreSQL, and hybrid storage designs.
4. A comparative runtime analysis focused on operational properties that matter for long-lived agent backends.

The argument is grounded in systems analysis, failure-mode reasoning, official database and runtime documentation, established distributed-systems literature, and case-based synthesis rather than in a new benchmark suite. Where the evidence is direct, the paper states it directly; where the paper infers consequences from documented mechanisms, it keeps those inferences explicit and bounded.

2 Background

2.1 Agents are becoming stateful workflows

Recent agent research points in two directions. First, agents increasingly interleave reasoning with action. ReAct is the canonical example: reasoning traces and actions are generated in a shared trajectory so that the model can revise plans while interacting with an external environment [33]. Second, agents are moving from ephemeral context windows toward accumulated experience. Voyager stores a growing library of executable skills for reuse across future tasks [42]. Recent surveys and benchmarks on agent memory make the same shift by treating memory as a distinct subsystem with its own sources, forms, operations, and evaluation criteria [22, 43].

These research directions line up with current runtime practice. LangGraph's persistence model names *threads* and *checkpoints* explicitly and uses them to support memory, human-in-the-loop control, time travel, and fault tolerance [21]. Its durable execution model requires persistence, a thread identifier, and deterministic or idempotent handling of side effects through tasks [19]. Interrupts persist graph state and allow later resumption after external input arrives [20]. Whether or not one uses LangGraph itself, the design lesson is

general: long-running agents are best understood as workflows with durable state transitions, not as plain chat transcripts.

2.2 Durability is not the same as serialization

Serialization formats answer the question "how do I encode a value?" Durable systems answer "what committed, what did not, and how can I recover?" These are different questions. Transaction processing literature has treated atomicity, durability, and recovery as first-order concerns for decades [13]. Durable systems distinguish stable from unstable state, define commit points, and provide mechanisms such as logging, journaling, and recovery procedures to re-establish a consistent state after faults.

This distinction matters for agents because agent steps are often effectful. A tool call may write a file, invoke a compiler, post a comment, open a pull request, or mutate a remote ticket. When such actions are involved, the persistence layer must answer not merely "what was the last serialized object?" but "which side effects were intended, which were dispatched, which were acknowledged, and which are safe to replay?" In distributed systems terms, the problem is close to the familiar tension between local state transitions and external effects, which often requires idempotency, outbox-style intent recording, or compensating actions rather than wishful "exactly once" thinking [10, 14].

2.3 Storage and runtime primitives that matter

SQLite and PostgreSQL are both transactional relational databases, but they occupy different operational points. SQLite is embedded, file-based, and deliberately optimized for low operational burden. It offers atomic commit, rollback journals, and a write-ahead log mode with concurrent readers but only one writer at a time [39, 40]. PostgreSQL is a client/server system built for concurrent sessions, MVCC, rich indexing, replication, and stronger operational tooling [26–29].

Runtime systems differ just as sharply. Erlang and Elixir expose lightweight isolated processes, monitors, links, and supervision trees as ordinary programming tools [1, 3, 7, 8]. Python, Node.js, Go, Rust, Ruby, and Java all support concurrency, but their default fault and lifecycle semantics differ. Python's default interpreter remains GIL-based unless using a special free-threaded build [25, 32]. Node.js centers on an event loop and a worker pool, and warns against blocking either [23]. Go uses goroutines and channels with explicit cancellation patterns [12]. Rust treats panics as unwind or abort events depending on configuration [37, 38]. Ruby now offers fibers with user-provided scheduling and actor-like Ractors, but neither maps directly to OTP supervision [34, 35]. Java added virtual threads and structured concurrency, which materially improve lifecycle management for many services [15–18].

3 Related Work

This analysis sits at the intersection of five neighboring lines of work.

The first is the literature on reasoning-and-acting agents and long-horizon memory. ReAct made interleaved reasoning and action traces a standard framing for tool-using LLM agents [33]. Voyager emphasized skill accumulation and reuse over time [42]. The survey of memory mechanisms in LLM agents organizes the field around memory sources, forms, operations, and evaluation [22]. Agent Workflow Memory extends the same line by extracting reusable workflows from prior experience [43]. These works establish that agent systems accumulate more than conversational state, but they do not mainly ask what the storage substrate for that state should be.

The second is the emerging practice literature on durable agent runtimes. LangGraph's documentation is especially useful because it names the mechanics directly: threads, checkpoints, persistence, interrupts, determinism, and idempotent side effects [19–21]. This literature is implementation-oriented rather than

comparative. It explains how to operate a runtime, not when plain JSON, SQLite, or PostgreSQL are the right primary state stores.

The third is the database and recovery literature. Gray and Reuter's transaction-processing treatment provides the general frame for atomicity and recovery [13]. SQLite's official documentation explains rollback journals, WAL, checkpointing, and the system's concurrency limits [39–41]. PostgreSQL's documentation explains WAL, MVCC, logical replication, and indexing [26–30]. The contribution here is not new database theory but a systems application of that theory to an agent workload that is often treated too casually as "just a file."

The fourth is the literature and documentation around failure containment and supervision. Joe Armstrong's thesis and the OTP documentation articulate a programming model in which isolated processes and supervisors are ordinary design tools rather than framework afterthoughts [1, 7, 9]. This literature is directly relevant to long-lived agent control planes because crash containment and recovery are not rare edge cases but expected events.

The fifth is the modern structured-concurrency literature in mainstream runtimes. Python's `TaskGroup` and Java's `StructuredTaskScope` embody a shift toward scoping groups of tasks as a unit with coordinated cancellation and error handling [15, 18, 31]. These developments narrow some of the lifecycle-management gap between OTP and mainstream environments, but they do not fully erase differences in isolation, supervision, and fault semantics.

The gap this paper addresses lies between these literatures. Existing work explains agent memory, durable execution frameworks, database mechanisms, and runtime fault models separately. What is missing is a single systems analysis that asks which kinds of agent state must be durable, why a mutable JSON blob becomes the wrong abstraction under long-lived conditions, how SQLite and PostgreSQL should actually be chosen, and where BEAM-level runtime semantics do and do not change the architecture.

4 Problem Formulation

This paper models a coding agent as a long-running stateful system that executes a sequence of steps over repositories, tools, humans, and external services.

Definition 1 (Agent session state)**.** At logical time t, let the durable state of an agent session be

$$S_t = (C_t, X_t, M_t, R_t, A_t),$$

where C_t is conversational state, X_t is execution state, M_t is long-lived memory state, R_t is recovery metadata, and A_t is audit and provenance state.

Definition 2 (Tool effect)**.** A tool invocation u_i is an attempted external action of the form

$$u_i = (tool, params, k_i),$$

where k_i is an idempotency key or other replay discriminator. A tool effect is *externally visible* if replaying u_i can change observable state outside the agent process, such as a repository, CI system, ticket, database, or messaging channel.

Definition 3 (Primary persistence model)**.** The *primary persistence model* is the mechanism the system treats as authoritative for reconstructing durable session state after interruption. A format such as JSON may appear inside the model, but the model also includes transaction, indexing, locking, migration, and recovery semantics.

Definition 4 (Recoverable state envelope)**.** The *recoverable state envelope* (RSE) at time t, written E_t, is the minimal durable subset of session state sufficient to resume the agent safely after interruption without introducing incorrect duplicate external effects. In most practical systems, E_t contains the current execution cursor, pending and completed tool intents, idempotency or compensation metadata, version pins for tools and prompts, and references to any artifacts required to continue.

Definition 5 (Safe resumability)**.** A session is *safely resumable* if, after a crash or restart, the system can reconstruct a state S'_t such that subsequent execution is observationally equivalent to some valid continuation of the pre-crash execution, modulo explicitly defined idempotency or compensation rules.

These definitions lead to a simple but important proposition.

Proposition 1. If a non-idempotent external effect can commit without a corresponding durable record inside the RSE, then there exists a crash schedule under which the system cannot determine whether replaying the effect is correct. Therefore, safe resumability for non-idempotent actions requires at least one of the following: (a) atomic joint commit of intent and effect, (b) a durable intent/result ledger with idempotency keys, or (c) a compensating action protocol.

Interpretation. This is not a theorem about one language or database. It is a design constraint. If an agent posts a pull request and crashes before its authoritative state records that post, a later restart cannot infer from a stale JSON snapshot whether it should post again. The failure is architectural, not syntactic.

The design problem can now be stated precisely. Given an agent workload with side effects, concurrent actors, and varying operational requirements, choose a persistence and runtime architecture that satisfies:

- crash consistency for E_t,
- queryable history for diagnosis and analytics,
- explicit schema evolution,
- support for human pauses and resumptions,
- auditable reconstruction of who did what and when,
- acceptable operational complexity for the deployment context.

The rest of the paper evaluates plain JSON, SQLite, PostgreSQL, and several runtime families against that problem formulation.

5 A Four-Plane Model of Agent State

The usual mistake in agent persistence is to flatten all state into one mutable object. That flattening is attractive because the agent process may hold a single in-memory object graph. It is a poor guide to durability design because different classes of state have different lifetimes, consistency requirements, and access patterns.

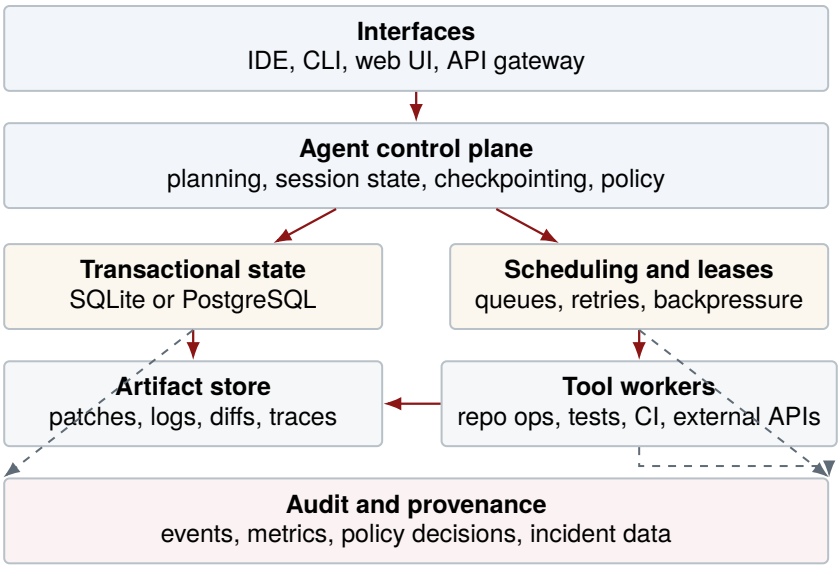

Figure 1: A long-running coding agent is better understood as a control plane around transactional state, scheduling, artifacts, and audit data rather than as a single mutable session object.

The paper uses a four-plane state model with audit as a cross-cutting concern.

5.1 Conversational state

Conversational state contains user requests, assistant responses, summaries, clarification history, and the rolling context that conditions future model calls. This plane is closest to what developers often picture as "the session." It matters for user experience, prompt construction, and some forms of explainability. It is not enough for durable execution because conversation alone does not tell the system which tools were dispatched, what effects committed, or which step should run next.

5.2 Execution state

Execution state contains the active plan, subplans, task graph, work cursor, pending branches, tool invocation records, retry counters, and references to intermediate results. This plane is hot, mutable, and deeply operational. It is the plane most harmed by a single mutable JSON file because it often needs partial updates, indexes, and transactional coupling with effect ledgers.

5.3 Memory state

Memory state contains reusable artifacts that outlive a single step or even a single run: learned workflows, repository facts, user preferences, code snippets, embeddings, summaries, and retrieved documents. The key systems point is that this plane is not synonymous with execution state. A memory item can be durable and queryable without being part of the hot execution cursor. Conflating the two leads to oversized session blobs and expensive replay paths.

5.4 Recovery state

Recovery state contains checkpoints, idempotency keys, leases, timeout markers, compensation metadata, model and tool version pins, and other information specifically needed to resume safely after interruption. This plane is not mere logging. It is what makes resumability testable rather than aspirational. In many

designs, recovery state is small but semantically dense. Losing a single idempotency key can be more harmful than losing many pages of chat transcript.

5.5 Audit as a cross-cutting concern

Audit state is best treated as a cross-cutting concern rather than a fifth peer plane because every other plane must often be observed through it. Audit records should make it possible to answer who initiated an action, under which policy and software version, with which input references, and with what result. Enterprise requirements turn this from an optional diagnostic aid into a first-class design target.

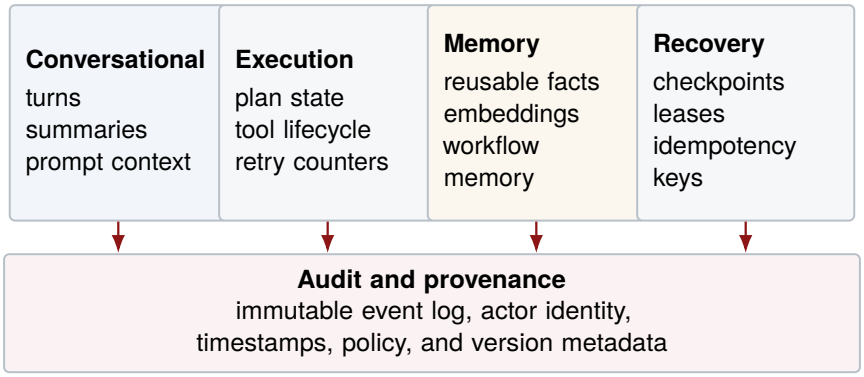

Figure 2: Agent state separates naturally into conversational, execution, memory, and recovery planes. Audit cuts across all four because each plane must remain reconstructable and attributable.

This model supports a practical design rule: keep identity, lifecycle, and invariants in relational form; keep large immutable artifacts in an artifact store; and use JSON as a value format only where its flexibility is helpful. In other words, JSON may still appear, but not as the whole architecture.

6 When JSON Stops Being Enough

The case against JSON is often overstated, but so is the case for making it the system of record. JSON is valuable; it is simply not a complete persistence architecture.

6.1 Where JSON works well

JSON is a widely implemented data interchange format [2]. It is human-readable, language-neutral, easy to diff at small scale, and suitable for transport across APIs, queues, and tools. For agents, JSON is often a good choice for:

- config files and small local preferences,
- exported session bundles,
- debug snapshots,
- immutable leaf payloads such as provider responses,
- values surfaced to users or external systems that require JSON-serializable structures [20].

If the system is single-writer, short-lived, and has no strong need for queryable history or crash-safe resumability, a JSON file may be entirely reasonable. The claim of this paper is not that JSON should disappear. The claim is that once JSON becomes the authoritative durable backbone for a serious long-running

agent, the engineering burden shifts from serialization to reimplementing database semantics in application code.

6.2 Why the session blob fails

A coding agent does not only produce messages. It produces tool intents, partial plans, retry metadata, checkpoint markers, repository snapshots, logs, and effect journals. These data have different write rates and different recovery importance. Flattening them into one mutable document creates write amplification, blurs commit boundaries, and makes hot and cold data inseparable.

The simplest implementation strategy is whole-document rewrite: deserialize the file, mutate the object, serialize the whole file back. That strategy is attractive precisely because it postpones schema thinking. Unfortunately, it pushes several systems problems into a corner where they become harder to see and therefore easier to mishandle.

6.3 Failure modes

Concurrent writes. A plain JSON file has no built-in concurrency protocol. Concurrency must be emulated through file locks, rename protocols, or a single external writer. Even with correct locking, the system usually converges on coarse-grained serialization because the update unit is the whole document. Once multiple workers or users touch the same session, lost updates and stale reads become application-level problems. By contrast, SQLite in WAL mode allows concurrent readers with a single writer, and PostgreSQL provides multi-session concurrency through MVCC and explicit locking when required [28, 40].

Crash recovery. A file rewrite gives no database-style notion of a commit record or recovery procedure. Applications can mitigate this with temporary files, renames, and `fsync()`, but those techniques must be implemented consistently. After a crash, the system often knows only that one of several possible versions exists on disk. It does not know which external effects were paired with which in-memory mutations. SQLite and PostgreSQL explicitly define commit and recovery procedures through journals or WAL [29, 30, 39, 40].

Partial updates. Agent execution state changes a little at a time. A retry counter increments. A lease expires. A single tool call moves from `planned` to `running` to `completed`. In a mutable JSON blob, these logically small updates often force a full read-modify-write cycle. That increases write amplification and makes hot spots impossible to isolate. SQL databases update rows and indexes in smaller transactional units.

Large traces. Tool-heavy coding agents accumulate terminal logs, diffs, stack traces, model outputs, and intermediate artifacts quickly. A monolithic JSON file couples hot metadata with large cold blobs. The result is slow parse times, high memory churn, and poor locality for both writes and reads. A better design stores large immutable blobs externally and keeps the hot relational metadata small.

Schema evolution. A mutable document appears schema-free only until software changes. Then version tags, migration code, and compatibility logic appear anyway. The difficult part is not adding a new field to a serializer. It is guaranteeing that partially migrated or stale session blobs can still be resumed safely. Databases do not eliminate migration problems, but they force them into explicit DDL and backfill processes rather than leaving them as ad hoc deserializer logic.

Indexing and querying. As soon as developers want to ask questions such as "which sessions are waiting for human approval?" or "show all failed tool calls in the last 24 hours" a blob file becomes awkward. The

system either scans all files or builds side indexes by hand. SQLite and PostgreSQL expose this capability directly through SQL and indexes [26]. Queryability is not a luxury feature for agent platforms; it is essential for operations.

Corruption detection. A JSON parser can detect syntactic invalidity. It cannot, by itself, tell whether the file encodes a semantically incomplete transition or whether one artifact reference is missing after a crash. Database engines pair parsing with recovery protocols and integrity tooling. SQLite's WAL and journal files are part of the persistent state and must remain paired with the database file; the documentation explicitly warns that separating them can cause loss of committed transactions or corruption [40]. PostgreSQL's WAL exists specifically for crash recovery and online backup [29].

Replay and resumability. Replay is where blob persistence fails most sharply. Suppose the agent invoked a remote code analysis service, that service completed, and the process crashed before the session blob was rewritten. On restart, the blob shows no completed call. The system cannot know whether to issue the request again, treat it as done, or ask a human. To solve this, one needs a durable tool-call ledger with idempotency keys or compensation semantics. At that point the architecture is already moving toward a database-backed design.

Auditability. A mutable snapshot is poor audit material because it overwrites history. Teams then add append-only log files next to the blob. Once they do that, the true system of record is no longer a single file but a small pile of partially synchronized files with distinct semantics. It is usually better to admit that reality early and store audit events in an append-only relational or log-structured form.

Versioning. Git or timestamped file copies can preserve document versions, but versioning a session snapshot is not the same as versioning transactional state transitions. Git preserves text; it does not tell the agent which tool intents had durable commit status or which retries remain safe.

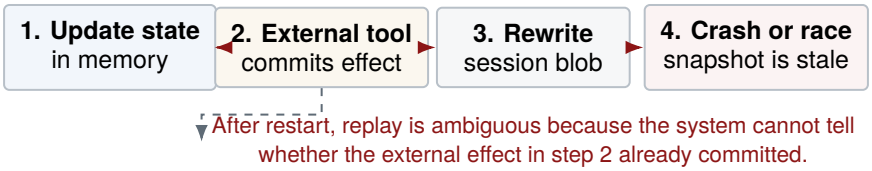

Figure 3: The visible bug in a file-backed design may be a stale or truncated snapshot, but the deeper failure is ambiguous replay once an external effect can commit before the authoritative durable state is updated.

6.4 When JSON is still acceptable

The failure analysis does not imply that JSON should vanish from agent systems. It implies that JSON should be used where its strengths match the requirement. Reasonable uses include:

- immutable exports and import bundles,
- transient snapshots for local debugging,
- payload columns inside a transactional database,
- boundary messages between services,
- very small single-user tools where the cost of a database would dominate the benefit.

The central distinction is therefore not "JSON or no JSON." It is "JSON at the edges or JSON at the core." Serious long-running agents usually want the former.

7 SQLite as a Local Durability Layer

For local and single-node agents, SQLite is often the most pragmatic durability layer. Its embedded, file-based design is frequently an advantage rather than a liability.

7.1 Why SQLite maps well to local and single-node agents

SQLite provides atomic commit and recovery through rollback journals, and it provides a WAL mode that allows readers and writers to proceed concurrently under a one-writer model [39, 40]. For a local or single-node coding agent, that property set is unusually attractive. The database lives with the application, deployment is trivial, backup is understandable, and the application still gets real transactions, indexes, and constraints rather than hand-built approximations.

This matters because many local agents are not high-concurrency systems. A desktop assistant, an IDE extension, or a single-node background agent usually has a naturally serialized write path. Even when it has several worker threads or subprocesses, those can often be funneled through one application-owned database. In such cases, "only one writer at a time" is not a practical defect. It is an acceptable design constraint in exchange for much lower operational overhead.

SQLite's own guidance is consistent with this interpretation. It recommends SQLite for many local, embedded, and application-server cases, while warning that direct concurrent access from many computers over a network filesystem is a poor fit because of latency and locking risks [41]. The same guidance matches agent deployments closely: a local-first tool or single application server is a natural SQLite workload; a shared multi-host cluster with many direct writers is not.

7.2 Why WAL, constraints, and indexes matter for agents

WAL changes the operational profile in a way that is directly relevant to agents. Readers do not block writers, readers see a consistent snapshot, and committed changes are appended to the WAL before being checkpointed back into the main database file [40]. For long-running sessions this means the system can record a sequence of small state transitions without rewriting a monolithic blob.

Constraints are just as important. A tool invocation table can require a unique idempotency key. A checkpoint can require a valid foreign key to a session. A run can require a terminal status from a finite set rather than arbitrary free text. These are small examples, but together they turn implicit invariants into enforced invariants. The practical effect is fewer silent state corruptions during refactors and partial failures.

Indexes matter because local agents still need queries. Developers want to reopen the last failed run, list tool calls for a session, or search cached summaries. SQL plus indexes makes these operations ordinary, not bespoke. SQLite can also support features such as full-text search when needed, though that is not the central argument here.

7.3 A good default architecture around SQLite

A strong pattern for a single-user or single-node agent is:

- SQLite as the primary durable store for sessions, checkpoints, tool intents, status transitions, and audit rows,
- filesystem or object-style storage for large immutable artifacts such as patches, logs, and screenshots,

- JSON used for import/export bundles and selected leaf payloads.

This hybrid already solves most of the pathologies of the mutable-blob design. The database holds the hot, queryable, invariant-rich part of the state. The artifact store holds large blobs. JSON stays where it belongs: at boundaries or in leaf values.

7.4 SQLite's limits

SQLite is not a general replacement for PostgreSQL. The WAL documentation is explicit that only one writer can append to a WAL at a time and that the WAL mechanism depends on all participants living on the same machine because the wal-index is maintained in shared memory [40]. It also warns about checkpoint starvation when long-running readers prevent checkpoints from completing [40]. Those are real constraints.

Operationally, SQLite also offers less built-in multi-node replication, fewer remote observability hooks, and weaker role separation than PostgreSQL. None of these are deal-breakers for a local tool. They become important as soon as the system turns into shared infrastructure.

The right conclusion is therefore conditional: SQLite is often the best default for local or single-node durable agent state, but it should not be stretched into a shared multi-host coordination system simply because it started as "good enough."

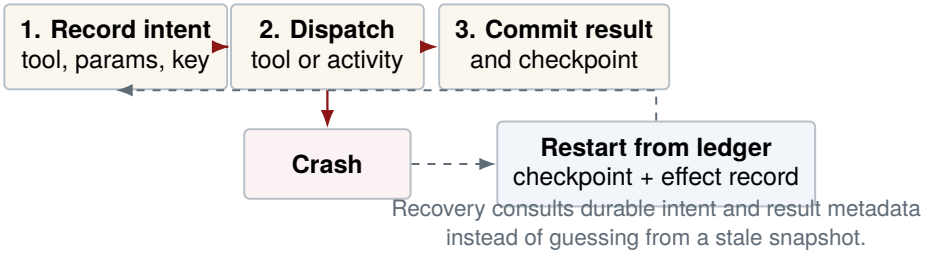

Figure 4: A durable design records intent before dispatch, then commits the result and next checkpoint. Recovery consults the checkpoint and the effect ledger rather than guessing from a mutable session blob.

8 PostgreSQL for Shared Durable State

When agent state becomes shared infrastructure, the design center changes. The hard problems are no longer only local durability and restart; they become concurrent access, multi-user visibility, security, backup, observability, forensics, and operational scaling.

8.1 What PostgreSQL adds

PostgreSQL uses WAL to ensure durability and to support crash recovery, online backup, and point-in-time recovery [29, 30]. It uses MVCC and explicit locking to allow concurrent sessions to read and write while maintaining data integrity [28]. It provides multiple index types and a mature SQL engine for operational and analytical queries [26]. It also supports logical replication for selective or cross-version replication scenarios [27].

For agent systems, this property set matters in concrete ways. MVCC and session concurrency matter when several workers, operators, or users inspect and mutate state at once. WAL and PITR matter when session history becomes operationally or legally significant. Rich indexing matters when the platform must search across tenants, runs, failure codes, tools, and time ranges. Logical replication matters when audit streams, read replicas, or selective data movement become operational requirements.

8.2 Why enterprise agent workloads are relational workloads

Enterprise agent platforms are often described in the language of LLMs, prompts, and model calls. Operationally they are closer to ordinary transactional systems. They manage identities, ownership, approvals, task leases, retries, entitlements, rate limits, artifact metadata, incident records, and retention rules. Those are relational concerns.

This does not mean that every field must be normalized into dozens of tables. It means that the core of the system benefits from relational identity and lifecycle control. PostgreSQL is especially strong here because it supports a mixed style: normalized tables for invariants and joins, optional JSON payload columns for sparse provider-specific data, and references out to large artifacts stored elsewhere. That hybrid is substantially safer than promoting raw document storage to the role of primary system of record.

8.3 Where PostgreSQL is a better fit than SQLite

PostgreSQL is usually the stronger choice when one or more of the following are true:

- several users or services need concurrent access to shared agent state,
- runs must be auditable and retained under explicit policy,
- recovery and backup need server-grade operational tooling,
- the state model is queried operationally by many dimensions,
- tenants or business domains require stronger access control and observability.

In those conditions, SQLite's simplicity becomes less important than PostgreSQL's concurrency and operations model.

8.4 Tradeoffs and costs

The cost side should be stated plainly. PostgreSQL is a separate service. It requires provisioning, monitoring, upgrades, backups, migrations, and some operational skill. That cost is real. It should not be paid for a toy local assistant. But once a team begins building hand-rolled indexing, side logs, lock daemons, multi-process mediators, or backup scripts around a file-based persistence layer, the complexity has already arrived. PostgreSQL often reduces total complexity once the system crosses from "personal tool" into "shared platform."

9 Runtime Tradeoffs

Runtime debates are often noisier than the engineering questions underneath them. For long-lived agent systems, the relevant criteria are more specific.

9.1 Operational properties that matter

The following properties matter more than language fandom:

- **fault isolation:** can one task crash without taking unrelated work down?
- **supervision and lifecycle control:** can parents observe, restart, or cancel children in a principled way?
- **concurrency model:** how easy is it to run many mostly-I/O tasks without hidden blocking hazards?
- **resumption semantics:** what happens after interruption, panic, timeout, or cancellation?

- **observability:** can operators inspect living tasks and their relationships?
- **ecosystem fit:** are model SDKs, repo tools, and enterprise integrations available where needed?

These properties are partly runtime-level and partly architectural. If durability and retries are pushed into an external engine, language differences narrow. If the language runtime itself owns the control plane, they matter more.

9.2 Elixir and Erlang: the BEAM argument

The strongest argument for Elixir in long-lived agent backends is not syntax. It is OTP on the BEAM runtime. Erlang processes are lightweight, isolated, and designed for massive concurrency; processes can monitor one another, and supervisors restart children according to explicit strategies [7–9]. Joe Armstrong's reliability thesis framed these capabilities as a way to build systems that continue operating in the presence of software errors rather than trying to eliminate failure from the model [1].

Elixir inherits those runtime properties because it runs on BEAM and uses OTP. Its `Supervisor` and `Task.Supervisor` abstractions make supervision trees ordinary application structure rather than special infrastructure code [3, 5]. Elixir's task documentation also makes ancestry and caller relationships visible in logs, which is useful for debugging long-running concurrent work [4]. For agent control planes, this means an implementation can model each session, subtask family, or tenant as supervised process structure with explicit restart policy.

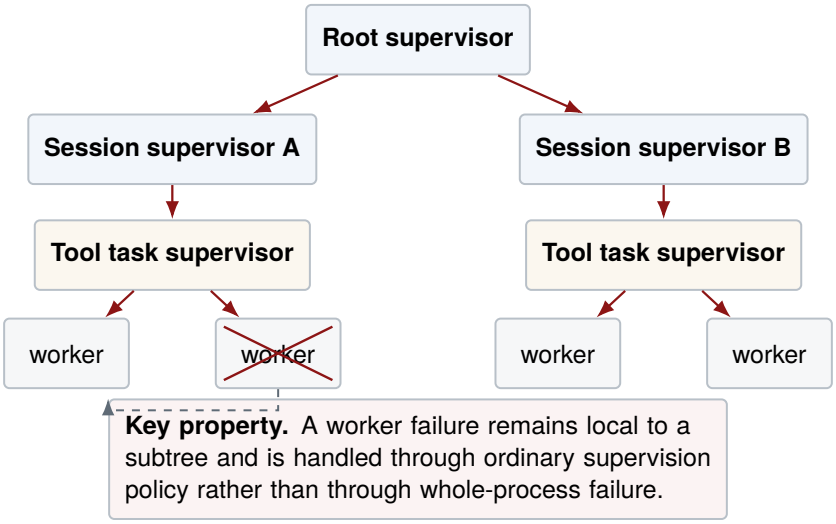

Figure 5: The BEAM model localizes failure by making supervision part of ordinary process structure. The operational advantage is not syntax but the fact that failure handling is built into the runtime model.

The argument transfers almost entirely to Erlang. If the claim concerns supervision, isolation, and runtime fault semantics, the relevant substrate is BEAM and OTP, not Elixir-specific syntax. Elixir's additional case is developer ergonomics, metaprogramming, modern tooling, and team accessibility. That is important, but it is a different kind of claim.

The BEAM argument still has limits. AI-model and developer-tooling ecosystems are usually wider in Python and often in TypeScript than in Elixir or Erlang. CPU-heavy or native code often needs careful isolation because unsafe native extensions can compromise VM guarantees. The implication is not "write everything in Elixir." A more defensible implication is "consider BEAM seriously for the orchestration and control plane of long-lived agent services."

9.3 Ruby

Ruby's case is materially different from Elixir's. Ruby offers excellent developer speed for many web and internal tools, and mature ecosystems for job queues and business application development. Modern Ruby also provides useful concurrency features. Ractors are actor-like and can run in parallel; Ruby's documentation explicitly frames them as a thread-safe parallel execution mechanism and explains that, on CRuby, the GVL is held per Ractor rather than globally across all Ractors [35]. Ruby fibers also support non-blocking scheduling, although the scheduler must be provided by the user or framework, and behavior is unchanged if no scheduler is installed [34].

These are meaningful capabilities, but they do not add up to OTP-style supervision as a dominant default programming model. In most Ruby systems, long-lived background reliability is still composed from external job systems, process supervisors, databases, and application conventions rather than from a pervasive runtime-level supervision tree. Ruby can therefore be a strong choice for internal tools, operator dashboards, and some agent services, but its strongest case is usually productivity and ecosystem fit, not first-class runtime fault containment.

9.4 Python

Python is often the path of least resistance for model integration, repository tooling, evaluation, and data work. That is a serious advantage in agent systems. The runtime case is more mixed. In CPython, unless one uses a free-threaded build, the interpreter remains protected by the GIL [25, 32]. This does not prevent I/O-bound agents or multi-process architectures, but it weakens the case for Python as a highly concurrent in-process control plane when compared with BEAM, Go, or Java virtual threads.

Python's structured-concurrency story has improved. `asyncio.TaskGroup` coordinates groups of tasks, cancels siblings when one fails, and gives stronger safety guarantees than `gather()` for nested subtasks [31]. That is important and narrows part of the lifecycle-management gap. But Python's dominant resilience model still tends to be external: process supervisors, durable queues, database state, and explicit retry ledgers. For many agent systems this is acceptable, especially when Python is used as a worker plane rather than the entire control plane.

9.5 TypeScript and Node.js

TypeScript and Node.js are attractive for agent products because they align well with web applications, API integrations, and modern developer tooling. Their runtime semantics require care. Node.js explicitly warns against blocking either the event loop or the worker pool because a small number of threads handle many clients, and blocked callbacks reduce throughput and can create denial-of-service hazards [23]. Worker threads are available and are explicitly recommended for CPU-intensive JavaScript operations rather than for I/O-heavy work, since Node's asynchronous I/O is already optimized for the latter [24].

For agent systems, the practical reading is that Node.js is strong for API-facing services and orchestration when work is mostly I/O-bound and developers are disciplined about blocking behavior. It is weaker as a "fire off arbitrary local tools and hope the runtime absorbs the chaos" platform. Its fault containment model is also more conventional than OTP-style supervision. As with Python, durable execution can be built successfully, but much of the resilience usually lives in storage, queues, and process boundaries rather than in the runtime itself.

9.6 Go

Go remains one of the strongest mainstream options for long-lived service backends. Goroutines are cheap, deployment is straightforward, and the standard library makes cancellation and deadline propagation natural

through contexts and channel patterns [12]. Go also exposes panic and recover explicitly; unrecovered panics that unwind out of the top of a goroutine can terminate the process, and a panic cannot be recovered from a different goroutine [11]. This is an important semantic difference from BEAM-style process crashes.

Operationally, Go is often an excellent fit for agent workers, schedulers, or control services that externalize durability into a database or workflow engine. What it lacks is first-class supervision as a language-wide idiom. Teams can build it, but they build it themselves or through frameworks.

9.7 Rust

Rust offers memory safety, resource control, and high performance. Those are valuable for effectful worker sandboxes, deterministic execution engines, and performance-sensitive subsystems. Its panic model is also explicit: `catch_unwind` catches unwinding panics, but not aborting panic configurations, and the language treats unwind safety as a distinct concern [37, 38]. Async Rust is powerful, but its runtime and ecosystem are more explicit and lower-level than those of languages oriented around service orchestration [36].

For long-lived agents, Rust is often strongest where control over resource usage, isolation boundaries, or deterministic execution matters more than rapid feature iteration. It is less obviously the fastest path for a feature-rich orchestration control plane unless the team already has strong Rust expertise.

9.8 Java

Java deserves more serious comparison than it often receives in agent discussions. JEP 444 standardized virtual threads, whose goal is to preserve a thread-per-request style while allowing large numbers of concurrent tasks with much lower cost than platform threads [17]. JEP 453 and `StructuredTaskScope` add a structured-concurrency model that treats related subtasks as a unit for joining, cancellation, and error handling [15, 18]. These are substantial improvements for long-lived service design.

Java is not BEAM. It does not make OTP-style supervisors a built-in universal programming idiom. But its current platform story – especially with virtual threads, structured concurrency, mature tooling, and enterprise operations – makes it a strong contender for serious agent backends. One caveat from the platform documentation is also worth noting: virtual threads are not intended for long-running CPU-intensive operations [16]. For agent control planes that are mostly I/O-bound, that is not a major defect. For computation-heavy workloads, it matters.

9.9 Reading the runtime tradeoffs

The comparative lesson is not that one runtime dominates every architecture. It is that different runtimes move resilience to different layers.

- BEAM moves a large share of failure containment and recovery into the runtime and OTP structure.
- Java moves more lifecycle control into the platform than it used to through virtual threads and structured concurrency.
- Go offers simple and effective service concurrency but expects supervision discipline to be built above the runtime.
- Python and TypeScript often win on ecosystem and agent tooling, but much of their long-lived reliability is externalized.
- Ruby's strongest argument is productivity rather than runtime-level fault semantics.
- Rust is strongest where resource control and safety dominate the design center.

A consolidated runtime comparison appears in Table 2 in the appendix tables.

10 A Practical Decision Framework

Persistence and runtime choices are contingent on deployment. Three distinctions matter most: local versus shared operation, single-node versus multi-service coordination, and light-touch versus compliance-heavy requirements.

10.1 Persistence choices

Table 3 in the appendix summarizes the main persistence options. Tables 4 and 5 collect the scenario and failure-mode comparisons.

A practical rule follows.

> **Rule of thumb.** Use plain JSON as a boundary format or bounded snapshot. Use SQLite when the authoritative state is local or single-node. Use PostgreSQL when the authoritative state is shared, concurrent, auditable, or enterprise-managed. Use a hybrid design whenever the system accumulates large immutable artifacts or irregular provider payloads.

10.2 Runtime choices

The runtime choice should follow the same logic.

- If the system is mostly a local agent or a thin service around model and repo APIs, Python or TypeScript may maximize developer speed.
- If the system is a long-lived control plane with many concurrent sessions and failure-recovery logic, Elixir, Java, and Go deserve special attention.
- If the system is polyglot, a common winning pattern is a strongly supervised or strongly structured control plane paired with Python workers for tool and model integration.

10.3 Decision tree

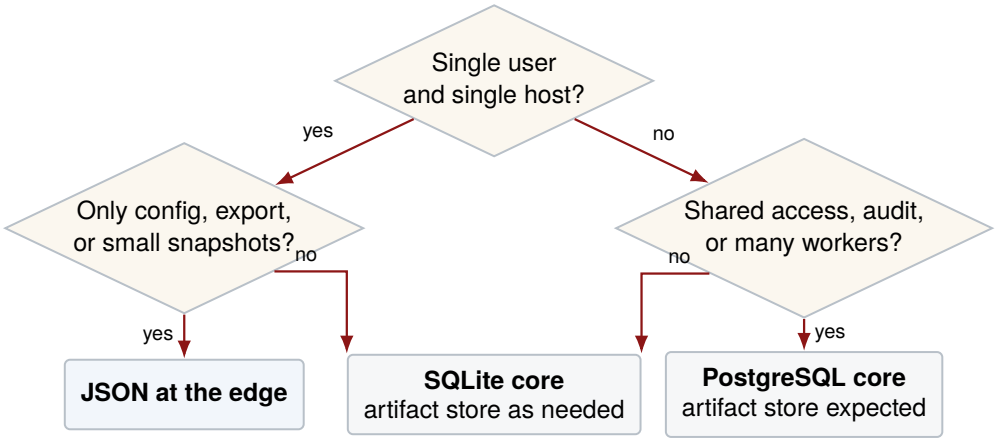

Figure 6: A practical decision tree for choosing plain JSON, SQLite, PostgreSQL, or a hybrid persistence design. In serious deployments, the durable core is usually relational even when JSON remains important at the edges.

11 Deployment Scenarios

The previous sections argued from documented system properties. This section applies that argument to three concrete deployment scenarios.

11.1 Scenario 1: A local coding agent on one developer machine

A local coding agent has one primary user, one machine, and usually one natural point of authority: the local application. The key requirements are resumability after laptop sleep or process crash, searchable history of recent runs, and bounded operational overhead. The persistence architecture that fits best is a hybrid with SQLite at the core, plus filesystem or object-style artifact storage for large logs, diffs, screenshots, and terminal output. Plain JSON remains useful for export bundles and debugging but should not be the primary state store.

Runtime choice in this scenario is mostly an ecosystem question. Python or TypeScript often wins because the local agent's usefulness depends on model providers, repository tools, and editor integration. Elixir is possible but rarely necessary. Go is attractive if the goal is a self-contained desktop or CLI binary. The main failure modes are crash during a tool call, divergence between repository state and checkpoint state, and bloated trace storage. These are well handled by a transactional tool-call ledger, checkpoint references to repository hashes or artifact IDs, and a rule that large artifacts are stored outside the hot state tables.

JSON alone is not well justified here unless the agent is extremely small and disposable. SQLite is usually enough. PostgreSQL is usually not worth the operational cost.

11.2 Scenario 2: A team internal coding agent with shared access, resumability, and audit needs

A team internal agent is no longer a pure local tool. Multiple developers may inspect the same session, approvals may be shared, and background work may continue on a server while users disconnect. Once that happens, the authoritative state is shared operational data. PostgreSQL is usually the safer default primary store because concurrent access, audit trails, and operator queries stop being edge cases. SQLite can still work if all access is mediated through one application server and the write rate remains modest, but the margin for future surprises is lower.

This scenario also changes runtime incentives. The control plane now benefits from strong lifecycle management, backpressure, and clear ownership of concurrent sessions. Go, Java, and Elixir all make sense. Python remains attractive for worker processes that run model calls or repository tools, but less obviously as the sole runtime of a shared, long-lived control plane. Ruby can be productive for the surrounding internal application but is less compelling as the center of fault-tolerant orchestration.

The key failure modes are concurrent edits to agent state, lost approval metadata, duplicate tool execution during retry, and backlog growth when workers stall. A relational intent-outbox or lease-based dispatch model addresses these better than document rewriting. A mutable JSON blob does not.

11.3 Scenario 3: An enterprise multi-tenant coding agent platform

An enterprise multi-tenant platform has different economics and different failure consequences. Sessions may run for days or weeks. Tenants may have separate retention and access rules. External effects may include ticket updates, pull requests, infrastructure changes, or notifications. Incident forensics and compliance matter. In this setting the persistence architecture is almost inevitably hybrid: PostgreSQL (or another enterprise-grade relational store) for the transactional core, an artifact store for large immutable outputs, and explicit audit tables or streams for provenance.

The runtime question also sharpens. BEAM-based runtimes deserve serious consideration here because supervision and failure containment are central control-plane problems, not nice-to-have abstractions. Java is likewise a strong choice because modern virtual-thread and structured-concurrency support pair well with enterprise operations. Go is strong when the team values operational simplicity and explicitness. Python often remains indispensable, but frequently as part of the worker plane rather than the whole platform.

The main failure modes are tenant cross-talk, ambiguous external effects, stuck workflows, schema changes while runs are active, and weak audit reconstruction after incidents. These are not problems that a document blob solves by being easy to read. They are precisely the problems for which transactional state, durable effect ledgers, version pins, and explicit recovery semantics were invented.

In this scenario, plain JSON as the primary durable session store is difficult to justify. PostgreSQL or an equivalent relational core is the conservative engineering choice. The more interesting design question is how much of the control plane should live in BEAM, Java, Go, or an external workflow system around that core.

12 Discussion

12.1 The deepest distinction is architecture, not format

The phrase "use a database instead of JSON" is directionally correct but too crude. The more useful distinction is between JSON as a format and JSON as the primary persistence architecture. JSON values inside SQLite or PostgreSQL do not recreate the file-level failure modes discussed earlier, because transaction boundaries, recovery semantics, and indexing strategy are still provided by the database. That hybrid pattern is often ideal for agent workloads whose payloads are irregular.

12.2 JSON at the edges, SQL at the core

The most practical recommendation is architectural: keep JSON at the edges and SQL at the core. Use JSON for transport, export, and flexible leaf values. Use a relational store for identity, lifecycle, coordination, and audit. Use an artifact store for large immutable outputs. This pattern follows directly from the fact that long-lived agent state has mixed access patterns and mixed recovery significance.

12.3 Why BEAM matters, and why it is not the whole answer

The BEAM case is real. A runtime that treats isolated processes and supervisors as ordinary structure changes how engineers think about failure. For long-lived agent control planes, that is valuable. But BEAM is not a substitute for good persistence architecture. A well-supervised process can still make incorrect replay decisions if the durable state model is wrong. Conversely, a Python, Go, or Java system with strong storage and workflow design can still be dependable even without OTP-style runtime semantics.

The practical synthesis is that BEAM is most compelling when the control plane itself is complex and long-lived. It is less compelling as a universal answer for every component. Polyglot architectures are likely to remain common: a strongly structured control plane, relational durable state, and worker processes in whatever language best matches the model and tool ecosystem.

12.4 External workflow engines narrow runtime differences

Durable execution frameworks can shift some responsibilities away from the language runtime. If a workflow engine already owns retries, checkpoints, timers, and deterministic replay, the difference between Python, TypeScript, Go, and Java narrows because more resilience lives outside the process. The runtime still matters for local concurrency, observability, deployment, and developer ergonomics, but the architectural weight

shifts. This is one reason the paper argues for matching language choice to the *control plane* rather than assuming one language must dominate every tier.

13 Limitations

This paper is a systems analysis and design synthesis, not an experimental benchmark paper. It does not claim measured throughput comparisons between runtimes or databases for every agent workload. It also does not claim that PostgreSQL and SQLite are the only defensible database choices; other relational or workflow-backed designs can occupy similar roles. The focus on JSON, SQLite, PostgreSQL, and the listed runtimes is motivated by their prevalence in current practice and by the need to keep the comparison concrete.

A second limitation is that runtime ecosystems evolve. Python's free-threaded work, Java's Loom effort, and continuing changes to Ruby and Rust all affect the surrounding engineering trade space. The paper therefore tries to tie claims to documented runtime semantics rather than to vague community impressions. Even so, some ecosystem judgments are necessarily qualitative.

A third limitation is scope. Security, policy enforcement, sandboxing, and compliance design are discussed only insofar as they affect persistence and runtime choice. A production platform would need a deeper treatment of secrets management, isolation boundaries, and regulatory obligations than fits here.

14 Conclusion

Long-running AI coding agents turn persistence and runtime design into central systems questions. The hardest problem is not how to serialize the last prompt. It is how to preserve a recoverable state envelope that survives interruption, supports safe replay, and makes audit and diagnosis possible.

Under that lens, plain JSON is usually the wrong primary persistence model once an agent becomes concurrent, resumable, queryable, or auditable. JSON remains useful as an interchange, export, and snapshot format, but when it becomes the authoritative system of record, developers end up rebuilding transactional semantics, indexing, and recovery logic by hand. SQLite is often an excellent default for local or single-node durable agent state because it provides real transactions and WAL without server operations. PostgreSQL is usually the stronger choice for shared or enterprise agent systems because concurrency, backup, observability, and audit become first-class.

The runtime analysis leads to an equally bounded conclusion. Elixir and Erlang have a real operational argument for long-lived agent control planes because supervision and fault isolation are runtime primitives. Ruby's case is different and leans more on productivity than on runtime-level resilience. Python and TypeScript remain powerful because of ecosystem fit, but their long-lived reliability usually depends more heavily on surrounding architecture. Go and Java are especially strong mainstream contenders for durable services. Rust is strongest where safety and control dominate.

The broad lesson is simple. Durable coding agents should be designed as stateful systems, not as chat sessions with a file attached. Once that shift is made, the persistence and runtime choices become clearer, less ideological, and easier to defend.

A Appendix A: Example Relational Schema Sketch

A practical relational schema for a durable coding agent often includes the following tables.

Table	Purpose	Typical key fields
sessions	Stable identity and ownership of agent sessions	session_id, tenant_id, creator, created_at, status
conversation_turns	Ordered user and assistant turns	turn_id, session_id, role, content_ref, created_at
plan_nodes	Task graph or plan tree	node_id, session_id, parent_id, state, summary
checkpoints	Durable snapshots of the execution cursor	checkpoint_id, session_id, parent_checkpoint_id, step_no, state_hash
tool_invocations	Intent and lifecycle of effectful actions	invocation_id, session_id, tool, idempotency_key, state, requested_at
tool_results	Durable record of returned values or effect metadata	result_id, invocation_id, artifact_ref, completed_at, exit_status
leases	Ownership and timeout control for distributed work	lease_id, invocation_id, owner, expires_at
audit_events	Immutable provenance and operator history	event_id, session_id, actor, event_type, payload_json, created_at
artifacts	References to large immutable blobs	artifact_id, content_hash, uri, media_type, size_bytes
memories	Long-lived reusable memory items	memory_id, tenant_id, session_id, type, embedding_ref, validity_scope
software_versions	Reproducibility pins for prompts, tools, and models	version_id, model, prompt_hash, tool_version, repo_commit

Table 1: Illustrative schema sketch for a durable coding agent. The point is not exact table names but the separation of identity, execution, effects, audit, and artifacts.

B Appendix B: Practical Selection Checklist

Before choosing the persistence model for an agent system, ask the following.

1. Can the system tolerate a duplicate external action after a crash?
2. Will more than one process, worker, or human mutate shared session state?
3. Do operators need to query sessions by status, actor, tool, tenant, or time range?
4. Will the state model evolve while old runs are still resumable?
5. Are auditability and incident forensics first-class requirements?
6. Are large artifacts likely to dwarf the hot execution metadata?

If the answer to most of these is no, a small SQLite-backed or even bounded JSON-backed tool may be acceptable. If several answers are yes, a relational core with explicit effect logging is usually the safer design.

Runtime	Fault isolation	Supervision model	Concurrency model	Runtime resilience	Developer speed	Library ecosystem	Fit for long-lived agents	Fit for enterprise agent platforms
Elixir	Per-process isolation on BEAM	Built-in OTP supervisors, monitors, and links	Lightweight actor-style processes	High for control planes	High	Moderate AI ecosystem; strong OTP and web tooling	Very high for orchestration and session control	High, especially as control plane in polyglot systems
Ruby	Mostly process-level shared fate; Ractors isolate selectively	Usually external (job systems, process managers)	Threads, fibers, and Ractors	Medium for carefully structured services	Very high	Strong web and business ecosystem; narrower agent tooling than Python	Medium for internal tools and moderate services	Medium to low as core control plane
Python	Threads share process fate; processes isolate	Mostly external; TaskGroup improves local structure	Threads, asyncio, and processes	Medium when durability is externalized	Very high	Very strong model and data ecosystem	High as worker plane; medium as orchestration core	Medium; strong in polyglot designs
TypeScript / Node.js	Process-level shared fate; workers isolate some work	Mostly external	Event loop, worker pool, and worker threads	Medium for I/O-heavy services	High	Strong web and integration ecosystem	Medium for API-oriented orchestration	Medium
Go	Goroutines share process fate; explicit recovery	Library or framework level, not a pervasive runtime primitive	Goroutines, channels, and contexts	High for straightfor-ward services	High	Strong infrastructure ecosystem	High when paired with durable storage or workflow engine	High
Rust	Process-level shared fate; strong memory safety	External or library-based	Threads and async runtimes	High for safety-critical components	Medium	Growing systems ecosystem; smaller agent stack	High for critical subsystems; medium for fast-moving control planes	Medium to high where performance or isolation dominates
Java	Thread isolation inside process; strong platform tooling	Framework or platform patterns; structured concurrency helps	Platform threads, virtual threads, and structured scopes	High	Medium to high	Very strong enterprise ecosystem	High for I/O-heavy, enterprise-grade services	Very high

Table 2: Elixir, Ruby, Python, TypeScript, Go, Rust, and Java compared in a common operational frame. “Library ecosystem” refers to likely fit for agent and enterprise work rather than a raw package count.

Store	Atomicity	Durability	Concurrent writes	Query ability	Schema control	Partial updates	Recovery semantics	Audit ability	Best fit scenarios
Plain JSON file	Application-defined	Application-defined	Manual locking only	Poor without side indexes	Convention only	Awkward	Ambiguous after crashes unless the application builds a protocol	Poor for mutable snapshots	Config, export, tiny local snapshots, and prototypes
SQLite	ACID transactions	Rollback journal or WAL	Readers with one writer at a time	Good SQL and indexes	Explicit DDL and constraints	Good	Defined transactional recovery	Good with append-only tables	Local-first tools, single-node agents, and mediated team services
PostgreSQL	ACID transactions	WAL, crash recovery, backup, and PITR	Strong multi-session concurrency	Strong SQL plus rich indexes	Explicit DDL, constraints, and roles	Strong	Defined transactional recovery plus replication	Very good with relational audit design	Shared platforms, multi-tenant services, and compliance-heavy systems

Table 3: JSON versus SQLite versus PostgreSQL as primary persistence models. The decisive difference is not syntax but transaction, concurrency, and recovery semantics.

Scenario	Recommended runtime	Recommended primary state store	Why	Main tradeoff
Local single developer agent	Python or TypeScript; optionally Go for packaged tooling	SQLite, plus filesystem/object store for artifacts	Gives real transactions and resumability with minimal setup while staying close to the local tool ecosystem	One-writer limit and weaker multi-device sharing
Single-server team agent	Go, Java, or Elixir for control plane; Python workers common	PostgreSQL if shared state is important; SQLite only if access is tightly mediated and low-contention	Shared sessions, approvals, and audits quickly become relational operational data	More operational overhead than a local-first design
Enterprise multi-tenant agent platform	Elixir, Java, or Go control plane; Python workers for model/tool integrations	PostgreSQL plus artifact store; queue or workflow engine around it	Multi-user concurrency, audit, backup, and security dominate the design	Higher complexity in operations, migrations, and governance
Compliance-heavy internal platform	Elixir or Java control plane	PostgreSQL with append-only audit tables and artifact retention	Governance, provenance, and forensics are first-class; relational control is valuable	Slower iteration and higher storage/retention costs
High-throughput orchestration backend	Elixir, Go, or Java	PostgreSQL or hybrid workflow-store design	Concurrency and lifecycle control matter more than a document-shaped session abstraction	Requires careful queueing, partitioning, and observability design

Table 4: Deployment scenario recommendation matrix. The runtime recommendation concerns the control plane; worker planes may still be polyglot.

Failure mode	JSON	SQLite	PostgreSQL	Mitigation notes
Crash during write	Whole-file rewrite can leave stale or invalid snapshot	Journal or WAL enables transactional recovery	WAL and crash recovery procedures are built in	Prefer transactional intent/result ledger; avoid whole-file authoritative snapshots
Partial state update	Usually forces read-modify-write of full document	Row-level or statement-level updates inside transaction	Same, with stronger multi-session concurrency	Keep hot execution metadata separate from large artifacts
Schema change	Manual version tags and deserializer branches	Explicit migrations; some DDL limits but manageable	Explicit migrations with mature tooling	Version the state model and test resume across versions
Concurrent update	Manual locks; high risk of lost update	Serialized writers; good if mediated	MVCC and locks support concurrent sessions	Access patterns, not ideology, determine the right store
Large trace query	Full scans and large parse cost	Good for moderate workloads with indexes	Strong for large shared workloads with richer indexing	Store large blobs outside the hot transactional core
Replay after restart	Ambiguous unless separate effect journal exists	Good if tool intents and checkpoints are transactional	Strong if ledger, leases, and coordination are designed relationally	Idempotency keys or compensation remain necessary
Audit reconstruction	Mutable snapshot overwrites history	Good with append-only tables and artifact references	Very good with append-only tables, replicas, and analytics	Treat audit state as first-class, not as leftover logs
State corruption	Syntax errors detectable; semantic gaps are hard	Better recovery and integrity tooling than plain files	Strongest operational recovery and verification tooling here	Hash artifacts; retain intent and result records separately
Tool call duplication	High risk if crash occurs between effect and file rewrite	Moderate if ledger and keys are in place	Moderate to low if ledger and keys are in place	No store eliminates duplication risk without protocol design

Table 5: Failure mode matrix for common persistence faults in long-running agent systems. The mitigation column matters as much as the store column because protocol design remains necessary for external effects.

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 7, 2026. https://hexdocs.pm/elixir/main/Supervisor.html.

[4] Elixir Documentation. Task. Elixir documentation, accessed March 7, 2026. https://hexdocs.pm/elixir/main/Task.html.

[5] Elixir Documentation. Task.Supervisor. Elixir documentation, accessed March 7, 2026. https://hexdocs.pm/elixir/1.18.4/Task.Supervisor.html.

[6] Erlang System Documentation. Errors and Error Handling. Erlang/OTP documentation, accessed March 7, 2026. https://www.erlang.org/doc/system/errors.html.

[7] Erlang System Documentation. Processes. Erlang/OTP documentation, accessed March 7, 2026. https://www.erlang.org/doc/system/ref_man_processes.

[8] Erlang/OTP Documentation. `supervisor` behaviour. Erlang/OTP documentation, accessed March 7, 2026. https://www.erlang.org/doc/apps/stdlib/supervisor.html.

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

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

[11] Andrew Gerrand. Defer, Panic, and Recover. The Go Blog, August 4, 2010. https://go.dev/blog/defer-panic-and-recover.

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

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

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

[15] Oracle. `StructuredTaskScope` (Java SE 21 and JDK 21 API). Oracle Java documentation, accessed March 7, 2026. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/StructuredTaskScope.html.

[16] Oracle. `Thread` (Java SE 20 and JDK 20 API), virtual-thread notes. Oracle Java documentation, accessed March 7, 2026. https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Thread.html.

[17] OpenJDK. JEP 444: Virtual Threads. OpenJDK, accessed March 7, 2026. https://openjdk.org/jeps/444.

[18] OpenJDK. JEP 453: Structured Concurrency (Preview). OpenJDK, accessed March 7, 2026. https://openjdk.org/jeps/453.

[19] LangChain. Durable execution. LangGraph documentation, accessed March 7, 2026. https://docs.langchain.com/oss/python/langgraph/durable-execution.

[20] LangChain. Interrupts. LangGraph documentation, accessed March 7, 2026. https://docs.langchain.com/oss/javascript/langgraph/interrupts.

[21] LangChain. Persistence. LangGraph documentation, accessed March 7, 2026. https://docs.langchain.com/oss/python/langgraph/persistence.

[22] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A Survey on the Memory Mechanism of Large Language Model based Agents. arXiv preprint arXiv:2404.13501, 2024. https://arxiv.org/abs/2404.13501.

[23] Node.js Documentation. Don't Block the Event Loop (or the Worker Pool). Node.js documentation, accessed March 7, 2026. https://nodejs.org/en/docs/guides/dont-block-the-event-loop/.

[24] Node.js Documentation. Worker threads. Node.js v22 documentation, accessed March 7, 2026. https://nodejs.org/download/release/latest-v22.x/docs/api/worker_threads.html.

[25] Sam Gross. PEP 703 – Making the Global Interpreter Lock Optional in CPython. Python Enhancement Proposals, accessed March 7, 2026. https://peps.python.org/pep-0703/.

[26] PostgreSQL Global Development Group. PostgreSQL 18 Documentation: Indexes. PostgreSQL documentation, accessed March 7, 2026. https://www.postgresql.org/docs/current/indexes.html.

[27] PostgreSQL Global Development Group. PostgreSQL 18 Documentation: Logical Replication. PostgreSQL documentation, accessed March 7, 2026. https://www.postgresql.org/docs/current/logical-replication.html.

[28] PostgreSQL Global Development Group. PostgreSQL 18 Documentation: Concurrency Control. PostgreSQL documentation, accessed March 7, 2026. https://www.postgresql.org/docs/current/mvcc.html.

[29] PostgreSQL Global Development Group. PostgreSQL 18 Documentation: Write-Ahead Logging (WAL). PostgreSQL documentation, accessed March 7, 2026. https://www.postgresql.org/docs/current/wal-intro.html.

[30] PostgreSQL Global Development Group. PostgreSQL 18 Documentation: WAL Internals. PostgreSQL documentation, accessed March 7, 2026. https://www.postgresql.org/docs/current/wal-internals.html.

[31] Python Software Foundation. Coroutines and Tasks. Python 3.14 documentation, accessed March 7, 2026. https://docs.python.org/3.14/library/asyncio-task.html.

[32] Python Software Foundation. Initialization, Finalization, and Threads. Python 3.14 C API documentation, accessed March 7, 2026. https://docs.python.org/3/c-api/init.html.

[33] 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.

[34] Ruby Documentation. `Fiber`. Ruby documentation, accessed March 7, 2026. https://ruby-doc.org/3.1.4/Fiber.html.

[35] Ruby Documentation. `Ractor`. Ruby documentation, accessed March 7, 2026. https://docs.ruby-lang.org/en/3.4/ractor_md.html.

[36] Rust Async Working Group. *Asynchronous Programming in Rust*. Online book, accessed March 7, 2026. https://rust-lang.github.io/async-book/.

[37] The Rust Project Developers. `std::panic::catch_unwind`. Rust standard library documentation, accessed March 7, 2026. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html.

[38] The Rust Project Developers. `std::panic::UnwindSafe`. Rust standard library documentation, accessed March 7, 2026. https://doc.rust-lang.org/std/panic/trait.UnwindSafe.html.

[39] SQLite Documentation. Atomic Commit in SQLite. SQLite documentation, accessed March 7, 2026. https://sqlite.org/atomiccommit.html.

[40] SQLite Documentation. Write-Ahead Logging. SQLite documentation, accessed March 7, 2026. https://sqlite.org/wal.html.

[41] SQLite Documentation. Appropriate Uses For SQLite. SQLite documentation, accessed March 7, 2026. https://sqlite.org/whentouse.html.

[42] 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.

[43] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. arXiv preprint arXiv:2409.07429, 2024. https://arxiv.org/abs/2409.07429.