Worldline: Causally Versioned Operational Twins for Agentic Developer Harnesses

Chaitanya Mishra
Independent Researcher

March 2026

Abstract

Agentic developer systems are currently built around a tool-centric assumption: the environment is treated as a collection of callable interfaces, and intelligence is expected to emerge from prompts plus tool selection. This assumption is useful for peripheral integration, but it is too weak to support deep autonomy, durable memory, reliable replay, precise permissions, or multi-agent coordination across the full surface of software work. Real software tasks span code, terminals, build systems, runtime state, logs, traces, infrastructure, documentation, version history, and human intent. They are not isolated calls. They are constrained transformations of a live world.

This paper introduces *Worldline*, a new primitive for agentic developer harnesses. A Worldline is a causally versioned, executable operational twin of a developer environment. It continuously projects raw environmental signals into a typed world model composed of entities, evidence, memories, branches, capabilities, and receipts. Agents no longer act by calling tools directly. They act by proposing verified transition contracts over world state. The runtime computes intent-scoped context closures, plans transitions, executes them through actuators, verifies postconditions through observers, and records receipts that support audit, replay, and recovery.

The result is a harness architecture in which context extraction, execution, memory, trust, observability, and coordination become parts of a single stateful substrate rather than separate ad hoc layers. The paper presents the conceptual model, semi-formal framework, architecture, security model, replay semantics, and coordination model for Worldline. It also outlines a concrete adoption path in which existing tool protocols become leaf adapters beneath the Worldline runtime, not the primary programming primitive. The central claim is that autonomous software work should be grounded in an executable world model, not a menu of remote procedures.

Keywords: agentic systems, developer tools, software engineering, operational twins, replay, autonomy, observability, orchestration, memory, trust

1 Introduction

The current wave of coding agents has produced a useful but incomplete abstraction. Agents are frequently given a prompt window, a short-lived message history, and a set of tool endpoints. When the task is local, linear, and short-lived, this arrangement can work. When the task crosses

repository boundaries, shell state, runtime telemetry, deployment context, incident history, or team knowledge, the abstraction begins to collapse. The agent is forced to reconstruct the world every few minutes. State is fragmented across logs, terminals, version control, CI, dashboards, documents, and human explanations. Memory drifts. Permissions are coarse. Debugging becomes archaeology.

This is not merely an implementation problem. It is a primitive problem. A developer environment is not fundamentally a set of tools. It is a living operational world whose state evolves over time and whose meaningful units are entities, constraints, causal dependencies, and legal state transitions. Treating that world as a set of tool invocations produces brittleness for the same reason that treating a database as a set of shell commands would produce brittleness. The representation is too weak relative to the structure of the work.

This paper proposes *Worldline*, a state-first harness architecture for agentic software development. In Worldline, the environment is continuously projected into a causally versioned operational twin. Agents receive not an arbitrary pile of recent outputs, but an intent-conditioned closure over that twin. They do not directly invoke external systems as their primary programming model. They propose transitions that are checked against preconditions, permissions, budgets, and postconditions, then executed through a transaction fabric and recorded as receipts. Execution becomes replayable. Memory becomes anchored. Multi-agent work becomes branchable. Audit becomes native.

The paper makes four claims.

First, the primary abstraction for agentic developer systems should be an executable world model rather than a tool registry.

Second, reliable autonomy requires unifying context extraction, execution, policy, and memory inside a single causally versioned substrate.

Third, multi-agent software work is best modeled as branch-based coordination over shared operational state, not as loosely coupled prompt exchanges.

Fourth, existing tool-call protocols remain useful, but only as device-driver layers beneath a stronger harness architecture.

The rest of the paper develops this claim from problem statement to architecture to evaluation.

2 Problem Statement

Software work is a special class of operational reasoning. It is not pure planning, because the environment is mutable, partially observed, and adversarially complex. It is not pure execution, because meaningful action depends on latent structure and developer intent. It is not pure memory retrieval, because what matters is often a live dependency across layers of the system. Agentic developer harnesses therefore face five simultaneous demands:

1. **Cross-layer visibility.** The agent must understand code, terminal activity, version history, build rules, dependency graphs, runtime signals, documents, and infrastructure as parts of one environment.

2. **Durable semantics.** Facts discovered during work must survive the session in a form that

remains attached to the entities they concern.

3. **Governed action.** Execution must be shaped by explicit permissions, trust zones, budgets, and secret boundaries.
4. **Replay and diagnosis.** Developers must be able to answer what the agent knew, why it acted, what changed, and how to replay or recover the episode.
5. **Long-horizon coordination.** Multiple agents and humans must be able to work concurrently on related subproblems without collapsing into unstructured chat.

Existing systems satisfy pieces of this list. None makes the list native. The gap becomes especially visible in long-running tasks such as framework migrations, dependency upgrades across service boundaries, flaky test diagnosis, incident response, secrets rotation, or infrastructure hardening. In those settings the main failure mode is not lack of raw capability. It is loss of coherent world state.

The correct design question is therefore not, "How do we expose more tools to the agent?" It is, "What representation and runtime would let an agent work against the software world itself?"

3 Why Tool-Centric Paradigms Fail

Tool-centric harnesses, including those that package capabilities behind standardized server interfaces, solve a narrow but important problem: they provide a disciplined way to reach external systems. That is valuable. It is not enough. Their deepest limitations arise from the fact that they make tool invocation rather than world state the central primitive.

3.1 The first-class object is wrong

A tool registry says what may be called. It does not say what the world currently is. The agent must infer environment structure from scattered outputs at runtime. This reverses the correct information hierarchy. In real development work, state precedes operation.

3.2 Context is assembled too late

Tool-centric systems are predominantly pull-based. Relevant context is fetched on demand, often in response to prompt-time guesses. That means the agent repeatedly rediscovers the same facts. It also means important context is absent unless explicitly requested. Cross-layer dependencies, such as "this flaky test is correlated with a deployment boundary and a specific infra change," are easy to miss because no stable operational model exists to connect them.

3.3 Memory is weakly anchored

Most current memory approaches are session transcripts, vector stores, or manually curated notes. These are not wrong. They are simply ungrounded. They lack typed attachment to the entities and

causal episodes they describe. As a result, memory retrieval is often semantically shallow and prone to staleness.

3.4 Execution lacks transactional meaning

A tool call is an action attempt. It is not a semantically complete transition. It usually does not encode preconditions, postconditions, invariants, compensation, or blast radius. As a result, developers cannot reliably reason about what the agent *intended* to change relative to the actual world.

3.5 Permissions are endpoint-centric

Allowing an agent to invoke a shell, write to a repository, or call a deployment API is too coarse. Real permission questions look like this: may an agent modify only files reachable from a declared service boundary, under a latency budget, without crossing into production write scope, while keeping secrets opaque and requiring human review for schema changes? Endpoint-centric policy has no native vocabulary for this.

3.6 Observability is after-the-fact

System logs can show that the agent called a tool. They do not naturally expose the chain from intent to context to action to verification to recovery. Developers are left with disconnected artifacts rather than a coherent causal record.

3.7 Multi-agent coordination is externalized

When multiple agents collaborate in a tool-centric setting, coordination is usually layered on top via prompts, shared notes, or thin task queues. There is no native merge model over shared operational state. The result is brittle decomposition and opaque handoff.

These are structural limitations. They do not disappear with larger models or more tools. A stronger primitive is required.

4 Design Principles

Worldline is guided by eight design principles.

Principle 1 (State first)**.** *The harness must maintain a live operational model of the environment rather than reconstructing the world from tools at decision time.*

Principle 2 (Evidence first)**.** *Every fact in the model must carry provenance, freshness, and confidence.*

Principle 3 (Intent first)**.** *Developer goals must be represented as typed objects with success predicates, invariants, budgets, and trust requirements.*

Principle 4 (Transitions, not calls)**.** *The primary action unit must be a state transition contract with preconditions, postconditions, observers, and compensation paths.*

Principle 5 (Branch by default)**.** *Speculative agent work must occur on isolated branches of world state that can be diffed, replayed, merged, or discarded.*

Principle 6 (Capabilities over endpoints)**.** *Authorization must attach to scopes, transition classes, budgets, and trust zones rather than only to raw interfaces.*

Principle 7 (Anchored memory)**.** *Memory must be attached to entities and episodes with validity conditions so that it can decay or invalidate as the world changes.*

Principle 8 (Local-first adoption)**.** *The architecture must provide value in a single-repo local mode, then expand to CI, runtime, and organization-scale knowledge without demanding a full-platform migration on day one.*

5 Conceptual Model

Definition 1 (Worldline)**.** *A* Worldline *is a causally versioned, executable operational twin of a developer environment. It is composed of typed entities, evidence-backed facts, anchored memories, branch histories, capability state, and transition receipts, all evolving over time.*

The central insight is simple: the agent should reason over an explicit model of the software world and express desired changes as legal transitions on that model. Tools still exist, but they become leaf actuators. They are no longer the primary ontology.

5.1 Core primitives

Worldline is built from ten primitives.

Table 1: Core Worldline primitives

Primitive	Meaning
Entity	A typed object in the developer world, such as a repository, file, symbol, test, process, service, endpoint, deployment, document, incident, branch, secret scope, developer, or agent.
Evidence	A provenance-bearing observation from sensors, logs, commands, traces, documents, or human declarations.
Facet	A derived view over entities, such as dependency graphs, ownership maps, blast-radius maps, or failure clusters.
Intent	A typed developer goal with scope, success predicates, invariants, budgets, and trust envelope.
Closure	The minimal useful slice of the Worldline needed for a particular intent or sub-intent.
Transition Contract	A legal, verifiable state change with preconditions, action semantics, observers, postconditions, and compensation.
Branch	A speculative timeline rooted in a base state of the Worldline.
Receipt	An immutable record of attempted or completed transition execution, including evidence, deltas, policy checks, and outcome.
Anchored Memory	Durable memory bound to entities or episodes with freshness and validity rules.
Capability Envelope	The permissions, trust zone constraints, budgets, and review requirements under which transitions may execute.

5.2 From tool menus to operational closure

A prompt and a tool list provide possibilities. A Worldline provides *closure*. Given an intent, the runtime extracts the subgraph of code, runtime state, history, tests, docs, policies, and prior episodes that makes the task actionable. The closure is not a bag of search results. It is a bounded operational slice with typed relations.

For example, the intent "repair the failing checkout integration test without changing public API semantics" expands naturally into a closure containing the test, touched modules, recent relevant commits, build targets, associated services, impacted endpoints, runtime errors, prior similar failures, owning team, and applicable policies. The agent begins from structure, not from guesswork.

5.3 Operational twins versus passive digital twins

The term "operational twin" is important. A passive representation of the environment is insufficient. Worldline is executable. It knows which transitions are legal, who may request them, what budgets apply, what observers verify them, and how to compensate when they fail. In other words, it combines model, control plane, and audit plane.

6 Semi-Formal Framework

This section provides a compact framework for reasoning about Worldline. The goal is not full formal verification. The goal is to make the architecture precise enough to implement and evaluate.

6.1 State

Let the Worldline state at time t be

$$\mathcal{W}_t = \langle G_t, E_t, M_t, B_t, \Gamma_t, R_t \rangle$$

where:

- G_t is a typed multigraph of entities and relations.
- E_t is the evidence ledger.
- M_t is the set of anchored memories.
- B_t is the set of branches and branch metadata.
- Γ_t is policy, capability, and trust state.
- R_t is the receipt ledger.

A fact in the graph is not a naked assertion. It is an evidence-backed claim

$$f = \langle s, p, o, \pi, c, \varphi, \sigma \rangle$$

where s is subject, p predicate, o object, π provenance, c confidence, φ freshness, and $\sigma \in \{\text{observed}, \text{inferred}, \text{stale}, \text{contradicted}\}$ is epistemic status.

6.2 Projection

Sensors produce raw observations $\omega_t \in \Omega$, where Ω includes file-system events, version control deltas, shell transcripts, process metadata, CI signals, logs, traces, documents, policy changes, and human declarations.

The projection kernel updates the Worldline via

$$\mathcal{W}_{t+1} = \Phi(\mathcal{W}_t, \omega_t)$$

where Φ performs normalization, identity resolution, entity updates, evidence attachment, and memory invalidation or reinforcement.

6.3 Intent

An intent is modeled as

$$I = \langle \Sigma_I, g_I, h_I, \beta_I, \tau_I, \alpha_I \rangle$$

where:

- Σ_I is the declared scope,
- g_I is the goal predicate,
- h_I is the set of invariants,
- β_I is the budget vector, including time, cost, blast radius, and review requirements,
- τ_I is the trust envelope,
- α_I is the acceptance policy.

6.4 Closure

Given intent I, the closure engine computes a task-specific context slice

$$\mathcal{C}(I, \mathcal{W}_t) \subseteq G_t \cup E_t \cup M_t$$

using a relevance function

$$r(v, I) = \lambda_s s(v, I) + \lambda_c c(v, I) + \lambda_t t(v, I) + \lambda_m m(v, I) + \lambda_h h(v, I)$$

where the terms correspond to structural proximity, causal dependency, temporal relevance, semantic similarity, and human or organizational priors. Expansion continues until either marginal utility drops below threshold or the closure budget is exhausted. The key property is that closure is *intent-conditioned* and *bounded*.

6.5 Transition contracts

A transition contract is

$$\theta = \langle q, \phi_{pre}, a, \phi_{post}, \kappa, \lambda, \mathcal{O} \rangle$$

where:

- q selects the target entities,

- ϕ_{pre} are preconditions over the current Worldline state,
- a is the action semantics or compiled actuation plan,
- ϕ_{post} are postconditions,
- κ is the compensation strategy,
- λ is the capability envelope,
- $\mathcal{O}$ is the set of observers that verify effects.

A plan for intent I is a sequence or partial order of contracts

$$\Pi_I = \langle \theta_1, \theta_2, \ldots, \theta_n \rangle$$

executed on a branch of the Worldline.

6.6 Receipts and replay

Execution of θ_i yields a receipt

$$\rho_i = \langle I, \theta_i, \Delta_i, \upsilon_i, \pi_i, \chi_i, t_i \rangle$$

where Δ_i is the state delta, υ_i is observer output, π_i is policy evidence, χ_i is terminal status, and t_i is timestamp. Replay re-executes Π_I from a checkpointed state with original closure, capability envelope, and observation schedule, allowing bounded-deterministic diagnosis.

7 System Architecture

Figure 1 presents the overall architecture. The design is layered to separate observation, state, reasoning, control, and human interface, but those layers share a common substrate.

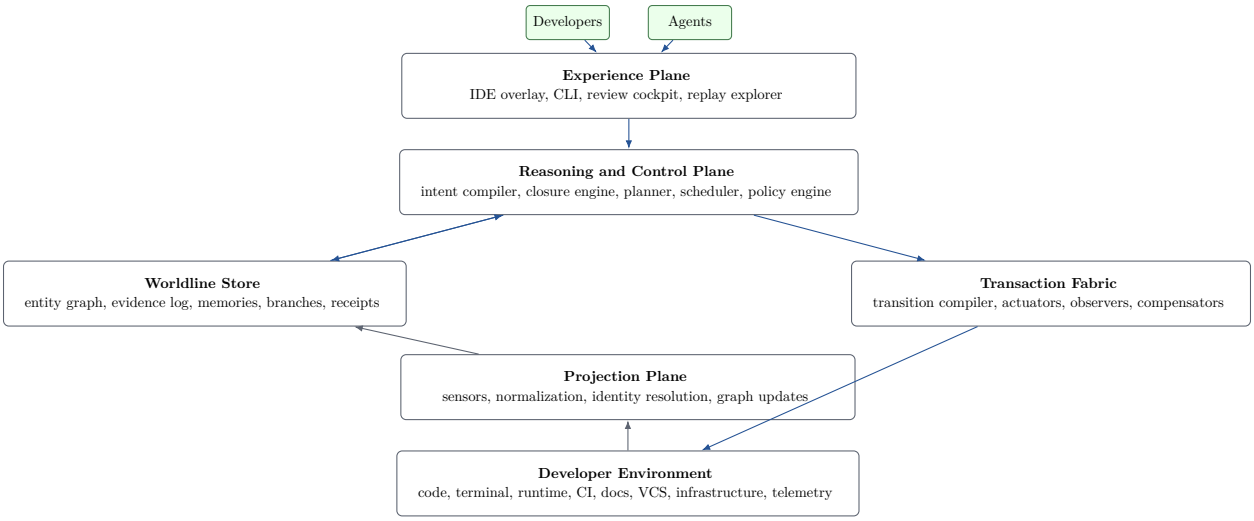

Figure 1: The Worldline architecture. The environment is continuously projected into a stateful substrate, and all agent action flows through a transaction fabric that operates on world state rather than direct tool selection.

7.1 Projection plane

The projection plane is responsible for turning raw environmental signals into stable world state. It includes collectors for repositories, file systems, editors, shell sessions, process trees, test runners, CI pipelines, logs, traces, tickets, documents, knowledge bases, and infrastructure APIs. Its two most important functions are *identity resolution* and *evidence management*. Without identity resolution, the same service may appear as a repository, a deployment, a dashboard label, and a package namespace without ever becoming one thing. Without evidence management, the system cannot distinguish observations from guesses.

7.2 Worldline store

The store is not merely a graph database. It combines several state forms:

- a typed entity graph for operational structure,
- an append-only evidence ledger,
- a branchable change log,
- a receipt ledger for execution,
- checkpointed snapshots for replay,
- anchored memory records with validity rules.

This mixed representation matters. Graphs are good for structure. Logs are good for causality. Snapshots are good for replay. Memories are good for continuity. No single storage idiom is sufficient on its own.

7.3 Reasoning and control plane

The control plane includes intent compilation, closure construction, planning, scheduling, work decomposition, policy evaluation, and merge arbitration. Its job is not to replace model intelligence. Its job is to provide a stable substrate within which model intelligence can behave consistently.

7.4 Transaction fabric

The transaction fabric compiles abstract transition contracts into actuator-level operations. It may invoke shells, editors, test runners, deployment APIs, or existing tool servers, but it does so under a contract with observers and receipts. This is where the architecture cleanly subsumes tool protocols without centering them.

7.5 Experience plane

Developers interact with Worldline through several surfaces: an IDE overlay that highlights closure and risk, a command-line interface for branch and replay operations, a review cockpit for approving envelopes and merges, and a timeline explorer for understanding causal episodes.

8 Context Extraction Model

Worldline separates *projection* from *closure*. Projection builds the global operational twin. Closure extracts the task-relevant slice. This distinction is essential because many current systems conflate indexing with context selection.

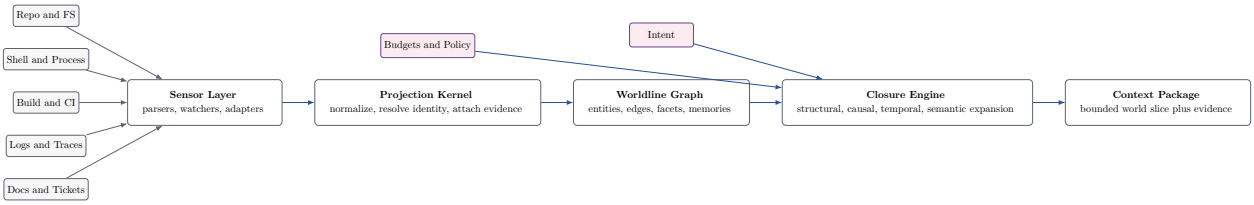

Figure 2: Worldline context extraction. Raw environment signals are projected into a canonical world model. Intent-conditioned closure then constructs the bounded context package used for planning and execution.

8.1 Projection is continuous

Projection is not invoked only when the agent asks. The harness continuously learns the environment. Repositories are parsed in the background. Branch topology is known before an action request. CI results are already attached to build targets. Runtime failures are already linked to services and releases. This shifts work from prompt time to system time.

8.2 Closure is relevance with structure

Closure uses multiple edge families:

- **Structural edges:** imports, ownership, build dependency, deployment dependency, package graph, test coverage.
- **Temporal edges:** recent changes, deployment windows, failure bursts, recent conversations, incident overlap.
- **Causal edges:** stack traces, trace spans, command outcomes, dependency breakage, issue lineage.
- **Semantic edges:** doc similarity, symbol meaning, natural language references, design notes.
- **Normative edges:** policies, coding standards, secret boundaries, review rules.

An intent does not need all reachable state. It needs the *smallest sufficient* operational slice. That is why closure is bounded by utility and budget. Worldline should improve agent autonomy without exploding context windows.

8.3 Epistemic clarity

Because every fact is evidence-backed, closure can preserve uncertainty. The context package can say, for example, that a deployment correlation is inferred with medium confidence from traces and timestamps, while a test failure is directly observed from CI output. This is more valuable than collapsing everything into flat text.

9 Execution Model

Worldline changes execution from ad hoc calls into verified transitions. Figure 3 shows the lifecycle.

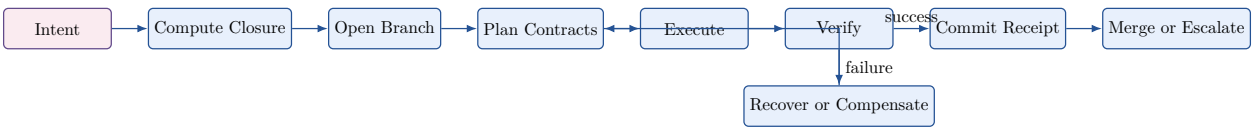

Figure 3: The Worldline execution lifecycle. Agents work on branches of world state, execute transition contracts through the transaction fabric, verify postconditions, and either merge, compensate, or re-plan.

9.1 Intent compilation

Execution begins with a typed intent, not raw natural language. Natural language can still be the authoring surface, but the system compiles it into explicit scope, goals, invariants, budgets, and trust requirements. This closes a common gap between developer meaning and agent execution.

9.2 Branches as execution spaces

A branch is more than a git branch. It is a speculative timeline over the full Worldline. It may include code changes, terminal state, observed build failures, generated hypotheses, partial deployments, and memory updates. This means an agent can experiment, verify, and recover without corrupting the canonical world state.

9.3 Transition compilation

Each transition contract is compiled into concrete operations. For a code edit, the actuator may be an editor integration plus a formatting pass plus a test invocation. For a runtime diagnosis, the actuator may be a trace query, log slice, or container inspection. For an infrastructure change, the actuator may be an IaC update followed by plan verification. The key is that the agent asks for *the transition*, not for specific low-level APIs.

9.4 Observers and postconditions

Observers verify whether the world moved as intended. They are independent of the actuation path. This separation prevents a common problem in current systems where the same tool that makes a change is implicitly trusted to describe its correctness. A deployment change may be observed by health checks, error budget monitors, and trace deltas. A code change may be observed by test results, compile status, and linter output.

9.5 Compensation and recovery

If postconditions fail, the runtime attempts compensation according to policy. Compensation may revert a file patch, restore a snapshot, halt a rollout, or quarantine the branch for human review. Recovery is first-class, not an afterthought.

10 State and Memory Model

Worldline distinguishes between *world state*, *episodic history*, and *anchored memory*.

10.1 World state

World state is the current best-known model of the environment. It is live, partially observed, and evidence-scored. The system must tolerate uncertainty and stale edges. This is why each claim carries epistemic status.

10.2 Episodic history

Episodes record what happened: which intents were active, which closures were used, which transitions executed, what observers saw, what branches were merged, and how recovery proceeded. Episodes make autonomous work inspectable.

10.3 Anchored memory

Memory is represented as typed records attached to entities or episodes. Useful memory categories include:

- **Factual memory:** durable facts such as ownership or service boundaries.
- **Heuristic memory:** repair patterns, migration rules, or known failure signatures.
- **Normative memory:** preferences, coding conventions, review practices.
- **Social memory:** who tends to approve, who owns a risk domain, who has historical context.
- **Incident memory:** what fixed a similar failure before and under what conditions.

Each memory record contains anchors, provenance, freshness policy, and a validity predicate. For instance, a memory that a certain migration script is safe may become invalid if the build graph or schema version changes beyond threshold. This is a direct answer to memory drift.

10.4 Memory invalidation

A crucial part of the model is forgetting. When the Worldline changes materially, some memories should weaken or expire. The system therefore supports invalidation triggers, such as dependency tree divergence, major release boundaries, policy changes, or contradictory evidence. An agent memory system that cannot forget is eventually untrustworthy.

11 Permission and Trust Model

Permission must be stated in the language of world transitions. This is one of the strongest departures from tool-centric systems.

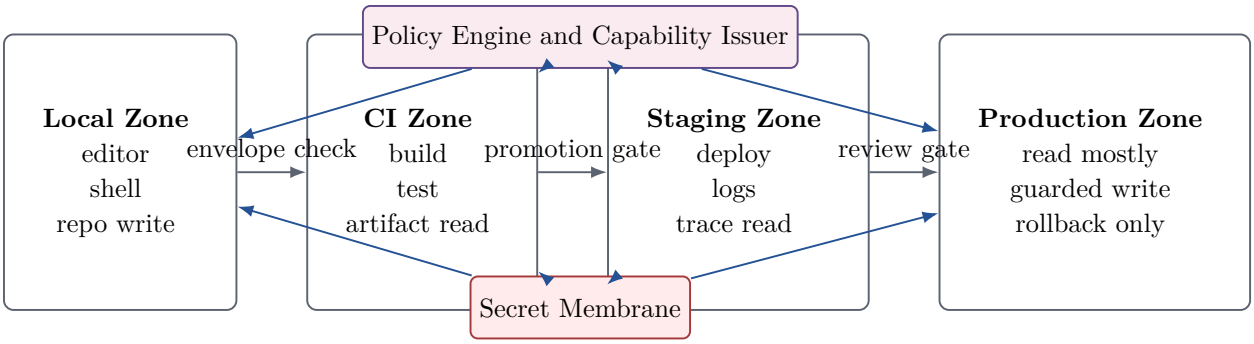

Figure 4: Trust boundaries in Worldline. Capability envelopes govern which transition classes may cross which zones. Secrets are handled through a membrane that exposes policy-checked handles rather than raw values to the broader Worldline.

11.1 Capability envelopes

A capability envelope answers not only *who* may act, but also *how*, *where*, *under what budgets*, and *with what review requirements*. An envelope can state, for example:

> The agent may modify application code under repository `checkout/`, may run local and CI tests, may query staging traces, may not cross into production write, may not materialize secrets, must request human approval before schema changes, and must stop if more than 12 files or 2 services are affected.

This level of expression is impossible to capture with endpoint allow-lists alone.

11.2 Trust zones

Worldline models explicit zones such as local workstation, repository write, CI, staging, production read, production write, internet egress, and secret domains. Policies define which transitions are admissible across zone boundaries.

11.3 Secret membrane

Secrets must not become generic facts in the Worldline. The secret membrane exposes references, handles, or derived predicates instead of raw values. For example, the runtime can verify that an environment variable is present or that a credential rotates successfully without leaking the credential into the general memory graph.

11.4 Proof-carrying actions

Worldline does not require formal proofs in the theorem-proving sense, but it does require *action certificates*. Each receipt includes machine-checkable evidence that preconditions, policy checks, and

postcondition observers were satisfied or violated. This creates a higher-trust execution trail than plain logs.

12 Observability, Replay, and Debugging

Observability must explain the agent's behavior at the level developers actually care about: what the agent believed, which entities it considered relevant, which transitions it attempted, what changed, and where divergence occurred.

12.1 Receipts as a causal ledger

Receipts form a causal ledger. A developer can query:

- Why did the agent edit this file?
- Which evidence linked this file to the declared intent?
- Which observer failed after the deployment step?
- What was the trust envelope at the moment of action?
- Which recovery pathway executed after the failed test?

The answer is computed from receipts and closure state, not reconstructed from chat.

12.2 Replay

Replay uses checkpointed state plus original receipts to reconstruct an episode. Figure 5 shows the model.

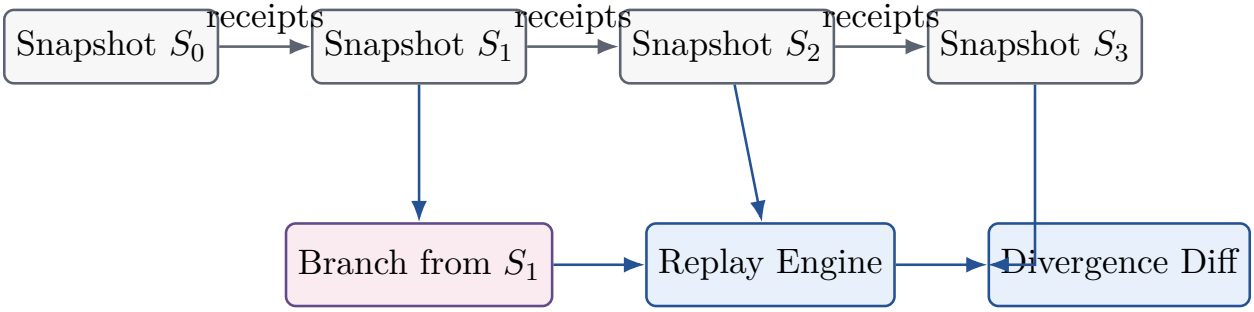

Figure 5: Replay in Worldline. Checkpointed snapshots and receipts allow re-execution or diagnosis from an earlier state. Divergence can be inspected as a state diff rather than guessed from terminal history.

Replay may be deterministic, bounded-deterministic, or counterfactual. Deterministic replay is possible when all relevant inputs are captured or stubbed. Bounded-deterministic replay is used

when external systems have changed but the harness can preserve the closure, policies, and action ordering. Counterfactual replay explores "what if" branches, such as trying a different patch or deployment ordering.

12.3 Divergence diagnosis

A powerful feature of a Worldline harness is that divergence has structure. The system can report whether failure arose from stale closure, unsatisfied preconditions, unreliable actuators, observer disagreement, environmental nondeterminism, or policy denial. This is a significantly higher level of introspection than raw task failure.

13 Multi-Agent Coordination

Multi-agent systems need a shared coordination substrate. Worldline provides it through branch-based work contracts rather than unstructured chatter.

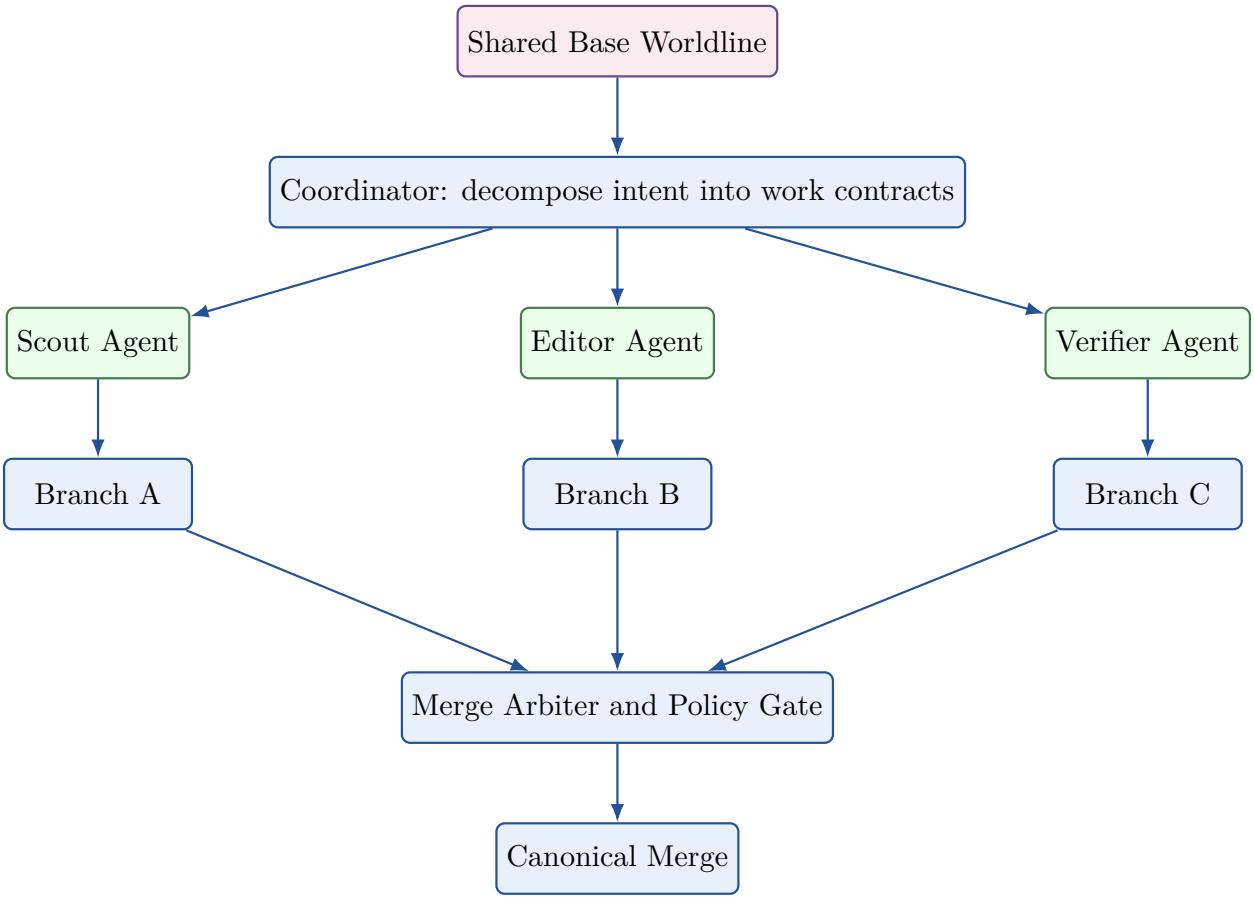

Figure 6: Multi-agent coordination in Worldline. A coordinator decomposes an intent into work contracts, assigns branch spaces, and merges results through an arbiter that enforces policy, conflict rules, and acceptance predicates.

13.1 Work contracts

A work contract is a sub-intent with explicit interface conditions: input closure, output schema, acceptance predicate, budget, dependency set, and capability envelope. This sharply improves decomposition quality. An agent is not merely told "investigate the bug." It is given an operational contract.

13.2 Branch isolation

Each agent works on its own branch of the Worldline. Branches may overlap in scope, but conflicts are explicit and mergeable because the underlying world model is structured. For example, two agents can both inspect the same service while only one is authorized to mutate deployment state.

13.3 Merge arbitration

The merge arbiter evaluates acceptance predicates, policy compliance, observer results, and entity-level conflicts. In simple cases this resembles version control merge. In richer cases it resembles transaction arbitration over operational state, such as choosing which runtime diagnosis becomes canonical or whether a deployment rollback branch supersedes a code-fix branch.

13.4 Long-horizon continuity

Because Worldline preserves shared state, agents can leave and re-enter work without relying on long prompt transcripts. The branch itself is the handoff artifact.

14 Failure and Recovery Model

Autonomous systems are only as useful as their failure semantics. Worldline treats failure as a first-class state transition outcome.

14.1 Failure classes

Failures are categorized into at least five classes:

1. **Context failure:** the closure omitted a relevant entity or over-trusted stale evidence.
2. **Planning failure:** the chosen transition sequence was invalid or incomplete.
3. **Actuation failure:** the low-level operation did not execute as intended.
4. **Verification failure:** the transition executed but postconditions were not met.
5. **Policy failure:** the action was denied, exceeded budget, or crossed a trust boundary.

This classification allows recovery to be precise.

14.2 Recovery pathways

Recovery pathways are modeled objects. They may include retry with refreshed closure, compensation, rollback, branch quarantine, human escalation, or alternate plan selection. Because recovery is captured in receipts, future agents can learn from past failures in a bounded and trustworthy way.

14.3 Leases and heartbeats

For long-running tasks, each active branch carries a lease. Agents must renew the lease via progress receipts. If the lease expires, the coordinator can reassign or freeze the branch. This addresses silent abandonment in multi-agent workflows.

15 Developer Experience Model

Worldline must not feel like a systems paper trapped in a data model. It must improve day-to-day developer life. Figure 7 sketches the workflow.

Figure 7: A simplified developer workflow in Worldline. The developer works in terms of intent, closure, envelopes, and receipts rather than manually curating prompts and post hoc logs.

15.1 The human interface

The developer experience should center on a few high-leverage concepts:

- **Intent authoring:** express outcome, scope, invariants, and budget.
- **Closure inspection:** inspect the world slice the agent will use.
- **Envelope approval:** approve or tighten the transition permissions.
- **Receipt review:** understand exactly what happened and why.
- **Replay and branch diff:** debug or compare alternative episodes.

These are far more legible than a chain of prompts plus tool logs.

15.2 A possible command surface

A reference CLI might include commands such as:

```
wl intent create "Upgrade billing service to API v3 without p95 regression"
wl closure inspect --intent billing-v3
wl branch open --intent billing-v3
wl run --intent billing-v3 --agent editor
wl replay --branch billing-v3/fix-attempt-2
wl explain --receipt rcp_01482
wl merge --branch billing-v3/fix-attempt-2
```

The important point is not the exact syntax. It is the user model. Developers reason about world branches and verified changes, not about which tools were called in which order.

16 Case Studies

This section presents scenario-based case studies. They are intended to illuminate the architecture rather than to claim empirical results.

16.1 Case study 1: framework migration across code, tests, and docs

Consider a migration from framework version N to $N + 2$ across a large monorepo. A tool-centric agent can call search, edit, and test endpoints, but it struggles to retain the full migration state over time. Worldline represents the migration as an intent with explicit invariants, such as passing target test suites and preserving public API contracts. The closure includes dependency graphs, migration guides, affected build targets, prior similar edits, known incompatibilities, and owner-specific review rules. Multiple agents can then branch into package-level remediation, documentation updates, and compatibility verification. Their results merge against the same operational twin.

16.2 Case study 2: flaky integration test diagnosis

A flaky test is rarely a text-local problem. It can involve timing, deployment history, trace anomalies, recent commits, or hidden shared-state assumptions. In Worldline, the closure for the intent "stabilize checkout integration test" naturally includes test history, runtime traces during failures, suspect services, recent dependency shifts, and past incidents with similar signatures. A scout agent explores causal hypotheses on one branch. An editor agent prepares a code fix on another. A verifier agent reproduces the failure under controlled branch snapshots. The developer reviews receipts rather than a maze of logs.

16.3 Case study 3: incident hotfix with guarded production scope

Suppose staging reveals that a recent release caused a read-path regression. The organization allows production read access to agents but restricts write access to human-approved rollback operations. Worldline can encode this directly. The agent closure includes the deployment diff, relevant traces, configuration drift, and rollback playbook. The transition fabric permits log and trace queries,

patch preparation, and rollback simulation, but production write remains behind a review gate. The result is more autonomy where safe, without pretending that all actions are equally permissible.

16.4 Case study 4: secrets rotation and policy hardening

Secrets rotation is an instructive stress case because the agent must reason across config, runtime, CI, deployment, and policy without leaking secret material. Worldline handles this through the secret membrane, which exposes presence and validity predicates instead of raw values. The rotation intent can require that all dependent services verify connectivity, all CI jobs consume the new handle, and no receipt persists secret content. A purely tool-centric design makes this far harder to guarantee.

17 Evaluation Framework

The architecture proposed here is ambitious. It therefore requires a serious evaluation framework.

17.1 Metrics

Table 2 lists representative metrics.

Table 2: Representative evaluation metrics for Worldline

Metric	Meaning
Closure Recall	Fraction of task-relevant entities present in the closure, measured against expert-labeled operational slices.
Closure Precision	Fraction of closure entities that prove relevant during execution, measuring over-expansion.
Replay Fidelity	Agreement between original and replayed outcomes, including postcondition and receipt consistency.
Permission Precision	Degree to which capability envelopes allow necessary actions while rejecting unnecessary or policy-violating ones.
Long-Horizon Success	Task completion rate for multi-step tasks requiring more than one branch or session.
Recovery Half-Life	Time from detected failure to safe compensated or escalated state.
Receipt Completeness	Fraction of meaningful actions with causal receipts sufficient for explanation and audit.
Merge Quality	Rate at which multi-agent branches combine without hidden conflict or semantic loss.
Trust Lift	Human-rated confidence in agent behavior compared with a tool-centric baseline.

17.2 Benchmark families

A robust benchmark suite should include at least:

- **MigrationBench:** large dependency or framework migrations with cross-cutting edits.
- **FlakeBench:** flaky test diagnosis with runtime and temporal context.
- **HotfixBench:** guarded incident response spanning code and deployment.
- **RecoveryBench:** induced actuator, observer, and policy failures.
- **MemoryBench:** long-horizon continuity across sessions and environment drift.
- **CoordBench:** multi-agent decomposition, branch merge, and handoff quality.

17.3 Baselines

A fair baseline should compare Worldline to strong tool-centric systems, not straw men. At minimum, baselines should include:

1. an agent with direct tool registry access,
2. the same agent plus retrieval-based memory,
3. the same agent plus scripted orchestration and approval gates.

The question is not whether Worldline can invoke tools. It obviously can. The question is whether Worldline's stronger state model materially improves autonomy, reliability, and trust.

18 Comparison with MCP-Style Systems

The comparison is best understood at the architectural level. Figure 8 illustrates the shift, and Table 3 summarizes it.

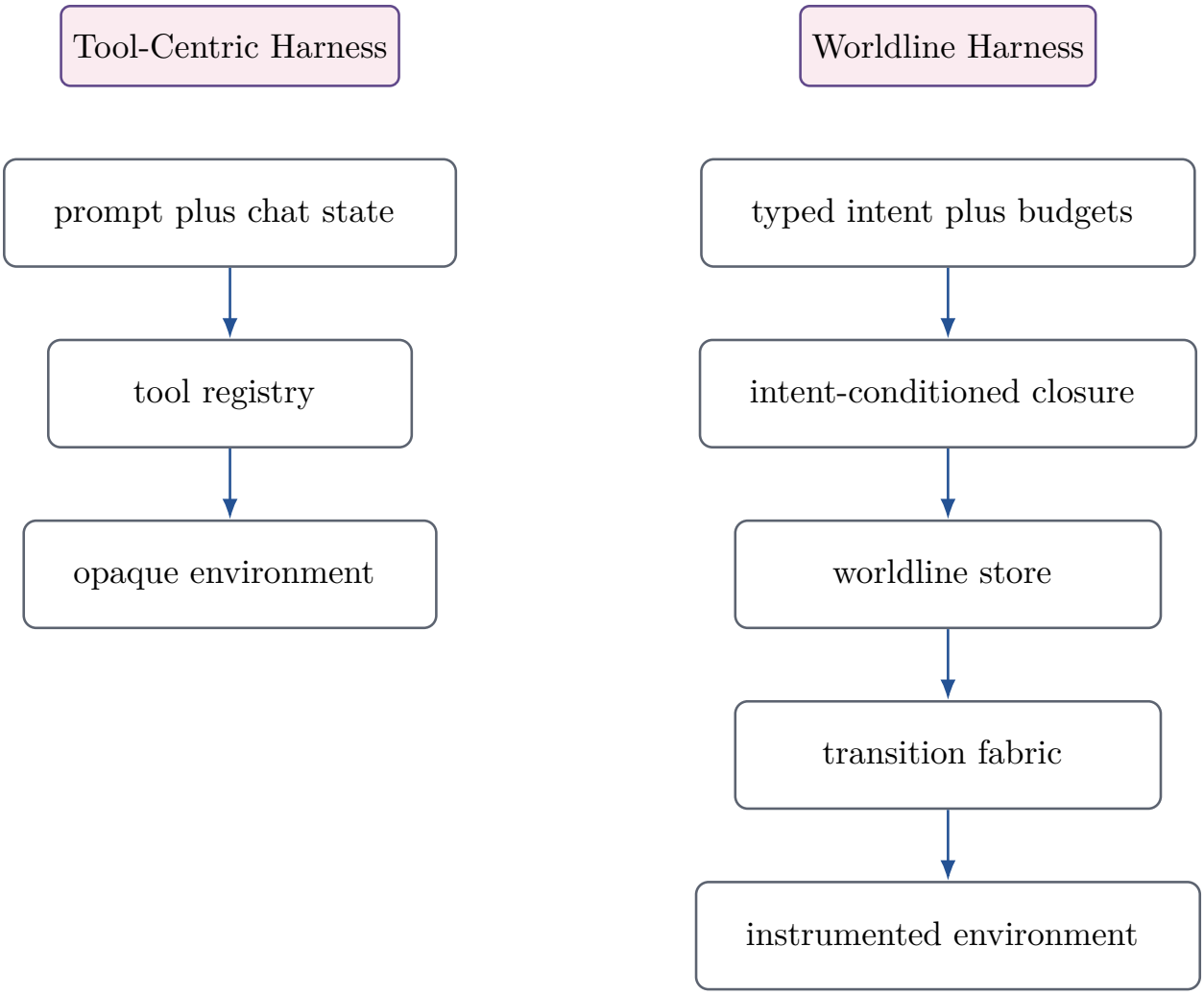

Figure 8: Architectural contrast. Tool-centric systems place an interface registry between the agent and an opaque environment. Worldline inserts a continuously maintained operational twin and a transaction fabric between intent and execution.

Table 3: Conceptual comparison between MCP-style systems and Worldline

Dimension	**MCP-style systems**	**Worldline**
Primary abstraction	Callable tools and schemas	Executable world model and transition contracts
Context model	Pulled on demand from tools	Continuously projected, intent-scoped closure
Memory	External notes, transcripts, or retrieval stores	Anchored memory tied to entities and episodes
Execution semantics	Imperative calls	Preconditioned, observed, receipt-bearing transitions
Permissions	Endpoint or tool level	Capability envelopes over scope, zones, and budgets
Replay	Limited session reconstruction	Checkpointed, branch-based replay and divergence analysis
Observability	Tool logs	Causal receipts from intent through recovery
Multi-agent coordination	External orchestration	Shared world branches and work contracts
Role of existing tools	Primary programming interface	Leaf actuators beneath the runtime

The most important sentence in this comparison is the last row. Worldline does not deny the usefulness of tool protocols. It demotes them. They become part of the actuation layer in the same way device drivers sit beneath an operating system.

19 Product Thesis and Adoption Path

A serious architecture should imply a serious product.

19.1 Product thesis

The company that controls the Worldline for software work controls the trust layer for autonomous engineering. This is strategically stronger than controlling one more code assistant because it becomes the system of record for how agents understand, change, and explain the software world. It sits at the seam between IDEs, source control, CI, runtime telemetry, knowledge systems, and infrastructure. That seam is where durable value accumulates.

The defensibility comes from five sources:

- **Cross-layer data network effects:** the Worldline improves as more environment surfaces are projected.

- **Policy embedding:** organizations codify trust, review, and blast-radius policy directly into the runtime.
- **Operational memory:** receipts and anchored memories form a proprietary corpus of engineering episodes.
- **Ecosystem position:** third parties can write sensors, facets, observers, and actuators against the Worldline substrate.
- **Switching costs:** once a team depends on replay, receipts, and branchable multi-agent state, returning to raw prompt orchestration is painful.

19.2 Adoption path

The adoption path should be progressive.

Phase 1: Local mode. Start with repository, shell, test runner, and git integration. Deliver closure, receipts, and replay for single-repo workflows.

Phase 2: Team mode. Add CI, build graph, and shared memory. Deliver better diagnosis and review.

Phase 3: Runtime mode. Add logs, traces, deployments, and incident workflows. Deliver cross-layer closure and guarded rollouts.

Phase 4: Policy mode. Add capability envelopes, secret membrane, compliance receipts, and org-level approval rules.

Phase 5: Coordination mode. Add multi-agent work contracts, merge arbitration, and workload-level scheduling across teams.

This path matters because it avoids the trap of requiring a full organizational platform bet before the first unit of value appears.

20 Implementation Sketch

Worldline can be implemented incrementally. A practical MVP would contain the following components.

20.1 Local runtime

A local daemon watches file system changes, git state, terminal transcripts, and test results. It stores an append-only event log plus snapshots and a typed graph in a local database. This is sufficient to create branchable replay for a single repository.

20.2 Closure engine

The closure engine uses static analysis, test coverage, recent diffs, semantic indexing, and terminal evidence to build intent-scoped packages. It should expose both a machine-readable schema and a developer-facing explainer.

20.3 Transition fabric

The transition fabric starts with a small set of compiled transition types: read, edit, test, build, search, inspect process, inspect trace, open issue, and propose patch. More types can be added over time.

20.4 Receipt ledger

Every transition attempt emits a receipt, even if denied or failed. This ensures the runtime can explain inaction as well as action.

20.5 Cloud control plane

A team deployment adds identity, policy synchronization, shared memory, runtime integrations, and multi-agent coordination. The architecture remains local-first because the local daemon can continue to operate when disconnected.

21 Limitations

Worldline is not free. It introduces real costs and open tensions.

21.1 Projection cost

Continuously projecting the environment consumes compute, storage, and integration effort. A naive implementation could become heavy. Practical systems will need progressive fidelity, selective sensor activation, and intent-aware caching.

21.2 Ontology risk

Any world model reflects an ontology. If the ontology is too rigid, it will fail to capture important project-specific meaning. If it is too loose, it will collapse into untyped mush. A successful implementation must support extensible schemas without giving up transactional semantics.

21.3 Replay limits

Perfect replay is impossible for many real systems, especially when external services, nondeterministic timing, or unrecoverable side effects are involved. Worldline therefore offers bounded-deterministic replay, not a universal guarantee.

21.4 Policy burden

Fine-grained capability envelopes are powerful, but they also create authoring burden. Organizations may initially misconfigure policies or over-constrain agents. Good defaults and policy simulation are required.

21.5 Privacy and concentration risk

A rich Worldline can become a sensitive concentration point for source code, traces, team knowledge, and operational history. Product design must therefore prioritize local-first storage, secret membranes, selective retention, and clear data governance.

22 Future Work

Several research directions follow naturally.

First, closure quality could be improved through learned relevance functions trained on successful engineering episodes.

Second, merge arbitration could benefit from richer semantic conflict models that reason beyond files and into service contracts, runtime risk, and policy implications.

Third, action certificates could be strengthened into more formal proof objects for especially sensitive transitions.

Fourth, the Worldline substrate could support simulation modes in which agents evaluate hypothetical refactors or migrations before acting.

Fifth, a mature ecosystem of third-party facets and observers could turn the Worldline into a category-level platform rather than a single product.

23 Conclusion

The dominant mental model in agentic developer tooling today is that the environment is a set of tools and the agent is a policy for invoking them. This paper argues that the mental model is wrong at its foundation. Software work is not primarily about calls. It is about transforming a live, constrained, partially observed world. The primitive should therefore be that world.

Worldline is an attempt to define such a primitive. It turns the developer environment into a

causally versioned operational twin. It gives agents intent-scoped closure instead of brittle prompt assembly. It expresses action as verified transition contracts instead of raw calls. It grounds memory in entities and episodes. It makes permissions explicit in the language of scope, budget, and trust zones. It makes multi-agent work branchable and mergeable. It makes replay and recovery native.

If this architecture is correct, then tool-call protocols become what they should have been all along: useful integration adapters beneath a much stronger control plane. Developers will not look back on tool servers as the final form of autonomous software work. They will look back on them as the stage before the software world itself became programmable.

A Appendix A: Example Schemas

The following sketches the kind of typed objects a Worldline runtime might persist.

```
Intent {
  id: string
  summary: string
  scope: [entity_ref]
  goals: [predicate]
  invariants: [predicate]
  budgets: {
    time_minutes: int
    max_files_changed: int
    max_services_touched: int
    review_required: bool
  }
  trust_envelope: {
    zones_read: [zone]
    zones_write: [zone]
    allow_secret_materialization: false
  }
}

TransitionContract {
  id: string
  target_query: world_query
  preconditions: [predicate]
  action: action_plan
  postconditions: [predicate]
  observers: [observer_ref]
  compensation: [action_plan]
  capability_envelope: envelope_ref
}

Receipt {
```

```
  id: string
  intent_id: string
  transition_id: string
  branch_id: string
  before_snapshot: snapshot_ref
  after_snapshot: snapshot_ref
  delta: [entity_delta]
  observer_results: [observer_result]
  policy_checks: [policy_result]
  outcome: success | failed | denied | compensated
}
```

B Appendix B: Example Episode

A typical episode may unfold as follows.

1. The developer creates an intent to stabilize a flaky integration test within one service boundary.
2. The closure engine returns a bounded slice containing the failing test, suspect modules, relevant runtime traces, recent commit history, and past incident notes.
3. A scout agent forms a hypothesis that a timeout threshold is coupled to a queueing delay introduced by a dependency bump.
4. An editor agent opens a branch and prepares a code patch plus a tighter retry policy.
5. A verifier agent replays the failure scenario against a checkpoint and observes reduced flakiness but increased latency.
6. The planner rejects merge because a declared invariant forbids latency regression.
7. A second patch is proposed, verified, and merged after human review of the receipt chain.

This episode is not just text history. It is a structured branch of the Worldline and can be replayed, audited, and used to improve future closure and planning.

References

[1] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[2] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

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

[4] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[5] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.