A Transaction Log for Source Control

Part I of *Source Code as a Transactional Abstraction*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

October 2026

GrokRxiv:2026.10.transaction-log-source-control [cs.DC] 20 Oct 2026

Abstract

Source code and source control can now be treated as transactional abstractions: a repository is a database, a change is a transaction with a read set and a write set, landing is a commit under optimistic concurrency, and a version is a snapshot. This paper gives the transaction model on which that view rests. In Git, when many always-online agents write against one moving branch, every lost compare-and-swap race on the branch costs a rebase, a retest and a retry. In the model, commits are built on the server and recorded without moving any ref. A landing validates the change against every log entry written since its base and appends one hash-chained entry in the same atomic step as the ref update, performed by a serialized authority per repository. We prove replay determinism, snapshot immutability, refs that advance only along first-parent chains, and serializability, relative to declared read sets, of landings that need no line-level merge. The companion paper's independence condition is exactly the condition under which two changes land without merging in either order. The model's known gap is in group landing: a combiner that validates batch members only against the log admits write skew between them, and we give the check that removes it. In a reference implementation, the merge matched `git merge-file` on 300 of 300 random merges, and a changeset landing path landed 1,000 of 1,000 concurrent writers in each of six hosted bursts; a sustained landing rate remains unmeasured.

1 Introduction

Source code and source control are now transactional abstractions. A repository is a database whose state is a set of refs over immutable objects. A change is a transaction that reads part of the source and writes another part. Landing a change is a commit under optimistic concurrency control, and a version of the repository is a snapshot. This paper lays the base layer of that view: a transaction model for source control, with its log, its snapshots, its isolation level, its validation rule and its serializability guarantee, together with the place where the implemented system falls short of the model.

We argue that this is the next step in how code is managed, and that data took the same step first. Data lakes began as files on object stores; concurrent writers and derived computation broke file-level semantics, and the field moved to tables with a transaction log, snapshots and optimistic concurrency [3, 21], with partitioned computation running over those snapshots [32]. Agents now put source control under the same pressure, by multiplying concurrent writers and the volume of derived work. Scion's published design makes the same move: it creates changesets on the server as entries in a write-ahead log and keeps the ref update as a compare-and-swap on one database row,

separate from commit creation [2]. This series supplies foundations for that move, and the proofs and measurements of Parts I to III are the evidence for the thesis.

Source control was designed for people. A developer clones a repository, edits files on a local disk, builds a commit, and pushes it. The server sees a finished commit and one request to move a branch from an old value to a new one. If the branch moved in the meantime, the push is refused and the developer fetches, rebases, retests and tries again. This loop is cheap when conflicts are rare and the writer is patient. It is expensive when writers are numerous, short-lived and always online, and when each of them writes a few files against the same branch head.

That situation now exists. Anthropic reports that its merge rate grew from about 2,500 code changes per day in May 2026 to more than 30,000 per day by September, and that git traffic, mirror drift and merge coordination became its largest development bottleneck [2]. Its response, a system called Scion, replaces commits created on clients with changesets created on the server as entries in a write-ahead log, and moves a ref by a compare-and-swap on a single database row that is not involved in creating or fetching commits [2].

This paper studies the same problem from the side of data systems. We treat a repository the way Delta Lake treats a table [3]. Immutable objects play the role of data files. An append-only, hash-chained transaction log records every ref change. Checkpoints summarize the log so that any version can be reconstructed cheaply. Readers pin a version and see a snapshot. Writers validate optimistically against what has been committed since they read. The analogy is close enough to transfer the main correctness arguments, and it fails in two places that matter: the snapshot of a repository is a map of refs, not a set of files, and two writers that touch the same file can often both succeed through a three-way merge, which has no counterpart in a table.

The architecture needs four primitives: an immutable, content-addressed object store; a serialized authority per repository that holds durable state and executes each command atomically; a durable queue with retries; and stateless compute. The log, the refs and the validation logic live in the serialized authority, which supplies the atomic step that Delta Lake obtains from put-if-absent writes to an object store [3]. Every theorem in this paper is stated over these primitives. Scion's ref update, a compare-and-swap on one Spanner row [2, 10], shows that a serialized authority of this kind is available outside the platform we used.

The reference implementation, YonedaRepo, is first-party work: the author designed and built it. It offers Git semantics (commits, branches, merges, diffs and logs) with Git object identifiers, so every commit it exposes can be fetched and verified by stock Git. It realizes the primitives on Cloudflare, with a Durable Object as the serialized authority [8, 9]; Part III gives the mapping of each primitive to a product and the measured cost of the authority (Part III, Sections 3, 5 and 6) [26]. Part II describes derived data as views maintained from the log [25].

The paper makes five contributions.

1. A formal model of a source repository as a transaction log over content-addressed Git objects, with snapshots, checkpoints and time travel by version or commit (Section 4).

2. Proofs that replay is deterministic, that snapshots are immutable, that refs only advance along first-parent chains in the implemented system, and that no landed change is lost (Section 5).

3. A serializability theorem for landings validated against declared read sets, with a mapping to the validation conditions of Kung and Robinson [23], and an exact characterization of the companion paper's independence condition (Eq. 1 of [24]) as the condition for order-insensitive landing without a merge (Section 6).

4. An analysis of the group-landing combiner that proves it preserves the single-writer property

of the ref update, shows by counterexample that it admits write skew between members of one batch, and gives a sufficient condition that removes the anomaly (Section 7).

5. A statement of the intended changeset landing path, in which the Git commit of a branch is a deterministic function of the log prefix, and a check of which results carry over to it unchanged (Section 8).

The measured evidence in Section 10 is correctness evidence first: Git object identity, agreement of the merge with Git, and concurrent landings without loss. The hosted latency of the write path misses its design target, and we say so there.

2 The coordination problem

2.1 Git's unit of coordination

Git names every object by the SHA-1 digest of a short header and the object's bytes [18]. A commit names one tree and its parents; a tree names blobs and subtrees. The naming makes objects immutable and verifiable by anyone who holds them, which is the property every later design keeps.

Coordination in Git happens at refs. A push sends, for each ref, a command with the old and new object identifiers, followed by a pack of objects; the server unpacks the objects, checks that each ref still has the old value the client named, and updates it [19]. The ref update is therefore a compare-and-swap. Everything before it, including building the commit, computing which objects the server lacks and producing the pack, happens on the client. When the swap fails, all of that work is stale. The client must fetch the new head, rebuild its commit on top of it, rerun whatever checks it trusts, and push again.

For a person, this loop is a minor tax. For a large population of writers that share a branch, the loop becomes the system. Each failed swap converts into a fetch, a rebase and a retry, and the probability of failing grows with the rate at which others land. The cost of a lost race is paid on the client, where the writer cannot see what else is happening, and it is paid in full even when the competing change touched unrelated files.

2.2 Always-online writers

An agent that edits code differs from a person in three ways that matter here. It is always online, so it does not need an offline copy of history. It is short-lived, so a full clone is mostly wasted work. It usually changes a few files and could tell the server what it read and what it wrote. None of these facts is exploited by the push protocol, which assumes that the client holds the history and the server holds only the result.

The Scion post quantifies the consequences at one company: a main-branch history of almost 100 GB, compressed working copies over 25 GB, on the order of 25 million CI checkouts per day, and an estimated 600 PB per day of git traffic if every checkout fetched naively [2]. Several hundred git mirrors were needed to carry the load, and their seeds drifted from the main branch between refreshes [2]. The post also argues that a strict merge queue does not suit agent-scale merging, because many changes land while any one change is being tested [2]. These are reported figures from one organization's production system. We use them as motivation, not as measurements of the system studied here.

Table 1: The four components. Scion's column paraphrases the Scion post [2].

Component	Scion	This architecture
Objects	Content-addressed compressed blobs in cloud storage behind cache tiers	Git blobs, trees and commits in an object store, read through caches keyed by object identifier
Workspace changesets	Write-ahead log entries listing files changed against the first parent	Commits built and recorded by the authority, with their changed paths, before any ref moves
Protected refs	A compare-and-swap on one Spanner row per ref	Refs and one transaction log per repository in the serialized authority
Compute	Server-side services for merge, derived data and checks	Stateless functions for merge, rewrite and search, container compute for checks, and a durable queue for follow-up work

2.3 Requirements

From these observations we draw five requirements for a replacement processing model. They are design requirements, not empirical claims.

D1, server-side changes. A writer sends file changes and receives a commit identifier. The server builds trees and commits and stores the objects.

D2, creation separate from landing. Creating a commit must not move a ref, so that commits can be created in parallel without coordination and checked before they land.

D3, cheap lost races. When the branch moved, the server should decide from recorded metadata whether the change still holds, and merge it without a client round trip when it does.

D4, contention-free reads. Reads should name an immutable version and never wait for writers.

D5, Git semantics. Writers and readers should see commits, branches, merges, diffs and logs with Git's meaning, and a branch should be exportable as Git objects that existing tools can fetch and verify.

Scion meets D1 to D4 with its own changeset format and describes itself as not a general Git replacement [2]. The design here meets D1 to D4 with the log described below. D5 constrains what the log must be able to produce, not what a landing must write: the Git objects of a branch can be built from the log, as Section 3.6 explains.

3 Design

The design has four components, each built on the primitives named in the introduction. Table 1 compares each with its counterpart in Scion. Part III maps the primitives to products in the reference implementation.

3.1 Objects

Objects are Git objects with Git identifiers. The platform computes the identifier of a blob, tree or commit as the SHA-1 of the header `<type> <size>\0` followed by the bytes, the same rule Git uses [18], and its codec reproduces identifiers that Git itself computes with `hash-object`, `mktree` and

`commit-tree` (Appendix A). Packs sent to the object store contain only whole objects; the reader refuses deltified entries.

In the reference implementation at `cfa4845`, Git objects live in a Cloudflare Artifacts repository, with R2 and an edge cache as read-through tiers. Later revisions of the development branch make R2 the store of record and push only protected refs to Artifacts (Part III, Section 3.2).

Every recorded commit is kept reachable in the object store by a hidden ref named after the commit, so that work which no branch points at yet cannot be collected. The only maintenance the platform performs is releasing these hidden refs once a commit is safely reachable from a branch or once unlanded work has expired; Part II analyzes that procedure in its Proposition 12.1.

3.2 Workspace changesets

A writer sends a list of file changes, a parent commit, a base log version, an optional declared read set and a message. The authority computes the new tree by rewriting only the directories on changed paths, asks the adapter for exactly the existing trees it needs, verifies every supplied tree against its identifier, and returns the objects to upload. After the adapter has pushed the objects and read the commit back from the object store, the authority records it. A recorded commit carries its tree, its parents, its base version, its read set and its list of changed paths relative to the first parent. In the words of the code, it is durable, addressable work that no ref points at yet.

The rewrite gives the scaling property Scion asks for: the number of trees read and written depends on the directories along the changed paths, not on the number of files in the repository. Proposition 4.3 states this precisely.

3.3 Protected refs

Refs live in one table inside the repository's serialized authority, next to the transaction log, the checkpoints, the land queue and the table of landings. A landing is one call into the authority. The call reads the ref, validates the change against the log, and either refuses or appends one log entry and updates the ref, all as one atomic command. The contract the design needs from the authority is that its commands behave as if executed one at a time, each committing all of its writes or none. No network input or output happens inside a command: object pushes and reads happen in the adapter before the authority is called. These two properties are the basis of the formal model in Section 4. The reference implementation obtains them from a Durable Object, which runs one request at a time and whose synchronous SQLite transactions are all-or-nothing [8, 9]. A transactional database that offers serializable transactions over the ref row and the log tail would provide the same contract.

The design deviates from Scion in one respect that matters for capacity. Scion places each ref on its own database row [2]. This architecture places all refs of a repository, and the preparation and recording of every commit, behind one authority. Serialization is therefore per repository, not per ref. That is a choice of key for the authority, not a property of the primitive, and a realization could key authorities by ref instead. Part III measures the consequences in its Sections 4 to 6.

3.4 Compute

Diffs, three-way merges, tree rewrites, symbol extraction and search run as stateless functions called from the adapter; in the reference implementation they are Rust compiled to WebAssembly and run in Workers. Checks run in container compute on the exact commit they certify. A durable queue carries derived-data jobs and ref projection to the Git-facing store. None of these components holds

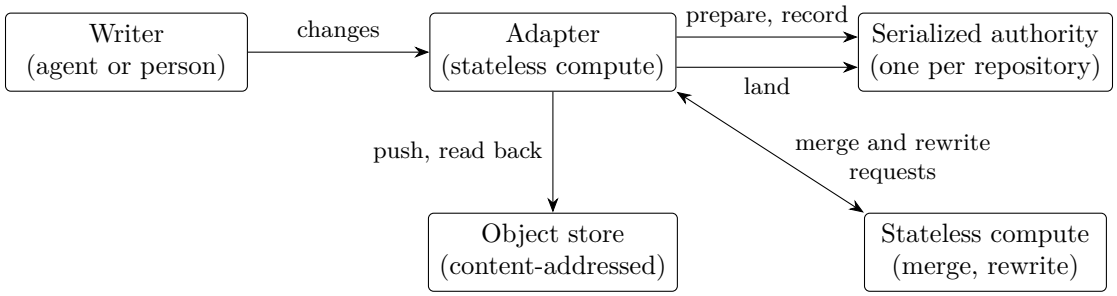


Figure 1: The write path. Creating a commit (prepare, push, read back, record) and landing it are separate calls. Only the land call reads and updates refs, and it performs no network input or output while it holds the authority's transaction.

authority: their outputs are either content-addressed objects, which anyone can verify by hashing, or proposals that the authority verifies before it accepts them.

3.5 Commit creation and ref moves

Figure 1 shows the separation that requirement D2 asks for. A commit is created by four steps that never touch a ref: prepare, push, read back and record. Reviews and check results attach to the commit identifier. Landing is a fifth, separate step. Its gates require the configured approvals and passing results for the required checks, bound to the policy in force; a change whose every file falls in a class the policy marks safe, such as comments only, counts as approved, and an unclassified change never does. Landing then validates and appends. If the branch moved and the change needs a merge, the platform builds a merge commit with its own compute, records it under the principal `platform:merge`, and lands that merge in place of the original commit. Original commit identifiers are never rewritten.

3.6 Realizations of landing

The reference implementation first landed commits: every landing that is not a fast-forward builds a platform merge commit, with new tree objects on every changed path, before the authority appends the entry. This tree-building landing is the path the code evidence of Sections 4 to 7 describes, at revision `cfa4845`, and it was still the landing path of the development branch at revision `03821e0`. Building Git objects on the landing path is a choice of that implementation, not a requirement of D5.

The architecture's intended landing path lands changesets instead. It was first implemented and measured at revision `dd6c9a8` on the branch `claude/native-p95e`, and it is merged into the development branch at revision `fef94e7`. A landing carries a base version, a declared read set and a list of file changes. The authority validates it by path overlap against the log entries since its base, which is a query on the log, and appends one entry; many landings share one authority command. Two paths result. When the changeset's paths are disjoint from every later entry's paths, landing is the validation query and the append, with no file content read. When a later entry wrote one of the same files, landing takes the merge path, which reads the three versions of each shared file and runs the three-way merge. The Git commit of the branch is built afterwards, in the background, as a deterministic function of the log prefix: the same prefix always yields the same commit identifier. Part II treats that projection as a derived view of the log. The cost is that a branch's Git identifier becomes available after its landing returns rather than with it. At `fef94e7`, 1,000 of 1,000 concurrent

writers landed in each of six hosted bursts on the reference implementation. Part III reports its latency and throughput.

Every result of Sections 5 to 7 is stated over log entries, read sets, write sets and validation outcomes. Section 8 checks which of them carry over to changeset landing unchanged and which become properties of the projection.

4 Mathematical framework

This section states the model. Each definition corresponds to a data structure or function in the implementation; Appendix A gives the file and commit for each.

4.1 Objects, trees and commits

Definition 4.1 (Object store)**.** Let Id be the set of 160-bit identifiers. An *object store* is a finite partial map $O : Id \rightharpoonup Type \times Bytes$, with $Type = \{\text{blob}, \text{tree}, \text{commit}, \text{tag}\}$, such that $O(x) = (k, b)$ implies $x = H(hdr(k, b) \,\|\, b)$, where H is SHA-1 and $hdr(k, b)$ is Git's header naming the type k and the length of b. The store only grows: if $O \subseteq O'$ describes two moments in time, every object of O is in O' with the same type and bytes.

Content addressing makes an object immutable by construction. Two stores that hold an identifier hold the same bytes, unless H has a collision. All results below assume that no collision occurs among the objects of one repository; Git makes the same assumption, and its hash transition to SHA-256 would weaken it further without changing any argument here.

A *tree state* τ is a finite map from file paths to blob identifiers and modes, the decoded form of a Git root tree. A *commit* c has a tree state $\tau(c)$ and a list of parents; $\pi_1(c)$ is its first parent, or $\perp$ for a root commit. Every recorded commit also has a *generation* one greater than the largest generation of its parents, so the parent relation is acyclic.

Definition 4.2 (Touched paths)**.** The *change* of a commit c is the set $\Delta(c)$ of paths whose entry differs between $\tau(c)$ and $\tau(\pi_1(c))$, with every path of $\tau(c)$ counted when $\pi_1(c) = \perp$. The platform records with c a set $T(c)$ of *touched paths*, viewed as file nodes of the location tree of Definition 4.4; alongside each it stores an operation (add, modify or delete) and the old and new blob, which the results below do not use.

The authority derives $T(c)$ from the change list it was given and builds $\tau(c)$ by applying exactly that list to $\tau(\pi_1(c))$. Hence $\Delta(c) \subseteq T(c)$ for every commit the platform prepares, including the merge commits the platform builds when landing work on a moved branch: the adapter submits each of them to the same preparation command, with the head as first parent and the merged commits as further parents. A property test checks that the incremental rewrite produces the same tree as building it from scratch.

Proposition 4.3 (Rewrite cost)**.** *Let c change the set of paths P relative to $\pi_1(c)$, and let $D(P)$ be the set of directories that are proper prefixes of some path in P, including the root. Preparing c reads exactly the trees of $D(P)$ that exist in $\tau(\pi_1(c))$ and encodes exactly the trees of $D(P)$ that are non-empty in $\tau(c)$; a directory left empty is removed from its parent instead. The work is $O(\sum_{d \in D(P)} |d|)$, where $|d|$ is the number of entries of directory d in $\tau(\pi_1(c))$ and $\tau(c)$ together, and is independent of the number of files outside $D(P)$.*

Proof. A tree's identifier depends only on its entries. A directory not in $D(P)$ contains no changed path, so its entries and hence its identifier are unchanged, and its parent can reuse the identifier

without reading it. A directory in $D(P)$ contains a changed path, so at least one entry changes and the directory must be re-encoded, which requires reading its existing entries. Re-encoding a directory costs time linear in its entries. Summing over $D(P)$ gives the bound. □

The bound depends on directory fan-out, so a flat directory of many files makes a one-file change expensive. It does not depend on repository size as such. The unit test `rewriting_reads_only_directories_on_changed_paths` checks that an unchanged subtree is never read.

4.2 Locations and access sets

Read and write sets name parts of the source at two granularities: files and directory prefixes, and symbols inside files. We model both as nodes of one tree.

Definition 4.4 (Location tree and overlap)**.** The *location tree* $\mathcal{T}$ is the rooted tree of all syntactically valid locations, whether or not a given tree state contains them. Its nodes are the root, directories, files and symbols; a directory's children are the directories and files named inside it, a file's children are its top-level symbols, and a symbol's children are its nested members. An *access* is a node of $\mathcal{T}$. Write $a \sqsubseteq b$ when a is an ancestor of b or equal to it. Two accesses *overlap*, written $a \bowtie b$, when $a \sqsubseteq b$ or $b \sqsubseteq a$. For sets of accesses, $X \bowtie Y$ holds when some $x \in X$ and $y \in Y$ overlap, and $X \not\bowtie Y$ otherwise.

In the implementation, an access is a string: a path such as `src/lib.rs`, a prefix such as `src/`, or a symbol entry such as `src/lib.rs#Policy::backoff`. Two paths overlap when they are equal or when one is a prefix of the other at a component boundary, so `src/a` and `src/ab` do not overlap. Two symbol entries overlap when they name the same file and one symbol path is a `::` prefix of the other. A path and a symbol overlap when the path overlaps the symbol's file. The code adds one further case, for files whose literal names contain `#`, which can only add overlaps. The implemented relation therefore contains the relation of Definition 4.4. Extra overlaps are the safe direction for the soundness results below, which can only gain conflicts and merges.

Lemma 4.5 (Overlap)**.** *Let* $\mathrm{sub}(a) = \{x \in \mathcal{T} : a \sqsubseteq x\}$. *Then* $a \bowtie b$ *if and only if* $\mathrm{sub}(a) \cap \mathrm{sub}(b) \neq \varnothing$. *The relation* $\bowtie$ *is reflexive and symmetric. If* $a' \sqsubseteq a$ *and* $a \bowtie b$, *then* $a' \bowtie b$.

Proof. If $a \sqsubseteq b$ then b lies in both subtrees, and symmetrically. Conversely, if x lies in both subtrees then a and b are both ancestors of x; the ancestors of a node in a rooted tree form a chain, so a and b are comparable. Reflexivity and symmetry are immediate. For the last claim, $\mathrm{sub}(a) \subseteq \mathrm{sub}(a')$, so a common element of $\mathrm{sub}(a)$ and $\mathrm{sub}(b)$ is also in $\mathrm{sub}(a')$. □

Accesses are granules in the sense of multigranularity locking, where a lock on a node implicitly covers its descendants [6, Section 3.9]; overlap is intersection of that implicit coverage. We validate rather than lock, so no intention modes are needed.

Definition 4.6 (Read and write sets)**.** A change carries a *declared read set* R, a finite set of accesses supplied by the writer, and a *write set*. The write set of a recorded commit c is $\mathsf{W}(c) = T(c)$ at path granularity. When the platform has parsed both sides of a touched file and found the symbols the change affected, it may replace that file by those symbols; the result is the *refined* write set $\mathsf{W}^{\rho}(c)$. A file the platform could not resolve stays a whole-file write.

A tree state assigns *content* to file and symbol locations: a file's content is its blob and mode, a symbol's content is the source text the parser attributes to it, and a location the state does not contain has content $\perp$. Directories have no content of their own. Two tree states τ, τ' *agree on* a

set of accesses X, written $\tau \equiv_X \tau'$, when they assign the same content to every file and symbol location in $\bigcup_{x \in X} \mathrm{sub}(x)$. Because absent locations have content $\perp$, agreement on a directory prefix includes agreement on which files exist below it. Agreement on a set of paths is decidable from blob identifiers alone.

Definition 4.7 (Independence)**.** Two changes i, j with read sets $\mathsf{R}_i, \mathsf{R}_j$ and write sets $\mathsf{W}_i, \mathsf{W}_j$ are *independent* when

$$\mathsf{W}_i \not\bowtie (\mathsf{R}_j \cup \mathsf{W}_j) \quad \text{and} \quad \mathsf{W}_j \not\bowtie (\mathsf{R}_i \cup \mathsf{W}_i). \tag{1}$$

Condition (1) is Eq. 1 of the companion paper [24], with set intersection replaced by overlap so that a directory prefix and a file inside it count as meeting. The companion paper calls it a conservative candidate-independence condition that is only as good as the declared sets [24]. Section 6.4 makes precise what it guarantees at landing.

4.3 The transaction log

Definition 4.8 (Log entry)**.** A *log entry* is a tuple $e = (v, b, k, r, o, n, \mathsf{W}, \mathsf{R}, a, t, h)$ with version $v \geq 1$, base version b with $0 \leq b < v$, kind $k \in \{\text{commit}, \text{merge}, \text{revert}, \text{import}, \text{ref_move}, \text{ref_delete}\}$, fully qualified ref name r, old and new values $o, n \in \mathit{Id} \cup \{\perp\}$, touched paths W in which no path appears twice, read set R, actor a, commit time t and chain digest h. Only ref_delete has $n = \perp$.

Definition 4.9 (Hash chain)**.** Let h_0 be the empty string and let $\langle e \rangle$ be the canonical encoding of e without its digest. The chain digest of entry e_v is $h_v = G(h_{v-1}, \langle e_v \rangle)$, where G is SHA-256 over a canonical encoding of the pair.

Definition 4.10 (Log and well-formedness)**.** A *log* is a finite sequence $\mathcal{L} = (e_1, \ldots, e_n)$. It is *well formed* when (I1) e_v has version v for every v; (I2) every h_v equals $G(h_{v-1}, \langle e_v \rangle)$; and (I3) for every v, the old value o_v equals the value of ref r_v after replaying $e_1, \ldots, e_{v-1}$. The *current version* of $\mathcal{L}$ is n.

The labels I1 to I3 match the invariant list in Table 3. In the implementation, the log is a SQLite table keyed by version. The touched paths are copied into each entry from the commit's recorded paths at the moment of appending, so an entry is self-contained: validation never needs to consult commit metadata.

4.4 Snapshots, replay and checkpoints

Definition 4.11 (Snapshot and replay)**.** A *snapshot* is a finite map $S : \mathit{Ref} \rightharpoonup \mathit{Id}$. Replay applies an entry to a snapshot: $\mathrm{apply}(S, e)$ removes r when $n = \perp$ and otherwise sets r to n. For a well-formed log, $S_0 = \varnothing$ and $S_v = \mathrm{apply}(S_{v-1}, e_v)$. We call S_v the snapshot at version v.

Definition 4.12 (Checkpoint)**.** A *checkpoint* is a triple $K = (u, S, h)$ claiming that the snapshot at version u is S and that the chain digest at u is h. The implementation writes one in the same transaction as every entry whose version is a multiple of 64, stores it in the authority's database, and accepts it for a read only when h equals the digest stored in the log at version u.

The replay function of the implementation takes a starting checkpoint and a tail of entries. For each entry it checks that the version is the next one, that the entry is internally valid, that the recomputed chain digest matches, and that the entry's old value equals the ref's current value in the state being built; any failure aborts the read with an integrity error. This is the executable form of well-formedness.

Definition 4.13 (Time travel)**.** A *read at version* u, for $0 \le u \le n$, returns S_u computed by replay from the latest accepted checkpoint at or before u. A *read at commit* x returns objects reachable from x. A read without a version uses the ref table of the current state.

Time travel works by log version and by commit identifier. A read by timestamp is not implemented: the entries carry commit times, so a timestamp could be mapped to the last version committed before it, but the read path does not do so at the commit we describe.

4.5 The authority and its commands

Assumption 4.14 (Serialized authority)**.** The authority's command executions are equivalent to some sequential order of those executions. Each command runs inside one storage transaction that either commits all of its writes or none of them, and a command that returns an error commits none. No command performs network input or output inside its transaction.

Assumption 4.14 is the only property of the execution platform that the proofs use. Under it, the history of the authority's database is a sequence of committed command executions, and each command is a function from one database state to the next. The authority is then a state machine in Schneider's sense, whose deterministic commands each execute atomically with respect to the others [30]. For the log, the assumption gives the contract Delta Lake states for catalog-managed tables: a catalog "must not ratify version v until it has ratified version $v-1$, and it must ratify version v at most once" [15].

Many stores satisfy the assumption. Any store with serializable, all-or-nothing transactions over the authority's state does. A weaker store suffices for the log alone: if appending version $n+1$ is a put-if-absent and every command re-reads the log tail it validates against, Delta's argument applies [3].

The reference implementation satisfies it with a Durable Object, which processes one request at a time and whose synchronous transaction API is all-or-nothing [8, 9]. The authority's dispatcher wraps every command in that API and turns any error into a rollback, and the authority's crate calls only its SQL store interface (Appendix A).

Definition 4.15 (Proposal and validation)**.** A *proposal* to land commit c on ref r is $P = (r, b, \pi_1(c), \mathsf{W}(c), \mathsf{R})$, where b is the base version recorded with c. Against a log of current version n, let $\mathrm{Later}(P) = (e_v : b < v \le n,\ r_v = r)$. An entry e_v is *opaque* when its touched path list is empty and its kind is neither commit nor merge. *Validation* returns

1. $\mathrm{Conflict}(v)$ for the first $e_v \in \mathrm{Later}(P)$ that is opaque or satisfies $\mathsf{R} \bowtie \mathsf{W}^{\rho}_{v}$;
2. otherwise FastForward if $\mathrm{Later}(P)$ is empty;
3. otherwise Rebase if $\mathsf{W}(c) \not\bowtie \mathsf{W}_v$ for every $e_v \in \mathrm{Later}(P)$;
4. otherwise Merge with the overlapping paths.

The four outcomes are those of the function `Proposal::check` in the implementation. Two details are worth stating because they decide what the theorems can claim. The stale-read test uses the later entries' refined writes, so a read of one symbol is not invalidated by a later change to another symbol of the same file. The write-overlap test that separates Rebase from Merge works at path granularity only.

Definition 4.16 (Atomic append)**.** Appending commit c to ref r with kind k and old value o, in a state with current version n, writes in one transaction: the entry e_{n+1} with ref $r_{n+1} = r$, old value

$o_{n+1} = o$, new value $n_{n+1} = c$, touched paths $T(c)$, the commit's recorded read set and base (the base capped at n), the landing principal as actor, the authority's clock as time, and digest h_{n+1}; the new value of r in the ref table; the projection target for the object store; a checkpoint if 64 divides $n + 1$; and one row of the landings table recording that c landed on r at version $n + 1$.

When the appended commit is a platform merge landing one commit, the entry takes the merged commit's read set and base; when it lands a batch, it takes the union of the members' read sets and the smallest member base (Definition 7.1).

5 Properties of the log

5.1 Replay and checkpoints

The first half of the next theorem is the state-machine property: replicas that process the same sequence of requests reach the same state [30]. Delta Lake computes a snapshot the same way, by replaying committed actions in ascending version order from a checkpoint no newer than the requested version [15].

Theorem 5.1 (Replay determinism)**.** *Let $\mathcal{L}$ be well formed with current version n, and let $K = (u, S_u, h_u)$ be the true checkpoint at some $u \le v \le n$. Replaying $e_{u+1}, \ldots, e_v$ from K succeeds and returns (v, S_v, h_v). Replaying the same tail from any $K' = (u, S', h_u)$ with $S' \neq S_u$ either fails or returns a snapshot that differs from S_v only on refs that no entry of the tail touches.*

Proof. For the first claim, induct on $v - u$. The case $v = u$ is the checkpoint itself. For the step, the replay state before e_{w+1} is (w, S_w, h_w) by hypothesis. The version check passes by I1, the digest check passes by I2, and the old-value check passes by I3 because the state's value of r_{w+1} is $S_w(r_{w+1}) = o_{w+1}$. Applying the entry gives S_{w+1} and digest h_{w+1}. The induction is the state-machine argument made concrete: each step uses one of the implementation's three validity conditions.

For the second claim, suppose replay from K' succeeds. Each step sets or removes exactly the ref it names, and the digest checks do not depend on the snapshot component. Let $D = \{x : S'(x) \neq S_u(x)\}$. A ref in D that some tail entry touches is checked at its first touch: the old-value check compares $S'(x)$ with $o_w = S_{w-1}(x) = S_u(x)$, because no earlier tail entry touched x, and so fails. Since replay succeeded, no tail entry touches a ref in D, and the replayed states agree with S_w outside D and with S' on D. □

The second half of Theorem 5.1 is a limit, not a guarantee. The implementation accepts a checkpoint when its chain digest matches the log, and the digest binds the checkpoint to a position in the log but not to its ref map. A checkpoint whose ref map was altered while its digest was kept is detected only when the altered ref is touched again before the requested version. Delta Lake faces the same issue and specifies the remedy: a version checksum records metrics of the table state, and readers validate a reconstructed state by recomputing those metrics and comparing them with the checksum [15]. Storing a digest of the ref map with each checkpoint and in the next log entry would be the counterpart here. It is not implemented, and no test exercises a corrupted checkpoint with a correct digest.

Proposition 5.2 (Tamper evidence relative to a trusted digest)**.** *Let a reader hold a trusted digest h^*_w for version w. If any entry e_u with $u \le w$ is replaced by $e'_u \neq e_u$ while the stored log still validates up to w against h^*_w, then G has a collision.*

Proof. Let h'_v be the digests of the altered log. Validation to w means $h'_w = h^*_w = h_w$. Take the largest $v \le w$ at which the inputs to G differ, which exists because $\langle e'_u \rangle \neq \langle e_u \rangle$. Above v the inputs agree, so the outputs agree down to v, and $G(h'_{v-1}, \langle e'_v \rangle) = G(h_{v-1}, \langle e_v \rangle)$ with distinct inputs. □

The chain is tamper-evident only relative to a digest held outside the altered region. The implementation stores the digest both in the entry and in its own column, and checkpoints repeat it, so a careless edit of one entry is detected on the next replay; the test `tampered_history_is_detected_on_time_travel` edits an entry in place and observes an integrity error on a read at that version. An adversary with write access to the authority's database can recompute every digest and every checkpoint. Defending against that adversary would require publishing digests elsewhere, which the platform does not do. Crosby and Wallach make the general point: a hash-chained log is tamper-evident only against commitments that auditors hold, and an unaudited logger can present inconsistent histories [11]. The chain is therefore an integrity check against accidental or careless modification, not an audit mechanism.

5.2 Refs only advance

Lemma 5.3 (Append discipline)**.** *At commit `cfa4845`, every code path that appends an entry e_v satisfies $o_v = S_{v-1}(r_v)$, uses one of the kinds commit, merge, revert or import, and satisfies one of the following: (a) $S_{v-1}(r_v) = \bot$; or (b) $\pi_1(n_v) = S_{v-1}(r_v)$.*

Proof. The append function has seven call sites. The direct land with no current head appends a commit with $o = \bot$, which is case (a); import is allowed only when the repository has no refs, also case (a). The direct land of a commit whose first parent is the head appends with o equal to the head, case (b). The direct land of a platform merge first verifies that the merge's parents are exactly the head followed by the merged commit, case (b). The land combiner appends either a single commit after checking that its first parent is the head, or an octopus merge after checking that its parents are exactly the head followed by the landed commits, both case (b). The break-glass path refuses any commit whose first parent is not the current head, case (a) or (b). In every case o is the value read from the ref table in the same transaction, which is $S_{v-1}(r_v)$ by Assumption 4.14. No call site appends the kinds ref move or ref delete, although the log format allows them. □

Theorem 5.4 (Refs advance along first-parent chains)**.** *Let $u \le v$. If $S_u(r) = x \neq \bot$, then $S_v(r)$ is defined and x is reached from $S_v(r)$ by following first parents. If $S_u(r) = S_v(r)$, including the case where both are $\bot$, then no entry with version in $(u, v]$ names r.*

Proof. Induct on $v - u$. If e_v does not name r the value is unchanged. If it does, Lemma 5.3 gives $o_v = S_{v-1}(r) \neq \bot$, so case (b) applies and $\pi_1(S_v(r)) = S_{v-1}(r)$; no entry deletes a ref, so the value stays defined. Composing first-parent steps gives the first claim. For the second, every entry naming r sets it to a commit, so once named, r stays defined. If both values are $\bot$, no entry in $(u, v]$ can therefore name r. Otherwise $S_u(r) \neq \bot$, every entry naming r in $(u, v]$ adds one first-parent step, and each step strictly increases generation, so the generation of $S_v(r)$ exceeds that of $S_u(r)$ whenever at least one such entry exists. □

Corollary 5.5 (No lost update)**.** *If commit c landed on r at version u, then for every $v \ge u$, c is an ancestor of $S_v(r)$. In particular no landing overwrites another, and every landed commit stays reachable from the branch it landed on.*

Proof. Landing c at u means either $S_u(r) = c$, or $S_u(r)$ is a platform merge with c among its parents. Either way c is an ancestor of $S_u(r)$, which is a first-parent ancestor of $S_v(r)$ by Theorem 5.4. □

Corollary 5.5 says that the history of a ref exhibits no lost update, phenomenon P4 of Berenson and colleagues [4]. On the ref item, the compare-and-swap plays the role of their first-committer-wins rule: a writer whose commit was built on a stale head cannot replace the head, because the fast-forward path requires the first parent to equal the current value, and must go through validation and a merge instead. Files are handled differently (Section 6.3).

Refs are never rewritten. Undoing a landed change is itself a new commit: post-land repair proposes a revert commit, and break-glass lands only fast-forwards.

Theorem 5.4 is a property of the code at the stated commit, not of the log format, which reserves kinds for ref moves and deletions. If an owner-reset operation were added, the theorem would hold only between resets; validation already treats such entries as opaque.

5.3 Snapshot immutability

Theorem 5.6 (Snapshot immutability)**.** *Let $\mathcal{L}$ have current version n and let $\mathcal{L}'$ be the log at any later time, with current version $n' \geq n$. For every $u \leq n$, the snapshot at version u computed from $\mathcal{L}'$ equals the snapshot at version u computed from $\mathcal{L}$. For every ref r with $S_u(r) = x$, the tree $\tau(x)$ and every object reachable from it are the same at both times, if they are present.*

Proof. The authority only inserts log rows, keyed by version; no code path updates or deletes a log row, and an insert at an existing version fails. Hence $\mathcal{L}$ is a prefix of $\mathcal{L}'$. S_u depends only on $e_1, \ldots, e_u$ (Definition 4.11), and replay from any accepted true checkpoint returns the same value (Theorem 5.1). The second claim follows from content addressing (Definition 4.1): an identifier determines its bytes. □

A read pinned at version u therefore has snapshot semantics in the strongest sense: it is a pure function of u, it takes no lock, and it does not contend with landings beyond sharing the authority's single thread for the duration of one read command. In the terms of Bernstein, Hadzilacos and Goodman, a read pinned at u is a query of the multiversion mixed method with timestamp u: it reads the latest version older than its timestamp, written by a committed transaction, so no later write can invalidate it [6, Section 5.5]. Delta Lake makes the same observation about its snapshot reads, which create no contention because they only read immutable objects [3].

Theorem 5.6 says that objects are unchanged if they are present. Presence needs its own assumption, because the object store is a separate system that may collect unreachable objects.

Assumption 5.7 (Retention)**.** The object store keeps every object reachable from the current value of some ref.

Corollary 5.8 (Readable snapshots)**.** *Under Assumption 5.7, for every $u \leq n$ and every ref r that is defined in both S_u and S_n, every object reachable from $S_u(r)$ is present.*

Proof. By Theorem 5.4, $S_u(r)$ is a first-parent ancestor of $S_n(r)$, so everything reachable from it is reachable from the current value of r. □

Since no implemented operation deletes a ref (Lemma 5.3), every ref defined at u is defined at n, and Corollary 5.8 covers every pinned read. Assumption 5.7 is the only property the object store must provide. The platform's procedure for releasing hidden refs is designed to preserve it, and Part II gives that argument in its Proposition 12.1; nothing in this paper depends on that procedure beyond the assumption.

Table 2: Correspondence with the Delta Lake log. Delta's column cites its protocol, paper and implementation as noted in the text.

Concept	Delta Lake	This model
Unit of state	A set of data files per table	A map from refs to commits per repository
Log record	A set of actions for version v, never overwritten	One ref move with its touched paths, inserted at version v only
Atomic step	Put-if-absent, atomic rename, or a catalog that ratifies each version once and in order	One command of the serialized authority
Checkpoint	Written by any client, by default every 10 commits	Written by the authority in the same command as the entry, every 64 versions
State validation	A version checksum compared with the replayed state	A hash chain over entries, with no check of a checkpoint's ref map
Read set	Read predicates and files, recorded by the engine	Declared by the writer, at path or symbol granularity
Overlapping writes	A conflict, for example two deletions of one file	A three-way merge when the line merge is clean
Time travel	By version or by timestamp	By version or by commit
Retention	Removed files kept for a period, by default 7 days	Objects kept while reachable from a ref

5.4 Relation to the Delta log

The structure matches Delta Lake's closely, and Table 2 lists the correspondence. Delta names log records by contiguous version numbers, requires that writers never overwrite an existing log entry, and reconstructs any version by replaying actions from a checkpoint [15, 3]. Its write protocol reads at version r and then attempts to create record $r + 1$ atomically, and its paper concludes that writes are serializable in increasing order of record identifiers because only one transaction can create each record [3]. That argument covers the ordering of commits. Whether a transaction that retries at a later version is still correct depends on conflict checks, which the paper does not specify and the implementation does: a committing transaction is checked against each concurrently committed one for added files that match its read predicates, deleted files it read, and files both deleted [12, 13].

Validation in this model is closest to Delta's Serializable isolation level, under which reads and writes behave as if executed one by one, with read predicates supplied by the writer instead of recorded by the engine [14]. It is not Delta's default WriteSerializable level, which orders only the writes and lets an update commit past a concurrent insert it should have read, as if the update had run first [14]. The authority never reorders: the serial order is always the log order, and a landing whose declared reads meet a later write is refused. It also departs from all three of Delta's levels, including the internal SnapshotIsolation level, on overlapping writes, which Delta fails and this model merges.

The atomic step is cheaper here than in Delta. It is a command of the serialized authority, not a conditional write to an object store; in the reference implementation that is one round trip to a Durable Object rather than a storage write with tens to hundreds of milliseconds of latency. Delta's paper attributes its limit of several write transactions per second to that latency and suggests a low-latency coordinator as the remedy [3].

The log is also smaller. It records only ref changes, so a checkpoint is a map of refs, and commits that have not landed are durable but outside the log; that is what lets commit creation proceed without touching it.

Like Delta, which supports transactions within one table only [3], the model gives no atomicity

across refs or repositories.

6 Concurrency control

6.1 Validation against committed entries

Landing uses optimistic concurrency. A change is built against a base version without any lock. At landing, the authority compares it with the entries that were appended to the same ref after its base, which is validation against transactions that have already committed; the literature calls this backward validation (see [20]). The comparison and the append happen in one command, so no entry can slip between them.

Assumption 6.1 (Consistent base)**.** A commit c landed on r has a base b with $\pi_1(c) = S_b(r)$, and the writer's reads were taken from $\tau(\pi_1(c))$.

Assumption 6.2 (Sound refinement)**.** For every entry e_v, every location whose content differs between $\tau(S_{v-1}(r_v))$ and $\tau(S_v(r_v))$ lies in $\mathrm{sub}(w)$ for some $w \in \mathsf{W}^{\rho}_{v}$.

The adapter satisfies Assumption 6.1 when a writer names a branch: it reads the branch's snapshot once and passes both the head and the version. A writer may instead pass a parent and a base version of its own choosing, and the authority checks only that the base is not newer than the log. A base that is older than the parent is harmless, because validation then checks more entries than it needs to. A base that is newer than the parent's version would let validation skip entries that the writer never saw. Requiring $\pi_1(c) = S_b(r_c)$ at landing, or deriving b from the version at which the parent landed, would close the gap; neither is implemented.

One consequence one might fear does not occur. A writer that claims the current version as its base while naming an old parent makes $\mathrm{Later}(P)$ empty, so validation returns FastForward. Both merge paths refuse that outcome: single-merge verification accepts only Rebase or Merge, and batch verification accepts FastForward only when the commit's first parent is the head (Appendix A). The exposure is limited to a base that lies strictly between the parent's version and the head, which makes validation skip the entries in between.

Assumption 6.2 holds at path granularity for every entry whose commit the authority prepared, including platform merge commits, by Lemma 5.3 and the inclusion $\Delta(c) \subseteq T(c)$: the first parent of every new value is the old value, so the content of a ref changes exactly on the change of the new commit. Import entries are the exception, because imported history is recorded by the import code rather than prepared; they occur only when a repository is empty. At symbol granularity it is a claim about the parser that derives symbol touches, which treats an edit outside any symbol, and any file it cannot parse, as a whole-file write.

Lemma 6.3 (Frame)**.** *Under Assumption 6.2, for any set of accesses X and any versions $b \le v$, if $X \not\bowtie \mathsf{W}^{\rho}_{u}$ for every entry e_u with $b < u \le v$ and $r_u = r$, then $\tau(S_b(r)) \equiv_X \tau(S_v(r))$.*

Proof. Entries on other refs do not change the value of r. For an entry e_u on r, a location in $\mathrm{sub}(x)$ for $x \in X$ whose content changed would lie in $\mathrm{sub}(w)$ for some $w \in \mathsf{W}^{\rho}_{u}$, so $\mathrm{sub}(x) \cap \mathrm{sub}(w) \neq \varnothing$ and $x \bowtie w$ by Lemma 4.5, contrary to hypothesis. Agreement is transitive, so it holds across all of $(b, v]$, and it holds trivially when $b = v$. □

Theorem 6.4 (Serializability relative to declared reads)**.** *Fix a version v_0 and let $e_{v_1}, \ldots, e_{v_m}$, with $v_0 < v_1 < \cdots < v_m$, be all the entries naming r with versions above v_0. Suppose each lands a single commit c_i with one parent, read set R_i and base $b_i \ge v_0$, either directly or through a platform*

merge commit whose parents are $S_{v_i-1}(r)$ and c_i, and not as part of a batch. Write $m_i = n_{v_i}$ for the commit the entry installs, so that $m_i = c_i$ on a fast-forward. Under Assumptions 4.14, 6.1 and 6.2, for every i:

1. *the values c_i read are the values current at its serialization point: $\tau(S_{b_i}(r)) \equiv_{\mathsf{R}_i} \tau(S_{v_i-1}(r))$;*
2. *if validation returned FastForward, $S_{v_i}(r) = c_i$ and $S_{v_i-1}(r) = \pi_1(c_i)$;*
3. *if validation returned Rebase and the platform merge is the three-way merge of $\pi_1(c_i)$, $S_{v_i-1}(r)$ and c_i, then $S_{v_i}(r) = m_i$, the entry's touched paths satisfy $\mathsf{W}_{v_i} = T(m_i) \subseteq \mathsf{W}(c_i)$, and $\tau(m_i)$ equals $\tau(c_i)$ on $\mathsf{W}(c_i)$ and equals $\tau(S_{v_i-1}(r))$ elsewhere.*

Part 1 holds for every landing, including those validated as Merge. If every landing in the sequence was validated as FastForward or Rebase, the history of r is equivalent to the serial execution of $c_1, \ldots, c_m$ in log order, in which each change reads its declared set from the state just before its own entry. Landings validated as Merge are covered by Proposition 6.5.

Proof. The landing that appended e_{v_i} either took the fast-forward path or ran validation, and in both cases it did so in the same command as the append, against the log of current version $v_i - 1$.

Consider the fast-forward path first. The direct land and the combiner take it whenever $\pi_1(c_i)$ equals the head, without running validation. By Assumption 6.1, $S_{b_i}(r) = \pi_1(c_i) = S_{v_i-1}(r)$, and by Theorem 5.4 equal values at b_i and $v_i - 1$ imply that no entry on r lies between them. So $\mathrm{Later}(P_i)$ is empty, validation would have returned FastForward, part 1 holds trivially, and the append sets r to c_i, which is part 2.

Otherwise validation ran and did not return Conflict, so no later entry was opaque and $\mathsf{R}_i \not\bowtie \mathsf{W}^{\rho}_u$ for every $e_u \in \mathrm{Later}(P_i)$. Lemma 6.3 with $X = \mathsf{R}_i$ gives part 1.

For part 3, write $B = \tau(\pi_1(c_i))$, $H = \tau(S_{v_i-1}(r))$ and $C = \tau(c_i)$. Rebase means $\mathsf{W}(c_i) \not\bowtie \mathsf{W}_u$ for all later entries on r. Lemma 6.3, applied with the unrefined write sets, for which Assumption 6.2 holds by Lemma 5.3, shows that B and H agree on $\mathsf{W}(c_i)$. Outside $\mathsf{W}(c_i)$, B and C agree because $\Delta(c_i) \subseteq \mathsf{W}(c_i)$. A three-way merge takes, at each path, the side that differs from the base. On $\mathsf{W}(c_i)$ only C can differ from B, so the result is C; elsewhere only H can differ, so the result is H. The authority prepared m_i with first parent $S_{v_i-1}(r)$ and verified that it touches only paths changed by commits reachable from c_i but not from the head, that is, $T(m_i)$ is contained in those paths (Appendix A, row 18); since c_i has the single parent $S_{b_i}(r)$, an ancestor of the head by Theorem 5.4, those paths are $\mathsf{W}(c_i)$.

For the equivalence statement we reduce to the serializability theorem of Bernstein, Hadzilacos and Goodman: a history is serializable if and only if its serialization graph is acyclic [6, Theorem 2.1]. Treat the state at v_0 as initial and model each landing c_i as a transaction that reads every location of R_i from the snapshot at b_i and writes the locations of its entry's refined write set $\mathsf{W}^{\rho}_{v_i}$ at v_i, which by Assumption 6.2 cover every location it changed, and let two operations conflict when they touch overlapping locations and one is a write. The graph has an edge $c_i \to c_j$ when an operation of c_i precedes a conflicting operation of c_j. Writes happen at append time, in log order, so every write-write edge points forward. A read by c_j of a location written by c_i sees that write only if $v_i \le b_j < v_j$, so every write-read edge points forward. A read-write edge $c_j \to c_i$ with $v_i < v_j$ would mean that c_j read a location at $b_j < v_i$ that c_i later wrote, so that $e_{v_i} \in \mathrm{Later}(P_j)$ and $\mathsf{R}_j \bowtie \mathsf{W}^{\rho}_{v_i}$. Validation refuses exactly that landing, and on the fast-forward path $\mathrm{Later}(P_j)$ is empty. Every edge therefore points forward in log order, the graph is acyclic, and its topological order is the log order. In a history with no Merge outcome, each write of c_i installs the value c_i computed (parts 2 and 3), so the read-write model is exact and the history is conflict-serializable in log order. □

Theorem 6.4 guarantees conflict-serializability on one ref for histories of FastForward and Rebase landings, relative to declared read sets. Classical serializability theory [6] and Delta's conflict checker [12] track the data a transaction actually reads; the authority validates only the reads the writer declares, and it ignores entries on other refs, which write other data. When writes overlap, landing falls back from Rebase to a line-level Merge, which Proposition 6.5 describes and which the theorem does not cover.

The theorem also assumes that the platform's merge is the three-way merge. The authority does not recompute it. It checks that the merge commit names the expected parents, was built by the `platform:merge` principal, and changes only paths in the merged work's write set. That the merged tree is the three-way merge rests on the stateless merge code, which the differential test of Section 10 checks against Git.

6.2 Correspondence with Kung and Robinson

Kung and Robinson assign each transaction a number at the end of its read phase and accept transaction T_j if, for every T_i with a smaller number, one of three conditions holds: T_i finished writing before T_j started reading; or the write set of T_i does not intersect the read set of T_j and T_i finished writing before T_j started writing; or the write set of T_i meets neither the read set nor the write set of T_j and T_i finished its read phase first [23]. They show that the conditions guarantee a serially equivalent schedule in transaction-number order [23].

Their serial validation implements the first two conditions: a transaction records the current transaction number when it starts, and at its end checks, inside one critical section, the write set of every transaction numbered after its start against its own read set, writing and taking the next number only if no intersection is found [23, Section 4]. A landing is that end step with start number b and finish number n, and Assumption 4.14 supplies the critical section. The mapping is direct. The log version is the transaction number, assigned at append time, which is the end of the read phase. Entries at or below the base satisfy the first condition: they were in the snapshot the writer read. Entries above the base are validated by the second condition: their writes must not meet the change's reads, and they finished writing before the change writes, because appends are serial. Entries on other refs write other data and never meet a read of r. The third condition, which allows overlapping write phases, is never needed, because the write phase is one transaction.

The departure is in write-write overlap. Under the second condition, a later transaction may overwrite a file an earlier one wrote, provided it did not read it. Taken literally for source code, that rule would let a change replace a whole file and silently discard a concurrent edit to other lines. The platform instead treats overlapping writes as a Merge and combines the two edits line by line. In effect it treats each change's write as a patch, a set of edited regions relative to its base, rather than as a new value of the file.

Proposition 6.5 (Merged writes)**.** *Suppose validation of c against head $S_n(r)$ returns Merge with overlapping paths M, and the platform's three-way merge succeeds. Then the landed tree equals $\tau(c)$ on $\mathsf{W}(c) \setminus M$, equals the head on the paths outside $\mathsf{W}(c)$, and on each $p \in M$ equals the line-level three-way merge of the base, head and change versions of p.*

Proof. The argument of part 3 of Theorem 6.4 applies verbatim to each path outside M. On $p \in M$, the platform's tree merge calls its line merge with the three versions, and the result is whatever that merge returns. □

Proposition 6.5 is a description, not a serializability claim. In serializability theory, operations other than reads and writes may be treated as non-conflicting only if their combined effect does not

depend on the order in which they run [6, Section 2.5]. Two merged edits to one file would qualify only if a clean merge always equaled applying the edits in either order. No such theorem holds for `diff3`: Khanna, Kunal and Pierce show that the most obvious formulations of the intuition that edits to well-separated regions never conflict are wrong, and that idempotence and stability fail in general [22]. The platform's merge is tested against Git's own, not proved: on 300 randomly generated three-way merges, it agreed with `git merge-file` using the histogram diff on every case about whether the merge was clean, and on the result whenever it was. A clean textual merge can still be semantically wrong. The platform therefore runs the policy's checks on every merged result after it lands and proposes a revert commit when a check fails.

6.3 Isolation levels

Reads are snapshot reads: a read at version u sees S_u and nothing later, by Theorem 5.6. Landings validated as FastForward or Rebase are serializable relative to declared reads, by Theorem 6.4, and merged writes satisfy Proposition 6.5. Overlapping writes are reconciled by a three-way merge instead of first-committer-wins.

The system therefore does not provide snapshot isolation at the granularity of files. Snapshot isolation lets a transaction commit only if no concurrent transaction that committed first wrote data it also wrote, the first-committer-wins rule [4]; in Adya's formulation, concurrent transactions may not modify the same objects [1, Section 4.3]. The Merge outcome lets two concurrent writers of one path both commit. A read that a writer does not declare is a snapshot read and carries no further guarantee. Its typical failure is write skew, phenomenon A5B of Berenson and colleagues, in which two transactions read overlapping data and write disjoint parts of it [4].

Example 6.6 (Write skew from undeclared reads)**.** Files `a.rs` and `b.rs` each define half of a protocol that must agree. Change c_1 reads both files and edits `a.rs`; change c_2 reads both and edits `b.rs`; both are based on the same version and neither declares a read set. Whichever lands second sees a later entry that wrote a file it did not write, so validation returns Rebase and both land. The result contains both edits, and neither change ever saw the other. This is the A5B pattern. If both declare $\mathsf{R} = \{\texttt{a.rs}, \texttt{b.rs}\}$, the second to land has $\mathsf{R} \bowtie \mathsf{W}$ of the first and gets a Conflict.

The example shows why declared reads carry the guarantee. The implementation keeps reads conservative in two ways. A read that cannot be resolved to symbols stays a whole-file read, and the plan builder of Part II (Section 10) adds the symbols that a read symbol references, to a chosen depth, before checking independence.

6.4 Independence as a scheduling rule

Condition (1) is two-sided and is evaluated before work runs; validation is one-sided and is evaluated at landing, against what actually landed. The following theorem states the exact relation at path granularity.

Theorem 6.7 (Independence is order-insensitive landing without merge)**.** *Let c_i and c_j be commits on the same base b of ref r, with read and write sets at path granularity, and suppose no other entry on r is appended between b and their landings. The following are equivalent.*

1. *c_i and c_j satisfy* (1)*.*
2. *In both landing orders, the second change to land is validated as Rebase.*

Proof. If c_i lands first, Later of c_j is exactly the entry that landed c_i, which writes W_i and is not opaque. Validation of c_j returns Conflict if $\mathsf{R}_j \bowtie \mathsf{W}_i$, otherwise Merge if $\mathsf{W}_j \bowtie \mathsf{W}_i$, otherwise Rebase. So c_j is a Rebase exactly when $\mathsf{W}_i \not\bowtie \mathsf{R}_j$ and $\mathsf{W}_i \not\bowtie \mathsf{W}_j$. Symmetrically, with c_j first, c_i is a Rebase exactly when $\mathsf{W}_j \not\bowtie \mathsf{R}_i$ and $\mathsf{W}_j \not\bowtie \mathsf{W}_i$. The conjunction of the two is (1), because $\bowtie$ is symmetric. □

Corollary 6.8 (Symbol granularity)**.** *If c_i and c_j satisfy* (1) *with refined write sets and symbol-level read sets, then in either order the second change is not a Conflict. It is a Rebase if their path-level write sets are disjoint and a Merge otherwise.*

Proof. The stale-read test uses refined writes, so the argument of Theorem 6.7 gives no Conflict in either order. The Rebase and Merge tests use path-level writes, which may overlap when two changes edit different symbols of one file. □

In the terms of serializability theory, condition (1) says that the two changes have no conflicting operations, and for such a pair both serial orders are equivalent [6, Sections 2.1 and 2.2]; it is the condition Bernstein gave for running two program segments in parallel (see [5]). Theorem 6.7 shows that the validator honours this exactly: its outcome is independent of landing order precisely when the pair has no conflict. The theorem carries over to the code because the scheduler's independence test and the landing validation use the same path comparison.

Example 6.9 (Validation accepts what independence rejects)**.** Let c_i read `config.toml` and write `src/a.rs`, and let c_j write `config.toml`. Condition (1) fails because $\mathsf{W}_j \bowtie \mathsf{R}_i$. If c_i lands first, c_j is validated against an entry that wrote `src/a.rs`, which it neither read nor wrote, and it lands as a Rebase. The result is serializable in the order c_i, c_j. If c_j lands first, c_i has a stale read and is refused.

Validation is a safety rule: it refuses every landing whose declared reads are stale at its position in the log. Independence is a scheduling rule: a scheduler that starts two items at once does not know which will land first, and Theorem 6.7 says that (1) is exactly the condition under which the outcome does not depend on that race. A scheduler that ignores (1) remains safe, because validation still holds, but it wastes work on whichever item loses. This division is also the one the companion paper draws when it says that integration should reject stale assumptions even for items believed independent [24].

Proposition 6.10 (Coarsening is sound)**.** *Replace any access a in a read or write set by an access a' with $a' \sqsubseteq a$, for instance a symbol by its file or a file by its directory. Then every Conflict and every Merge outcome of the original sets is still a Conflict or a Merge, every pair that satisfies* (1) *after coarsening satisfied it before, and Assumption 6.2 is preserved.*

Proof. By Lemma 4.5, $a \bowtie x$ implies $a' \bowtie x$, so every overlap between the original sets persists, and the coarse sets meet whenever the fine ones do. Each outcome and condition (1) are monotone in overlap in the stated direction. For Assumption 6.2, $\mathrm{sub}(a) \subseteq \mathrm{sub}(a')$, so any location covered by the fine write set is covered by the coarse one. □

Proposition 6.10 is the validation counterpart of multigranularity locking, in which a lock on a granule implicitly covers all of its descendants and coarser granules trade concurrency for lower overhead [6, Section 3.9]. It justifies the rule that anything not resolved to symbols stays a whole path: coarsening can only cost concurrency, never correctness. The tests `symbol_reads_conflict_only_with_writes_to_that_symbol` and `unresolved_writes_stay_whole_files` exercise the two directions in the implementation.

7 The land combiner

7.1 Mechanism

Validation makes a lost race cheap, but it does not make it free. When a hundred writers land on one branch at once, each landing after the first finds that the branch moved and needs a merge commit, and each merge commit is stale as soon as another landing succeeds. Without coordination the writers would rebuild merges against a moving head. The land combiner replaces that race by one merge per batch.

Definition 7.1 (Land combiner)**.** For each branch r the authority keeps a queue of *tickets*, each naming a commit and a status of pending, landed or failed, and at most one *lease* $(r, \epsilon, t_{\text{exp}})$ with an epoch ϵ and an expiry time. The combiner has four commands.

Enqueue adds a pending ticket for a commit whose gates are satisfied, or returns the existing pending ticket for the same commit and branch.

Claim returns a wait hint if an unexpired lease exists. Otherwise it takes up to 64 pending tickets in ticket order, sets the lease to epoch $\epsilon + 1$ with expiry 30 seconds ahead, fails at once every ticket whose validation is a Conflict or whose commit is stale, and returns the rest together with the head and the trees each merge needs.

Land batch takes an epoch, a list of tickets to land, a list of failures and optionally a merge commit. It refuses unless the lease's epoch equals the given epoch. It records the failures. Without a merge it lands a single ticket whose commit has the head as first parent. With a merge it verifies that the merge's parents are exactly the head followed by the landed commits, that the merge was built by the platform's merge principal, that every landed commit still validates against the log, and that the merge changes only paths that some landed commit changed. It then appends one merge entry whose touched paths are those of the merge commit relative to the head, whose read set is the union of the members' read sets and whose base is the smallest member base, writes one landings row per member, marks the tickets landed and deletes the lease.

Release deletes the lease of a given epoch.

Only the platform's merge principal may claim. The adapter's land route first tries a direct landing. If the branch has queued work or the commit needs a merge, the direct landing is refused, the commit is enqueued, and the route alternates between polling its ticket and trying to claim the queue itself; whichever request holds the lease builds the merge for the whole batch. While building, the adapter merges each claimed item against the head in ticket order, fails items whose three-way merge conflicts, defers to the next batch any item whose merged paths overlap those of an earlier accepted item, and builds one octopus merge commit of the head and every accepted item. A batch that contains a fast-forward item lands that item alone.

The combiner is group commit applied to refs. DeWitt and colleagues describe committing all transactions whose commit records share a log page with a single log write, and call the idea database folklore [16]. Here the shared resource is the head of a branch rather than a log device, and the unit amortized is the construction of a merge against that head. Mononoke's pushrebase solves the same problem differently: the server rebases pushed commits onto the moving bookmark when no file conflicts, treating two commits that modify the same file as conflicting even when they change different lines, and gives the rebased commits new hashes [28]. The combiner keeps the landed commits' identifiers, records the batch as a merge, and merges same-file edits at line level.

7.2 Safety

Theorem 7.2 (Combiner safety)**.** *Under Assumption 4.14:*

1. *at most one entry is appended per version, and every entry appended by the combiner satisfies Lemma 5.3;*
2. *a land-batch command whose epoch is not the lease's current epoch changes no state;*
3. *each ticket is landed at most once, and each landed commit has exactly one landings row on its branch;*
4. *each member c of a landed batch at version v satisfies $\tau(S_{b_c}(r)) \equiv_{\mathsf{R}_c} \tau(S_{v-1}(r))$, where b_c is its base, provided Assumptions 6.1 and 6.2 hold;*
5. *properties 1 to 4 hold whatever the lease state, including after a lease expires while its holder is still building a merge.*

Proof. Properties 1 to 3 say that the authority meets Delta's ratification contract for the batch path: each version is ratified once, after its predecessor [15]. Property 1: the log's version column is a primary key and commands are serial, so two appends cannot share a version. The combiner's two append sites are covered in the proof of Lemma 5.3. Property 2: the epoch check is the first state-dependent step, and its failure returns an error, which rolls the command back. Property 3: landing requires each ticket to be pending, marks it landed in the same transaction, and the landings table has a primary key on commit and ref. Property 4 is the serial validation of Kung and Robinson applied to each member [23, Section 4]: verification re-runs validation for each member against the log of current version $v-1$ inside the appending command and accepts only Rebase, Merge, or a fast-forward on the current head. In the first two cases no entry in Later of c is opaque or meets R_c. In the third, $\pi_1(c) = S_{v-1}(r)$ and $\pi_1(c) = S_{b_c}(r)$ by Assumption 6.1, so Later of c is empty by Theorem 5.4. Lemma 6.3 gives the agreement in all three cases. Property 5: none of the arguments for 1 to 4 uses the lease. The head is re-read and the parents re-checked in the appending transaction, so a merge built on a head that has since moved fails with a stale-parent error and rolls back, whoever holds the lease. □

The lease is therefore a performance device. It stops several requests from building merges against the same head at once, and it orders merge construction, but correctness rests on the checks inside the appending command. This is the right division for a platform where any adapter request can die mid-flight: a lease held by a dead request expires after 30 seconds, and its holder, if it wakes up, can do no harm.

7.3 Write skew inside a batch

Property 4 of Theorem 7.2 validates each member against the log. It does not validate members against each other, and the adapter's deferral rule compares only merged write paths. That leaves an anomaly.

Theorem 7.3 (Batch write skew)**.** *At commit `cfa4845`, there are histories in which the combiner lands two commits c_a and c_b in one entry such that $\mathsf{R}_a \bowtie \mathsf{W}_b$ and $\mathsf{R}_b \bowtie \mathsf{W}_a$. For such a pair no serial order of c_a and c_b lets each read its declared set from the state the other leaves.*

Proof. Let the head be h at version b. Let c_a and c_b have first parent h and base b, with $\mathsf{R}_a = \{\mathtt{x}\}$, $\mathsf{W}_a = \{\mathtt{y}\}$, $\mathsf{R}_b = \{\mathtt{y}\}$ and $\mathsf{W}_b = \{\mathtt{x}\}$. Let a third commit c_0 that writes only `z` land as a fast-forward

at version $b+1$, so neither c_a nor c_b is a fast-forward any more. Both direct landings are refused with a merge-required error and both are enqueued. At the next claim, validation of each against Later $= (e_{b+1})$ finds that `z` meets neither read set and neither write set, so both are Rebase items. The adapter merges each against the head; the merged paths are `y` and `x`, which do not overlap, so both are accepted into one octopus merge. Land batch re-validates each against the same log and finds no conflict, checks the parents and paths, and appends the merge. In the serial order c_a, c_b, the change c_b would read `y` after c_a wrote it, which differs from what it read; in the order c_b, c_a, the change c_a would read `x` after c_b wrote it. □

The anomaly is a known one. The two members form a cycle $c_a \to c_b \to c_a$ of read-write anti-dependency edges in Adya's direct serialization graph, which is phenomenon G2-item [1, Section 3.2.4], and the history has the shape of Berenson's write skew A5B [4]. Fekete and colleagues prove that every non-serializable history under snapshot isolation contains two consecutive anti-dependency edges between concurrent transactions, possibly returning to the same transaction [17, Theorem 2.1]; the batch pair is the smallest instance. Validating each member against the committed log but not against the other members is certification in the style of snapshot isolation, and this is its expected failure. We do not call the combiner snapshot-isolated, because the check that keeps members' writes disjoint runs in the adapter and the authority does not repeat it.

The finding has since changed the code. Revision `c4c8f47` added the check of Proposition 7.4 to the combiner and to the authority's batch verification, refusing a member that reads or writes what an earlier member wrote with a `WRITE_SKEW` error. The batch landing path merged into the development branch at `fef94e7` calls the same check for every member before routing it, and defers a member that fails it to the next batch (Section 8). The check has not been exercised: the hosted benchmarks write disjoint files, so they could not show write skew with or without it. At `cfa4845` and `03821e0`, the combiner has no such check.

The same pair landed one at a time would not both succeed: whichever landed second would see the other's entry in Later and fail with a stale read. The anomaly is specific to batching, and it arises exactly when the branch has moved, which is when the combiner is in use. It is a gap between the implementation and Theorem 6.4, not a property of the log design, and the following rule closes it.

Proposition 7.4 (Serializable batches)**.** *Suppose the authority's batch verification also requires, for the accepted members $c_1, \ldots, c_k$ in ticket order, that $(\mathsf{R}_j \cup \mathsf{W}(c_j)) \not\bowtie \mathsf{W}(c_i)$ for all $i < j$, using the recorded write sets, and that the merge of each member is the three-way merge against the head. Then:*

1. *the batch entry produces the same branch state as landing $c_1, \ldots, c_k$ one at a time in ticket order, and each of those sequential landings would have been validated without a Conflict;*
2. *if, in addition, every member's validation against the log returned Rebase or FastForward, the history consisting of the preceding entries and the batch, with members ordered by ticket, is conflict-serializable.*

Proof. Let H_0 be the head tree and let H_j be H_{j-1} with the merged changes of c_j applied. The merged changes of c_j are confined to $\mathsf{W}(c_j)$, because a three-way merge differs from the head only where the member differs from its base, and $\Delta(c_j) \subseteq \mathsf{W}(c_j)$. The recorded write sets are pairwise disjoint by the new rule, so the merged changes of distinct members touch disjoint paths and H_k is the tree of the octopus merge. Consider member j. Its reads agree between its base and H_0 by Theorem 7.2. Between H_0 and H_{j-1} only paths in $\bigcup_{i<j} \mathsf{W}(c_i)$ change, and the rule keeps these disjoint from R_j, so its reads agree with H_{j-1} as well. Sequential validation of c_j after $c_1, \ldots, c_{j-1}$

would find no stale read for the same reason, and no new write overlap because $\mathsf{W}(c_j) \not\bowtie \mathsf{W}(c_i)$. Finally, H_0 and H_{j-1} agree on $\mathsf{W}(c_j)$, so the merge of c_j against H_{j-1} equals its merge against H_0 on $\mathsf{W}(c_j)$ and keeps H_{j-1} elsewhere. This is the state that sequential landing in ticket order produces, which proves part 1.

For part 2, build the serialization graph as in the proof of Theorem 6.4, with the members placed in ticket order after the entries that precede the batch. Edges between earlier entries and members point forward by that proof, because each member was validated against the log. Between members, writes are pairwise disjoint, so there are no write-write edges, and the rule forbids a read by c_j of a location written by an earlier member c_i, so every read-write edge between members points forward in ticket order. The graph is acyclic, and Theorem 2.1 of Bernstein, Hadzilacos and Goodman gives conflict-serializability [6]. Without the extra hypothesis of part 2, a member validated as Merge carries the weaker guarantee of Proposition 6.5. □

The rule is sufficient, not necessary, in two ways. It compares reads with the unrefined write sets $\mathsf{W}(c_i)$, whereas sequential validation compares them with the refined sets W^ρ, so with symbol-level reads the rule refuses some members that sequential landing would accept; using $\mathsf{W}^\rho(c_i)$ in the rule would remove that difference. More generally, a certifier that builds the batch's serialization graph and accepts whenever the graph is acyclic, in the manner of serialization-graph-testing certifiers [6, Section 4.4], accepts strictly more batches: those that are serializable in an order other than ticket order. Because members' writes are pairwise disjoint, every topological order yields the same tree, so the batch entry would not depend on which order the certifier found. Serializable snapshot isolation is a further alternative (see [7]).

The rule costs one overlap test per pair of accepted members, at most $64 \cdot 63/2$ tests per batch. It belongs in the authority's verification, because the authority is the component that must not trust its callers: today the only write-disjointness check inside a batch is the adapter's comparison of merged paths, which the authority does not repeat. Members that fail the rule need not fail; deferring them to the next batch preserves the stale-read semantics of sequential landing.

7.4 Liveness and post-land checks

The combiner trades a liveness assumption for its efficiency. While a branch has pending tickets, direct landings on it are refused and join the queue, so the branch advances only if some request claims. In the implementation every waiting land request tries to claim, so progress needs only one live request per branch; a request that gives up after its deadline leaves its ticket pending for others to land. A lease abandoned by a crashed request blocks claims for at most 30 seconds. We have not proved a bound on how long a ticket can wait, and the hosted measurements of Section 10 show that waiting for claims dominates landing latency under load.

Definition 7.5 (Repair commit)**.** When a required check fails on a commit that has already landed, the platform records, for each branch the commit landed on, a *repair*: a proposed commit on that branch that reverts the failing change, or the reason no such commit could be built. The repair is a new commit that lands through the ordinary path.

Post-land checks exist because validation sees only declared sets, and merged changes have never been tested together. The companion paper's warning that independent textual patches still require combined acceptance [24] is implemented as a check stage after every merge or landing and a repair proposal on failure. Repairs and break-glass actions add commits; neither rewrites a ref, which is Corollary 5.5 applied to the implementation.

8 Changeset landing

This section states the changeset landing path of Section 3.6 in the model and checks each earlier result against it. The statements are about the design; the paragraph after the proof records what the implementation at `dd6c9a8` does.

Definition 8.1 (Changeset landing)**.** A *changeset* on ref r is $\chi = (r, b, \mathsf{R}, \mathsf{W}, \Phi)$: a base version b, a declared read set R, a set W of file paths, and a map Φ giving for each path in W its new content or its deletion. The *branch state* $\sigma_v(r)$ is the tree state obtained by applying, in log order, the effective changes of the entries naming r with version at most v. A changeset validated as FastForward or Rebase is appended with effective change Φ. A changeset validated as Merge is appended with the effective change produced by the three-way merge of each shared file. A *projection* is a function Π that maps each log prefix $(e_1, \ldots, e_v)$ and ref r to a Git commit whose tree is $\sigma_v(r)$.

Assumption 8.2 (Deterministic linear projection)**.** Π is deterministic, and when e_v names r the commit $\Pi(e_1, \ldots, e_v; r)$ has first parent $\Pi(e_1, \ldots, e_{v-1}; r)$.

Proposition 8.3 (Carry-over)**.** *Under Assumptions 4.14 and 6.2, with validation as in Definition 4.15 and the changeset's* W *in place of* $\mathsf{W}(c)$*:*

1. *Theorems 5.1 and 5.6 and Proposition 5.2 hold unchanged for the log, and $\sigma_u(r)$ is a function of u alone.*
2. *Assumption 6.1 holds by construction, because a changeset has no parent that could disagree with its base.*
3. *Lemma 6.3, part 1 of Theorem 6.4 and its serializability statement hold, with the entry's own changes in place of the installed commit. Parts 2 and 3 hold by definition: for FastForward and Rebase, $\sigma_v(r)$ equals $\sigma_{v-1}(r)$ with Φ applied, which is what the three-way merge of Theorem 6.4 produces when write sets are disjoint.*
4. *Theorem 6.7, Corollary 6.8, Propositions 6.5 and 6.10 hold unchanged.*
5. *Under Assumption 8.2, Theorem 5.4 and Corollary 5.5 hold for projected commits: the projected commit at version u is a first-parent ancestor of the projected commit at any later version, and has the same identifier whenever it is computed.*
6. *Theorem 7.3 applies to any batched command that validates members against the log but not against each other, and Proposition 7.4 applies to batches of changesets with* W *and* R *of each member.*

Proof. Part 1: those results use only the log, the chain and the insert-only table; objects enter Theorem 5.6 only through content addressing, and $\sigma_u(r)$ is a fold over $e_1, \ldots, e_u$. Part 2: the writer reads $\sigma_b(r)$ and states b; there is no second datum to disagree with it, which removes the exposure discussed after Assumption 6.1. Part 3: the proof of Theorem 6.4 uses commits only to identify the entry's write set and the state before and after the entry. With changesets, the entry's write set is W, the state change is Φ on W, and Assumption 6.2 holds at path granularity because σ changes only on W. The serialization-graph argument is unchanged. For disjoint writes, a three-way merge with base $\sigma_b(r)$, head $\sigma_{v-1}(r)$ and side $\sigma_b(r)$ with Φ applied takes Φ on W and the head elsewhere, which is the definition. Part 4: these results mention only access sets and validation outcomes. Part 5: by Assumption 8.2, each entry naming r adds one first-parent step to the projected commit, and induction gives the chain; determinism gives the fixed identifier. Part 6: the constructive history in

the proof of Theorem 7.3 uses only read sets, write sets and per-member validation against the log, and the proof of Proposition 7.4 uses only write-set disjointness and the read rule. □

The implementation at `dd6c9a8` realizes the design with two refinements. A changeset is the set of touched paths of a recorded commit, and that commit takes the changeset route only if its single parent is the branch's projected commit at its base, so Assumption 6.1 is checked rather than assumed; anything else takes the merge path. The authority lands up to 128 pending changesets in one command, appending them one at a time, and checks each member's touched paths and read set against the log's path index above its base. Earlier members of the batch are already in the log when a later member is checked, so the condition of Proposition 7.4 holds by construction on this path, and part 6 raises no anomaly there. The batch landing path merged at `fef94e7` submits up to 256 landings per command and states the condition explicitly: before routing a member it asks whether the member's read set or touched paths meet the writes of members already accepted in the batch, and defers it if they do. Part 5 moves the no-lost-update guarantee from the ref table to the projection, which Part II proves deterministic in its treatment of Git as a derived view (its Lemma 13.2).

9 Formal invariants

Table 3 lists the invariants the platform maintains, with the result or code check that establishes each. The first thirteen are the subject of this paper. The last two concern access control and review policy; they are enforced and tested in the code but not analyzed here.

Two gaps qualify the table. I3 is checked on replay, but a checkpoint's ref map is trusted once its digest matches (Theorem 5.1). I8 holds for single landings under Assumption 6.1, which the authority does not check for writer-supplied parents, and does not hold across members of one batch (Theorem 7.3).

10 Evidence

10.1 Correctness evidence

The theorems above are proved against a model, and the model is tied to the code by the file-and-commit table in Appendix A. The code's own tests check the properties the theorems depend on. Object identifiers match identifiers computed by Git for blobs, trees and commits. The incremental tree rewrite equals a rewrite from scratch on randomly generated trees. Replay rebuilds refs and rejects a broken chain. Validation rebases disjoint writes, conflicts on overlapping reads, treats opaque ref moves as conflicts, respects path component boundaries, and lets a symbol read conflict only with writes derived to touch that symbol. The combiner fails a stale read at claim time, refuses a batch under a stale epoch, and lands two members with one merge whose parents are the head followed by the members.

Table 4 lists the measured results this paper uses, with the conditions under which each was obtained.

Concurrent landing lost no work in the local test that checks Corollary 5.5. One hundred writers each committed one new file on the same branch and landed it, and all 100 landed. The commit message reporting the run gives a landing p95 of about 450 ms and a commit p95 of about 680 ms. Like the local commit-cost figures below, these come from the commit message that reported the run, because the run's raw output was not kept; the hosted figures in this section come from preserved raw output.

Table 3: Invariants of the repository authority. A result number refers to this paper; a test name refers to the code at commit `cfa4845`.

Id	Invariant	Established by
I1	Log versions are contiguous from 1, and every base precedes its entry.	Definition 4.10, replay checks
I2	Each chain digest hashes the previous digest and the entry.	Proposition 5.2, tamper test
I3	Each entry's old value equals the ref's value before it, so the ref table equals the replay of the log.	Theorem 5.1
I4	A ref move, its log entry, the projection target, any checkpoint and the landings row commit in one transaction with no network input or output.	Assumption 4.14, Definition 4.16
I5	A commit identifier is the Git hash of its bytes, and every tree supplied by the adapter is verified against its identifier.	Golden identifier tests and the untrusted-input test
I6	A commit is recorded only after the object store's readback matches what the authority built, and a hidden ref keeps it reachable.	Record command
I7	A read pinned at a version is unaffected by later writes.	Theorem 5.6
I8	A landed commit has no stale declared read since its base.	Theorem 6.4
I9	A batch merge sits on the current head, is built by the platform, and changes only paths its members changed.	Theorem 7.2
I10	A batch lands only under the lease's current epoch.	Theorem 7.2, fenced-lease test
I11	Landing a commit that already landed is a no-op, and landings are unique per commit and branch.	Land command, table key
I12	Refs only advance along first-parent chains, and repair and break-glass add commits.	Theorem 5.4
I13	Branch refs in the object store follow the authority, are never forced, and report a refused update as drift.	Projection test
I14	A path outside a token's grant is refused in the same way whether or not it exists.	Directory-grant test
I15	An unclassified or unknown change never counts as approved.	Auto-approval test

Commit cost did not grow with repository size, as Proposition 4.3 predicts. Locally, a one-file commit had a p50 of about 13 ms on a 100-file repository and about 14 ms on a 10,000-file repository. On the hosted deployment, the p50 was 1,725 ms at 100 files and 1,627 ms at 3,000 files in one run, and 1,302 ms and 1,085 ms in another. The hosted values measure network and object-store round trips more than the tree rewrite, and each comes from 20 edits on one client.

10.2 Hosted landing behaviour

The figures in this subsection come from the tree-building landing at `cfa4845`-era revisions. Under a burst of 100 concurrent landings, landing latency came from waiting on the lease and polling, not from the atomic append. In an instrumented hosted run, the direct compare-and-swap call had a p50 of 86 ms. The batch-landing call, which occurred about three times per hundred landings, had a p50 of 579 ms. By contrast, each landing request spent a summed p50 of 2.6 s in 13.6 claim attempts on average and 2.6 s in 14.6 status polls. These phase values are sums of overlapping waits, not service times, and the same run lost 7 of 100 commits to an object-store rate limit before landing began. The gap between them is still large enough to identify the next change: a completion signal pushed to waiting requests in place of polling.

End-to-end landing p50 ranged from 3.5 s to 5.3 s, and p95 from 5.0 s to 6.7 s, across two earlier hosted runs of 100 concurrent writers in which all 100 commits were accepted. The first run had p50 5.3 s and p95 6.7 s; the second had p50 3.5 s and p95 5.0 s, measured from each writer's land

Table 4: Measured results used in this paper, all from the Cloudflare reference implementation. Local runs used the workerd runtime on one developer machine with simulated storage latency; hosted runs used an isolated Cloudflare deployment with the client on one developer machine.

Quantity	Result	Setting
Concurrent landings without loss	100 of 100 writers committed and landed on one branch of a 100-file repository	Local, reported in a commit message
Three-way merge against Git	Agreement with `git merge-file` on 300 of 300 random merges of 20-line files	Unit test
Commit cost and repository size	One-file commit p50 about 13 ms at 100 files and about 14 ms at 10,000 files	Local, 30 edits each, reported in a commit message
Commit cost and repository size	One-file commit p50 1,725 ms at 100 files and 1,627 ms at 3,000 files	Hosted run 1, 20 edits each
Commit cost and repository size	One-file commit p50 1,302 ms at 100 files and 1,085 ms at 3,000 files	Hosted run 2, 20 edits each
Git compatibility	A clone of the projected repository returned the landed commit and its content, and passed `git fsck`	Hosted

request.

At those revisions the commit path missed its design target: across four hosted runs with 100 concurrent writers, commit p95 was between 2.8 s and 3.7 s, against a target of 1 s. Every commit waited on the Git store of the reference implementation, Cloudflare Artifacts, which accepted one receive-pack per repository at a time, and on several round trips to the authority. The first is a constraint of that product, not of the object-store primitive, whose contract is only immutable puts and gets.

Later committed revisions meet the commit target, and landing became the measured problem. With 100 writers, 28 of 39 hosted runs on later committed revisions had commit p95 under 1 s, over a full range of 437 to 2,356 ms. The four runs at revision `4407515` had 437 to 753 ms and the two at `03821e0` had 638 and 805 ms. Three caveats apply. These revisions acknowledge a commit once the authority or a commit shard durably holds its objects, before the copies in R2 and Artifacts exist. The benchmarks do not record whether the deployed tree was clean. And at 1,000 writers, commit p95 was 2,643 ms in a single run with overlapping landings. Landing with 100 writers had p95 of 4.2 to 5.4 s with polling at `4407515` and 25.7 to 30.1 s with the non-polling combiner at `03821e0`, measured from a developer machine and from a Worker in one cloud region respectively. At 1,000 writers the authority refused most landings as overloaded: 980 of 1,000 with polling, and 547 of 990 plus 155 timeouts with in-memory waiters. These are negative results for the tree-building landing, and changeset landing is the response to them. Part III analyzes the write path and its capacity in its Sections 5 and 6; nothing in this paper's analysis depends on those latencies.

11 Related work

The idea of a table as a log of atomic changes over immutable files comes from the lakehouse formats. Delta Lake's paper gives the log, the checkpoints and the argument that write transactions are serializable in record order because only one writer can create each record [3], and its protocol specification fixes the naming and the rule that writers never overwrite an entry [15]. Iceberg reaches

the same guarantees by an atomic swap of a metadata pointer [21]. The present design takes this structure intact and changes two things: the atomic step is a command of a serialized authority rather than a conditional write, and the conflict rules, which Delta's paper does not specify [3], are stated explicitly over declared read and write sets.

Optimistic concurrency control begins with Kung and Robinson, whose serial validation Section 6 maps onto landing [23]. The serialization-graph theorem and the treatment of granularity come from Bernstein, Hadzilacos and Goodman [6]. The isolation-level vocabulary is that of Berenson and colleagues [4] and of Adya, whose graph phenomena classify the batch anomaly of Section 7 [1], and the question of when snapshot isolation yields serializable executions is treated by Fekete and colleagues [17]. What is specific to source control is the write-write case. A database transaction that writes a row replaces it; a change to a file is a patch, and two patches to different regions of one file can both apply. The Merge outcome and Proposition 6.5 capture this, with the three-way merge whose semantics Khanna, Kunal and Pierce formalize [22].

Among source-control systems, Scion is the closest in intent: server-side changesets in a write-ahead log, refs moved by a compare-and-swap on one row, and no global merge lock [2]. Its row lives in Spanner [10]. The Scion post credits Mononoke, the server of Meta's Sapling, with pioneering many of its techniques, in particular the Bonsai data model [2], in which a changeset records its parents and a flat list of file changes [27]. Mononoke's pushrebase is the nearest precedent for server-side landing [28]. Google's Piper is the long-standing example of a single very large repository with a server-side source of truth [29]. YonedaRepo differs from all of these in keeping Git's objects and identifiers, so its changesets are Git commits and its trees are Git trees, and in validating declared read sets rather than file overlap alone.

12 Limitations

The theorems are proved about a model. The model's correspondence with the code was established by reading the code at one commit, not by mechanized verification or model checking, and the land, claim and lease interleavings have been tested by example, not explored exhaustively. A randomized or model-checked test of those interleavings is the strongest missing piece of evidence.

The serializability guarantee is relative to declared reads. Agents may read more than they declare, and Example 6.6 shows the consequence. Path-level reads are conservative but expensive in concurrency; symbol-level reads depend on a parser and on the soundness of derived symbol touches, which we assume rather than prove. Neither level captures semantic dependencies that cross files without a lexical reference, which is why post-land checks remain necessary.

Three implementation gaps were found in the course of this analysis: writer-supplied parents are not checked against the base version (Assumption 6.1); checkpoints are trusted on their digest alone (Theorem 5.1); and batch members are not validated against each other (Theorem 7.3). Each has a small fix, stated with the result that exposes it. The changeset route checks the base (first at `dd6c9a8`, Section 8) and the batch path checks members against each other (from `c4c8f47`, Section 7); both are on the development branch from `fef94e7`, neither is tested on a workload that could expose the gap, and the checkpoint gap is open.

The design serializes all landings, and all commit preparation, through one authority per repository. This choice keeps the atomic step simple and bounds throughput per repository; Part III (Section 6) models that bound.

Changeset landing (Section 3.6) is merged into the development branch at `fef94e7`; apart from its one headline result and the code-evidence rows that name its revisions, every measurement and code-evidence row in this paper concerns the tree-building landing. Several features of a complete

log are not built: time travel by timestamp, log compaction and retention, and splitting refs across several authorities.

The performance evidence is thin. The local figures lack surviving raw output, and the hosted runs come from one client machine, with run-to-run variation large enough that no single run should be read as precise. Part III states the planned evaluation in its Section 9.

13 Conclusion

A repository can be run as a database with a transaction log. With Git objects as the immutable data and a serialized authority as the atomic step, the main properties of the Delta design carry over with proofs: deterministic replay, immutable snapshots, and writes serializable in log order relative to what they declare they read, whenever no line-level merge is needed. Source control adds a write-write case that tables lack, and three-way merging handles it at the level of patches rather than values. The companion paper's independence condition has a precise meaning at landing: it is the condition under which two changes land without merging regardless of which goes first, which makes it the right rule for a scheduler and the wrong rule for a validator.

These results are what the transactional view of source code needs from its lowest layer. They let derived data be defined as views of a snapshot, which Part II does, and they reduce the cost of the system to the cost of one serialized authority per repository, which Part III measures. Two questions remain open. Batched landing needs one pairwise check before it is serializable; the changeset path satisfies it by construction and the merged batch path checks it explicitly, but neither has been tested on a workload that could exhibit write skew. And serializability holds only relative to declared reads, so the guarantee is as good as the read sets that agents and tools report.

References

[1] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, Massachusetts Institute of Technology, March 1999. MIT-LCS-TR-786. `https://dspace.mit.edu/handle/1721.1/149899`.

[2] Nick G. and the source control team. Redesigning source control at Anthropic. Anthropic Engineering, 13 October 2026, updated 15 October 2026. `https://www.anthropic.com/engineering/redesigning-source-control`.

[3] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411-3424, 2020. doi:10.14778/3415478.3415560.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1-10, 1995. doi:10.1145/223784.223785.

[5] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757-763, 1966. doi:10.1109/PGEC.1966.264565.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. `https://www.microsoft.com/en-us/research/people/philbe/book/`.

[7] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 729-738, 2008. doi:10.1145/1376616.1376690.

[8] Cloudflare. What are Durable Objects? Cloudflare Durable Objects documentation. https://developers.cloudflare.com/durable-objects/concepts/what-are-durable-objects/.

[9] Cloudflare. SQLite-backed Durable Object Storage. Cloudflare Durable Objects documentation. https://developers.cloudflare.com/durable-objects/api/sqlite-storage-api/.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, et al. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

[11] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, 2009. https://www.usenix.org/legacy/event/sec09/tech/full_papers/crosby.pdf.

[12] Delta Lake project. ConflictChecker.scala, commit d99585d. https://github.com/delta-io/delta/blob/d99585d85ad94660dcd88fd798a1d5241edebc4c/spark/src/main/scala/org/apache/spark/sql/delta/ConflictChecker.scala.

[13] Delta Lake project. Concurrency control. https://docs.delta.io/latest/concurrency-control.html.

[14] Delta Lake project. isolationLevels.scala. https://github.com/delta-io/delta/blob/master/spark/src/main/scala/org/apache/spark/sql/delta/isolationLevels.scala.

[15] Delta Lake project. Delta transaction log protocol (PROTOCOL.md). https://github.com/delta-io/delta/blob/master/PROTOCOL.md.

[16] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1-8, 1984. doi:10.1145/602259.602261.

[17] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492-528, 2005. doi:10.1145/1071610.1071615.

[18] S. Chacon and B. Straub. Git internals: Git objects. In *Pro Git*, second edition. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects.

[19] Git project. gitprotocol-pack: How packs are transferred over-the-wire. https://git-scm.com/docs/gitprotocol-pack.

[20] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111-120, 1984. doi:10.1016/0306-4379(84)90020-6.

[21] Apache Iceberg project. Iceberg table spec. https://iceberg.apache.org/spec/.

[22] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 485-496. Springer, 2007. doi:10.1007/978-3-540-77050-3_40.

[23] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213-226, 1981. doi:10.1145/319566.319567.

[24] M. Long and The YonedaAI Collaboration. The inversion of software maintainability in AI-native repositories. YonedaAI Research Collective, author's manuscript, 14 October 2026.

[25] M. Long and The YonedaAI Collaboration. Partitioned derived data. Part II of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[26] M. Long and The YonedaAI Collaboration. A cloud-portable architecture for serverless Git. Part III of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[27] Sapling project. Mononoke: Bonsai data model. `https://github.com/facebook/sapling/blob/main/eden/mononoke/docs/2.1-bonsai-data-model.md`.

[28] Sapling project. Mononoke: Pushrebase. `https://github.com/facebook/sapling/blob/main/eden/mononoke/docs/4.1-pushrebase.md`.

[29] R. Potvin and J. Levenberg. Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78-87, 2016. doi:10.1145/2854146.

[30] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299-319, 1990. doi:10.1145/98163.98167.

[31] M. Long. YonedaRepo source repository, branch `dev`, commit `cfa4845`, 20 October 2026. Cited by file and commit in Appendix A. Private repository; not publicly available.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15-28, 2012. `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.

A Code evidence

Table 5 maps each claim this paper makes about the implementation to a file in the YonedaRepo repository [31]. Every row can be checked by reading the named file at the commit in its row. Rows 1 to 29 use `cfa4845` on the development branch, the revision the model of Sections 4 to 7 describes; rows 30 to 33 use revisions of the branch that implemented changeset landing, which is merged into the development branch at `fef94e7`. Paths are relative to the repository root; `core` abbreviates `backend/crates/yoneda-core/src`, `platform` abbreviates `backend/crates/yoneda-platform/src`, and `store` abbreviates `backend/crates/yoneda-platform/src/repo_store`.

Table 5: Claims about the implementation and the code that shows them.

Row	Claim	File	Commit	What the code shows
1	Object identifiers are Git's	`core/git_object/id.rs`	`cfa4845`	`hash_object` computes SHA-1 over the type, a space, the length, a zero byte and the data.
2	Identifiers match Git's own	`core/git_object/tests.rs`	`cfa4845`	Golden identifiers from `git hash-object`, `mktree` and `commit-tree` are reproduced by `object_ids_match_git` and `commits_match_git_and_expose_trailers`.
3	Deltified pack entries are refused	`core/git_object/id.rs`	`cfa4845`	`from_pack_code` returns an error for pack types 6 and 7.

Row	Claim	File	Commit	What the code shows
4	Tree rewrite touches only changed directories	`core/git_object/rewrite.rs`, `core/git_object/tests.rs`	`cfa4845`	`rewrite_tree` rewrites directories on changed paths, and `rewriting_reads_only_directories_on_changed_paths` asserts an untouched subtree is never read.
5	Log entry fields and validity rules	`core/txlog/entry.rs`	`cfa4845`	`TxEntry` holds version, base, kind, ref, old, new, paths, read set, actor, time and chain; `validate` checks the base precedes the version, qualified ref names, deletion as the only entry without a new value, no repeated path, and the chain digest.
6	Chain digest definition	`core/txlog/entry.rs`	`cfa4845`	`chain_digest` hashes the previous digest together with the entry stripped of its own digest.
7	Replay checks contiguity, chain and old values	`core/txlog/checkpoint.rs`	`cfa4845`	`replay` rejects a version gap, calls `validate` with the previous digest, and rejects an entry whose old value differs from the replayed ref.
8	Checkpoints every 64 versions, trusted on digest	`store/rows.rs`	`cfa4845`	`CHECKPOINT_EVERY` is 64; `checkpoint_at_or_before` accepts a checkpoint only when its digest equals the log's digest at that version and does not recompute its ref map.
9	Every command runs in one transaction	`platform/repo_store.rs`, `backend/crates/yoneda-cloudflare/src/lib.rs`	`cfa4845`	`execute_repo_store` wraps dispatch in `db.transaction`; the Cloudflare store implements it with `transactionSync` and throws on an error result so the transaction rolls back.
10	Append writes entry, ref, projection, checkpoint and landing together	`store/land.rs`	`cfa4845`	`append` inserts the log row, upserts the ref, sets the projection target, calls `maybe_checkpoint` and inserts the landings row using only SQL calls.
11	Every append keeps the head as first parent or creates the ref	`store/land.rs`, `store/land_queue.rs`, `store/break_glass.rs`	`cfa4845`	The seven call sites of `append` use kinds commit, merge, revert and import only, and each checks an absent head, a first parent equal to the head, or merge parents beginning with the head.
12	Retried landing is a no-op	`store/land.rs`	`cfa4845`	`land` returns an already-landed result when the ref equals the commit or a landings row exists.

Row	Claim	File	Commit	What the code shows
13	Validation outcomes and opaque entries	`core/txlog/occ.rs`	`cfa4845`	`Proposal::check` returns Conflict, FastForward, Rebase or Merge; `stale_read` treats path-less entries of kinds other than commit and merge as conflicts.
14	Overlap of paths and symbols	`core/txlog/occ.rs`, `core/txlog/access.rs`	`cfa4845`	`overlaps` compares paths at component boundaries; `symbols_overlap` compares `::` prefixes; `Access::overlaps` handles the mixed case.
15	Unresolved writes stay whole files	`core/txlog/access.rs`	`cfa4845`	`AccessSet::writes` keeps a path whole unless a non-empty symbol refinement exists, as the test `unresolved_writes_stay_whole_files` checks.
16	Independence condition both ways	`core/txlog/access.rs`	`cfa4845`	`independent` returns true only when neither write set meets the other change's reads or writes.
17	Validation tests	`core/txlog/tests.rs`	`cfa4845`	Tests `independent_writes_rebase_and_overlaps_conflict`, `opaque_ref_moves_are_never_assumed_independent` and `path_overlap_respects_component_boundaries`.
18	Merge landing is verified, not recomputed	`store/merging.rs`	`cfa4845`	`verify` checks the parents, the `platform:merge` principal, the validation outcome and that the merge touches only paths the merged work touched.
19	Snapshot reads by version replay the log	`store/reads.rs`	`cfa4845`	`at` accepts only a version from 0 to the current version; `refs_at` replays from the nearest accepted checkpoint; no timestamp form exists.
20	Tampering is detected on time travel	`store/tests.rs`	`cfa4845`	`tampered_history_is_detected_on_time_travel` edits a log entry and expects an integrity error on a read at that version.
21	Combiner lease, batch size and claim	`store/land_queue.rs`	`cfa4845`	`LEASE_MS` is 30,000 and `BATCH` is 64; `claim` waits on an unexpired lease, increments the epoch, and fails stale reads at claim time.
22	Batch landing checks	`store/land_queue.rs`	`cfa4845`	`land_batch` refuses a stale epoch, and `verify_octopus` checks parents, principal, per-member validation against the log and changed paths, with no check between members.

Row	Claim	File	Commit	What the code shows
23	Adapter defers only on merged write overlap	`cloudflare/worker/repo-merge.ts`	`cfa4845`	`landBatch` defers an item whose merged paths overlap an earlier accepted item and does not compare read sets.
24	Combiner behaviour under test	`store/tests.rs`	`cfa4845`	`queued_landings_batch_under_a_fenced_lease` checks claim-time conflict, the held lease, refusal of a stale epoch and one merge landing two members.
25	Base consistency is not checked	`store/prepare.rs`, `cloudflare/worker/repo-ops.ts`	`cfa4845`	`prepare` checks only that the base is not newer than the log; `commit` takes the base and parent from one snapshot only when the writer names a branch.
26	Three-way merge agrees with Git	`backend/crates/yoneda-runtime/src/merge_interop_tests.rs`	`cfa4845`	`three_way_merge_matches_git_merge_file` compares 300 random cases with `git merge-file` and asserts agreement on every case.
27	Post-land failures propose repairs	`store/checks.rs`	`cfa4845`	A failing check on a landed commit lists each branch it landed on for a revert proposal, and `record_repair` links the proposed commit.
28	Merge verification refuses a FastForward outcome on a stale parent	`store/merging.rs`, `store/land_queue.rs`	`cfa4845`	`verify` accepts only Rebase or Merge outcomes; `verify_octopus` accepts FastForward only when the commit's first parent is the head.
29	Platform merge commits are prepared by the authority	`store/prepare.rs`, `cloudflare/worker/repo-merge.ts`	`cfa4845`	`prepare` accepts `merge_parent` and `merge_parents` and builds the tree from the parent and the change list; `landBatch` calls `commit` with the head as parent and the accepted members as merge parents.
30	Changeset landing validates by path index and appends in batches	`store/changesets.rs`	`dd6c9a8`	On branch `claude/native-p95e`: `route` sends a commit to the changeset route only if its single parent is the projected head at its base, and checks its touched paths and read set against `txlog_paths` above that base; `land` appends up to `BATCH` = 128 members one at a time in one command.

Row	Claim	File	Commit	What the code shows
31	Batch members that meet an earlier member's writes are refused	`store/land_queue.rs`, `store/merging.rs`	`c4c8f47`	On branch `claude/native-p95e`: Batch verification returns `WRITE_SKEW` when `BatchWrites::admits` finds that a member reads or writes a path an earlier accepted member wrote.
32	The combiner adapter on that branch still compares only merged writes	`cloudflare/worker/repo-merge.ts`	`dd6c9a8`	On branch `claude/native-p95e`: `landBatch` defers an item whose merged paths overlap an earlier accepted item and does not compare read sets.
33	The merged batch path checks members against each other before routing	`store/submitted.rs`	`fef94e7`	On the development branch: the submit command accepts at most 256 landings, keeps a `BatchWrites` set, defers a member when `admits` fails on its read set and touched paths, and otherwise calls `route_writing`, appending a changeset and recording its writes with `accept`.