Chorus: Order-Independent Structural Merging for Fleets of Concurrent AI Coding Agents

Yiming Beckmann
MingLLM
yimingbeckmann@gmail.com

July 4, 2026

Abstract

Fleets of autonomous coding agents invert a long-standing assumption of collaborative version control: concurrent edits to the *same file* are the common case, not the exception. The two standard responses both fall short. Merge queues serialize landings and sacrifice throughput; textual three-way merge (`diff3` and its descendants) merges lines in ways that are order-dependent and formally weak. We present Chorus, a multi-writer structural merge engine in which a source file is decomposed into per-symbol cells and writers submit typed *intents* (edit, add, remove, import-set) rather than diffs. The engine never mutates the canonical text incrementally: it accumulates all intents into order-independent sets keyed against a fixed lease-time origin, and recomputes the converged file and conflict set as a pure function of (origin, accumulated intent set). Land-order independence therefore holds by construction: any arrival order of the same request set yields an identical (canonical, conflict-set) pair, a property enforced by a standing cross-permutation oracle rather than claimed as a mechanized proof. The engine fails closed on every ambiguity, supports partial progress, prevents dangling references through a unified cross-symbol fixpoint, and adds a member-level merge sub-tier for provably commutative containers. Above the structural floor, a semantic gate operationalizes semantic conflict-freedom as a per-test differential over $N{+}2$ materialized trees. Chorus is implemented in Rust for twelve languages (about 25,000 lines across engine and fuzz harness; 565 tests). We evaluate with three instruments: a sixteen-round adversarial review panel of LLM agents, held to a mechanical-reproduction bar, that produced 42 counted findings (38 engine defects fixed fail-closed with pinning tests) and a taxonomy of silent-loss classes; a mutation-kill audit that showed the fuzz oracle was blind to three planted silent-loss mutants (voiding a prior 94.9-million-case campaign) and drove five new oracle floors; and a generative fuzzing campaign design targeting a 10^{-9} residual-risk gate. We report the audit's negative result deliberately: an oracle that cannot kill mutants proves nothing, and hardening the oracle is as much a contribution as hardening the engine.

1 Introduction

Autonomous coding agents are now capable of resolving real repository issues end-to-end [11, 25], and practical systems increasingly run *fleets* of such agents in parallel against a single codebase. In the system that hosts Chorus (Loom Conductor, an orchestrator that drives six concurrent Claude Code sessions and their subagents), a mission planner partitions work into tasks with declared write-sets, provisions an isolated git worktree per task, and lands finished work through a speculative merge queue: textual mergeability is probed with `git merge-tree`, the post-merge tree is gated by compile-and-test, and the exact gated commit is landed by compare-and-swap on the integration ref. This queue-level architecture is the industry-standard shape [4, 9], and it is kept as the outer backstop.

The queue does not, however, solve the problem that fleets actually create. When several agents must touch one file, a queue serializes them (destroying the parallelism the fleet was built for), and the underlying textual merge is the wrong primitive for the residue. Khanna et al. [12] showed that `diff3` satisfies almost none of the properties users assume: it is not idempotent, and the folk rule that well-separated edits never conflict is false. Worse for a fleet, a sequential fold of pairwise merges makes the *result depend on arrival order*: which agent finishes first changes the bytes that land, and for order-sensitive constructs it can change them silently. Convergence-oriented mechanisms (CRDTs) remove the order dependence but keep the deeper

failure: recent measurements of multiple LLM agents editing one code document through a CRDT report 100% convergence with 5–10% semantic breakage such as duplicate declarations and broken references [18]. Converged is not correct.

Chorus is a structural merge engine designed for exactly this setting, under one prime directive taken from its design documents: *soundness over completeness*. A spurious surfaced conflict costs efficiency; a silent loss of a writer's change is unforgivable. Every uncertain path in the engine terminates in an explicit, reason-coded refusal rather than a best-effort merge.

The engine's central idea is to make the merged file a *pure function of an unordered set*. A file is parsed (via pinned tree-sitter grammars) into per-symbol cells with kind-namespaced keys, plus an import cell and fail-closed "other" regions. Writers submit typed intents against a fixed lease-time origin, not diffs against a moving tip. Landing never splices into the live text incrementally; instead each landing folds the request's intents into per-cell sets and recomputes the entire (canonical, conflict-set) pair from the fixed origin and the accumulated sets. Because set union is commutative and associative and the recompute reads only the origin and the sets, any interleaving of landings over the same request set converges to the identical output (Section 3.4). "First to land" ceases to be observable.

This paper makes four contributions:

1. **An intent-set merge model** in which the converged file is a pure function of (fixed origin, accumulated intent set), yielding land-order independence by construction, partial progress (uncontended cells land even when siblings conflict), a conflict set that can *shrink* as sibling intents arrive, and a unified cross-symbol fixpoint that prevents both dangling references and their reintroduction by later phases (Section 3).
2. **A fail-closed multi-language structural tier** covering twelve languages at symbol granularity with a member-level sub-tier restricted to provably commutative containers in TypeScript/TSX, Rust, and Python, plus an advisory staleness guard (`StaleBase`) that blocks replay-induced reversion of landed work without reintroducing order dependence (Sections 3.6 and 5).
3. **A semantic gate** that operationalizes the semantic conflict-freedom criterion of Sousa et al. [22] (their Definition 4.4) as a per-test differential over $N+2$ materialized trees, with fail-closed treatment of errored and absent suites and a mechanical "may-only-add-conflicts" monotonicity invariant (Section 4).
4. **An assurance methodology and its honest results**: sixteen rounds of adversarial review by independent LLM-agent lenses whose findings count only when mechanically reproduced against the real engine; a mutation-kill audit that exposed the fuzz oracle as blind to three planted silent-loss mutants and voided a prior 94.9-million-case campaign; and the hardened oracle floors and campaign design that followed (Section 6).

We do not claim performance results against other merge tools, and we make no throughput measurements here; behavioral differences from `diff3`-style merging are characterized analytically (Section 3.8). The engine's evaluation is a correctness story, including its negative chapters.

2 Setting and Correctness Frame

The fleet context. Chorus sits below a conflict-aware merge queue and above nothing: it is the terminal authority on whether a set of same-file edits can be reconciled structurally. Tasks carry declared write-sets enforced at admission; a lease assigns each writer the symbols it may change. Cross-file coupling, wave scheduling, gating, and atomic landing remain the queue's job. Chorus's job is the file.

Correctness predicate. The design documents define a merge over base B and writer versions $W_1, \ldots, W_n$ onto a canonical C to be *Chorus-correct* iff it is MERGE-correct or CONFLICT-correct. MERGE-correctness is a conjunction: **C1** union-completeness (the merged symbol/member/import set equals the union of every writer's non-conflicting change: no drop, no duplication, no spurious content, no meaning-changing reorder); **C2** semantic preservation (the merge introduces no behavior absent from $\{B, W_1, \ldots, W_n\}$, adopted verbatim from Sousa et al. [22], Def. 4.4); **C3** determinism and idempotence (the merge is a function of the input set only, pinned across grammar versions). CONFLICT-correctness requires **C4** surfaced, never picked (no silent auto-resolution); **C5** conflict-set completeness over the full writer set at once, not pairwise (pairwise-clean-but-jointly-broken triples must surface); **C6** partial progress (a conflict among some writers must not block

the clean rest); and **C7** resolution stability. The measured bar is $P[\text{silent loss}] = P[\text{unsound clean}] = 0$ over an adversarial campaign; surfacing a conflict is always a correct outcome, and stopping the campaign is a risk decision, never a correctness claim.

The N-ary requirement (C5) is where prior art thins out: essentially the entire merge literature is two-variant (base/left/right). The canonical counterexample is three writers A, B, C where A and B each add a use of a symbol C removes: every pair is clean, the triple dangles. Chorus detects conflicts jointly over the full writer set.

3 Design

3.1 Symbol cells

A file is parsed with a pinned tree-sitter grammar into top-level *regions*: named declarations, imports, and *Other* (anything that is neither). Each declaration becomes a cell whose key is *kind-namespaced* wherever the language has multiple symbol tables: `fn f` versus `struct f` in Rust, `def f` versus `assign f` in Python, `phpfn foo` versus `class foo` in PHP (where a function and a class of the same name legally coexist). TypeScript keys are bare names. The import surface of the file is a single keyed cell whose reserved key carries a U+0001 control prefix so it can never collide with a real symbol. Other regions are first-class fail-closed citizens: a change that cannot be attributed to a named declaration or the import block surfaces rather than merges. Files the fence cannot key soundly are rejected wholesale: an unsupported language, a parse error (any tree-sitter `ERROR`/`MISSING` node), or a duplicated top-level name (`ambiguous_symbol`) is not a well-defined membership domain.

3.2 Intents, not diffs

Writers do not submit textual diffs. A `LandRequest` carries a writer id, advisory lineage fingerprints (`base_versions`, Section 3.6), and a list of typed operations: `Edit{key, new_text}`, `Remove{key}`, `Add{key, text, anchor}`, and `Imports{new_imports}`. Recovering operations from final-file diffs is lossy and ambiguous (moves and renames degenerate into delete-plus-insert); because the fleet *controls* its writers, the writers can declare intent instead, and the engine certifies rather than trusts it (a declared rename hint is verified by alpha-equivalence and reference-rewrite checks and otherwise fails closed). On the adapter side, a total translation exists: any residue that cannot be attributed to a named cell is packaged as a single whole-file edit under a sentinel key, which the engine deliberately refuses to merge silently: a whole-file intent is either the only intent (and then applied, reparse-verified) or it surfaces.

3.3 Accumulate, then recompute

The live file object holds the current canonical text, the *fixed* lease-time origin O (which never advances), and per-cell intent sets: edit texts with their proposing writers, removals, adds keyed by (text, anchor), import proposals, and the surfaced-cell map. `try_land` performs no splicing of its own. It (1) collapses the request's own repeated intents per key to the last one (a single request is one authored intent), (2) unions the collapsed intents into the global per-cell sets, and (3) invokes `recompute`, a pure deterministic function of (O, sets) that rebuilds the entire canonical and the entire conflict set from scratch. The recompute proceeds in phases:

```
recompute(origin O, accumulated sets S) -> (canonical, surfaced):
  A. per origin symbol: 0 changers => keep O's text; 1 => tentative
       Edit/Remove fate; Edit-vs-Remove => SURFACE (pin cell to origin);
       >=2 changers => StaleBase lineage check; layout-attribute gate;
       member-merge if the keyed-set conjunction holds, else SURFACE
  B. adds: origin-key collision, add-vs-edit/remove of one new key,
       body-must-define-exactly-this-key, or >=2 distinct (text,anchor)
       => SURFACE; else tentative Add fate
  C+D. unified cross-symbol fixpoint (Section 3.5)
  E. imports: one distinct proposal => 3-way union; else SURFACE
  F. render: AST-bounded splices into O (edits, adds in origin-anchor
       order, removals, imports), each reparse-verified with an
       intent-parity guard; any failure => revert cell to origin, SURFACE
  finally: any touched key not accounted for => SURFACE, never "landed"
```

Two engineering rules keep the recompute sound. Splices are AST-bounded: byte ranges come from the parse tree, never from line-prefix scans or whole-file string normalization (two early bugs, a license-header

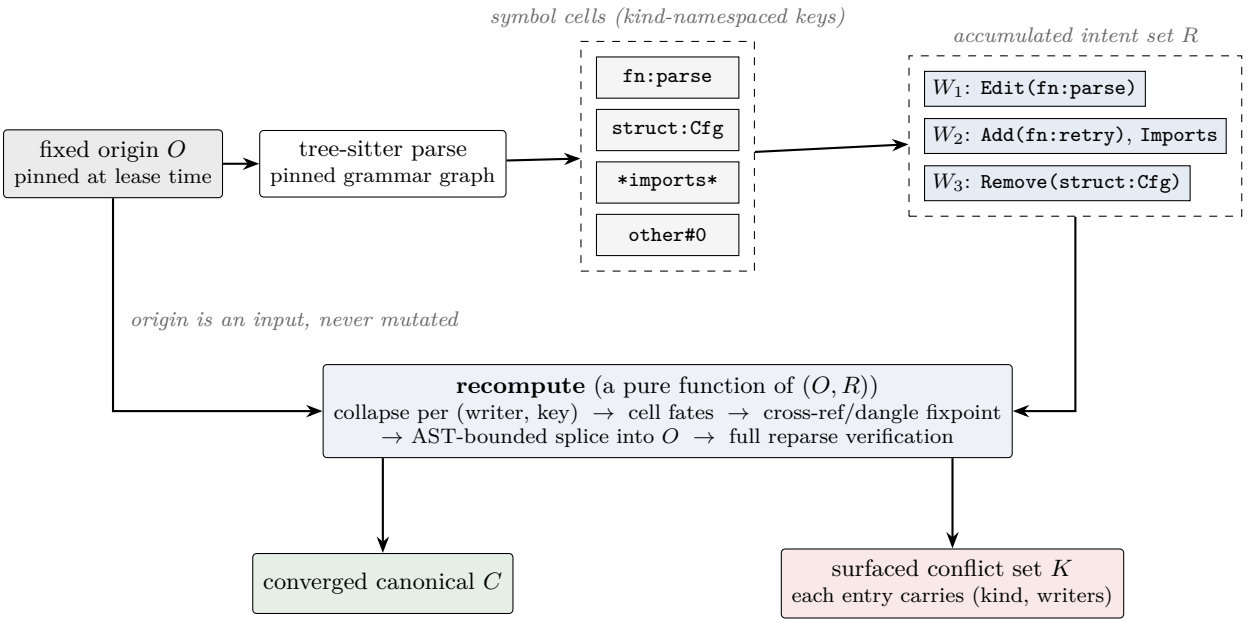

Figure 1: The Chorus pipeline. The origin is pinned at lease time and parsed once into keyed symbol cells; writers' typed intents accumulate into per-cell sets; the converged canonical and the conflict set are recomputed from scratch as a pure function of the pair. Every splice is AST-bounded and reparse-verified; every conflict entry carries its kind and the writers involved.

drop and a blank-line collapse that corrupted a surviving template literal, were both caused by textual scans and are now pinned regressions). And a work budget (`XREF_WORK_BUDGET` = 2,000,000 node visits) bounds the fixpoint; exceeding it fails closed by surfacing every pending removal rather than skipping the check. Figure 1 shows the pipeline.

3.4 Land-order independence

Property 1 (Land-order independence)**.** *Fix an origin O, a language, and a finite request set $R = \{r_1, \ldots, r_m\}$. For a permutation σ of R, let* fold_σ *denote sequential application of* `try_land` *in order σ starting from the live file of O. Then for all permutations σ, τ:*

$$\text{canonical}(\text{fold}_\sigma) = \text{canonical}(\text{fold}_\tau) \quad \textit{and} \quad \text{conflicts}(\text{fold}_\sigma) = \text{conflicts}(\text{fold}_\tau),$$

and both equal the output of the stateless batch entry point `resolve_requests`(O, R)*.*

Proof sketch. (i) `try_land` mutates only the accumulated per-cell sets, by set union of the request's collapsed intents; the collapse is local to each request, and union is commutative and associative, so the accumulated state after all of R is a function of R alone. (ii) The visible state is not threaded through landings: `recompute` reads only (O, sets), never the previous canonical, so the final (canonical, conflicts) pair equals `recompute`$(O, S(R))$. (iii) `recompute` is deterministic: it iterates ordered containers (B-tree maps and sets), resolves add anchors against the fixed origin with a total order over co-anchored adds, and runs a fixpoint whose cell fates move monotonically toward *Keep*, which terminates with a unique result. □

We state this as a property with mechanized *test* evidence, not a mechanized proof. The cross-permutation identity oracle is a standing fuzz judge: every generated concurrent scenario is landed under all permutations for $m \leq 4$ (Heap's algorithm) and under eight deterministic shuffles beyond, asserting bytewise-identical canonicals and identical surfaced sets; any divergence is a hard escape. The oracle demonstrably has teeth: deliberately breaking clause (ii) during development (for example, rendering co-anchored adds in land order instead of fixed-origin order) produced divergence escapes at specific seeds (25018, 14098, 15659, 4739), and an earlier removal of the surfaced-cell revert produced 220 land-order divergence escapes in one run. Two caveats delimit the property. First, it concerns the *converged* pair: the per-request outcome a writer observes at its own land time (`Landed`/`Partial`/`Conflict`) may reflect a transient conflict that a later sibling resolves.

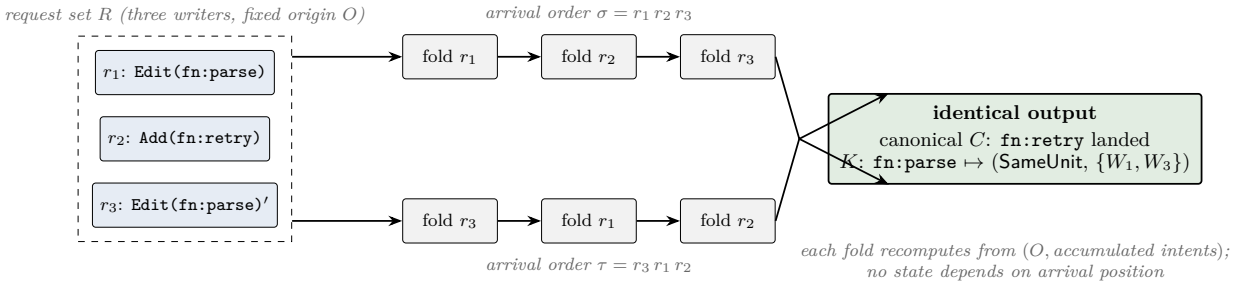

Figure 2: Two arrival orders over one request set converge to the identical (canonical, conflict-set) pair. Writers 1 and 3 submit distinct edits to `fn:parse` (a true contention: surfaced as SameUnit with both writers recorded); writer 2's disjoint add lands in either order. No outcome depends on who finished first.

Second, the property is conditional on the orchestration invariant of Section 3.6: while the origin is pinned, every outstanding request to a cell must remain in the resolved set.

Figure 2 illustrates the hero case: two arrival orders over the same three-writer request set, converging to one canonical and one conflict set.

3.5 The cross-symbol fixpoint and partial progress

Symbol-local reasoning cannot see the N-ary dangling-reference class, so removals and edits are checked jointly. Phase C *reinstates* any removal still referenced by a surviving symbol's resolved body (surfacing CrossSymbolRef with both the remover and the referencer recorded). Phase D *reverts* any edit, and drops any add, whose resolved body references a declared-but-absent name. The two must converge together: a Phase-D revert of an edit back to its origin body can reintroduce a reference to a symbol Phase C had already cleared for removal (the clearance was validated against the edited body; the revert withdraws that premise). The engine therefore wraps both in an outer loop that re-enters Phase C after any Phase-D change until a full pass changes no fate. Termination is by monotonicity: every change moves a fate toward *Keep* or drops an add, so the active set strictly shrinks. Three separate ledger entries (bugs S, W, and a render-path variant II) document rediscoveries of this interaction before the unified fixpoint closed the class.

Because the surfaced set is rebuilt from scratch on every recompute, *conflicts can shrink*: a removal that dangles only until its referencer is also removed stops being a conflict once the sibling intent arrives. And because fates are per-cell, a request touching $\{A, B, C\}$ where only B contends receives `Partial`$\{\text{landed} = [A, C],\ \text{conflicts} = [B]\}$: the clean subset is never blocked (correctness axis C6).

3.6 The member-merge sub-tier and the staleness guard

When two or more writers edit the *same* symbol, surfacing is sound but pessimistic: often they touched disjoint members of one container (two methods of a class, two fields of an interface). The member tier merges exactly that case, and only under a five-part conjunction: (A) the container kind is a proven commutative keyed container; (B) member identity is complete and unambiguous; (C) added or edited members have pairwise-distinct keys across writers; (D) no two writers edit the same-key member to different content; (E) no key is concurrently added by one writer and removed by another. Any failure, and any result that does not reparse, surfaces.

The tier is deliberately narrow: TypeScript/TSX interfaces, classes, object literals, and plain enums; Rust structs, enums, and inherent/trait impls; Python classes. Rust and Python join only for the provably order-irrelevant subset: a Rust container with any `#[repr(...)]` attribute, an enum with explicit discriminants, or a tuple struct is order-sensitive and surfaces, as does a Python class that is decorated, declares `__slots__` or a metaclass, or has base classes (method-resolution order), and any class body with mixed tab/space indentation (which tree-sitter mis-groups silently). Every other language fails closed at this tier; Go stays out by decision, not omission, because struct field order is reflection-, layout-, and tag-significant. A refinement from the final review round counts contention per *runtime namespace cell* rather than per syntactic form: Python `def x` and `x = ...` bind one attribute, and a Rust `fn hot` and `const hot` share the value namespace (their union is a compile error), so concurrent bindings of both forms are one contended cell, never a silent union.

Condition	Treatment
Duplicate top-level keys (split `impl` blocks, `#[cfg]` twins)	`ambiguous_symbol`: whole file refused
Namespace/module/class reopens (C++ reopened namespace, Ruby reopened module, Swift same-type extensions)	duplicate key ⇒ fail closed
Order-sensitive containers (`repr(C)`/packed structs, tuple structs, explicit discriminants, `const enum`, `@dataclass`, `__slots__`)	never member-merged; ≥ 2 writers surface
Ruby magic comments (all MRI-honored spellings) and Go directive comments (`//go:build` etc.)	fenced trivia: any toggle or drop surfaces
Multi-binding declarations (grouped `const`/`iota`, `int a,b;`, tuple `let`, Kotlin destructuring)	Other region or refusal; never split-keyed
Same-cell distinct edits; edit-vs-remove; add-vs-add with distinct bodies; add colliding with an origin key	SameUnit surfaced with writers recorded
≥ 2 distinct import proposals	ImportContention
Removal referenced by any survivor; edit/add referencing an absent declaration	CrossSymbolRef via the fixpoint
Stale absence-as-removal in a member fold	StaleBase, pinned to origin
Rename without a certified move hint; hint failing alpha-equivalence or reference-rewrite checks	MoveHintUnverified
Whole-file sentinel co-occurring with per-symbol intents; edits to absent/non-declaration keys	surfaced, never reported landed
Any parse error, reparse failure, or splice-parity mismatch	revert to origin and surface

Table 1: The fail-closed catalog (abridged). Every row ends in an explicit conflict or a closed reason code; no row ends in a best-effort merge.

Staleness without a CAS. An optimistic per-symbol compare-and-swap was designed and then deliberately rejected: gating a land on lineage fingerprints makes "who landed first" observable, which is exactly the order dependence the model exists to remove. The request's `base_versions` therefore remain *advisory*, consulted for a single fail-closed check (`StaleBase`): before any absence-of-a-member is folded as a removal in a multi-writer member merge, the request's recorded fingerprint for that cell must equal the pinned origin's; a missing or mismatched claim marks the request stale and surfaces the cell pinned to origin. This closed a replay hazard found in the final review round, in which re-submitting an already-landed request set against a re-pinned merged origin read each stale text as "removed the sibling's member" and silently deleted landed members with a success status. The guard compares against the fixed origin only, so order independence is preserved. The complementary orchestration invariant is documented and asserted: the host may not commit-and-drop a sibling's intent and then re-pin the origin; re-pinning is sound only after all writers' intents are folded (or committed and the origin advanced to the committed state).

3.7 The fail-closed catalog

Table 1 summarizes the refusal catalog. The umbrella rule, quoted from the invariants document: when the engine cannot *prove* a merge safe, it surfaces; false positives are a tolerable efficiency cost, false negatives are unforgivable.

3.8 A worked contrast with textual merge

We ran no comparative benchmark and claim none; the difference is architectural and can be stated analytically. Consider a base file with functions `alpha` and `beta`, writer 1 appending function `gamma` after `beta`, and writer 2 appending function `delta` after `beta`. A sequential textual fold merges one writer, then merges the other against the moved tip: with `diff3`-class tools the second merge sees overlapping context at the append point, so the outcome (clean append, interleaved lines, or a conflict marker) depends on which writer landed first and on context-line luck; Khanna et al. [12] give the general form of this instability. In Chorus, both intents are `Add`s with distinct keys anchored against the *fixed* origin; a deterministic total order over co-anchored adds places them identically under either arrival order, and the cross-permutation oracle asserts exactly that. Conversely, when the two writers edit the same function to different bodies, a textual merge with non-overlapping hunks inside the function may silently interleave them into a function neither writer wrote; Chorus surfaces SameUnit with both writers recorded, unless the edits fall in the provably commutative member subset. The trade is explicit: Chorus refuses more, and never picks.

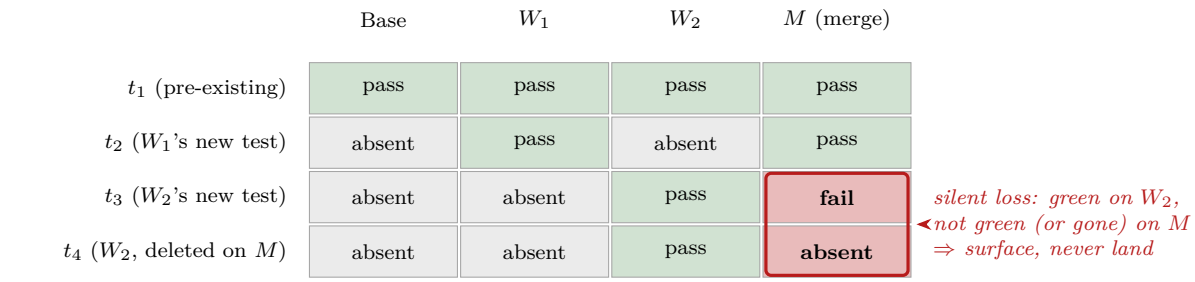

Figure 3: The C2 vector diff over {Base, W_1, W_2, M}. Test t_3 passes on W_2's own tree and fails on the merge; t_4 passes on W_2 and is absent on the merge. Both are silent-loss events and surface. A pre-existing test that stays green (t_1) and a writer's preserved new test (t_2) impose no conflict.

4 The C2 Semantic Gate

The structural floor proves no drop, duplication, spurious content, or reorder of tracked units. It is necessary, not sufficient: a merge can be structurally clean and behaviorally wrong. The C2 gate targets exactly the residual class, adopting the semantic conflict-freedom criterion of Sousa et al. [22] (Definition 4.4: wherever writer i changed an observable output relative to base, the merge reproduces writer i's output; where no writer changed it, the merge matches base) and replacing their relational verifier with test execution, in the spirit of test-based semantic conflict detection. SafeMerge itself is rejected as a shipped component for documented reasons: it is defined for one small imperative language, does not support renames or signature changes, and reports no counterexample; the gate approximates its criterion with real execution and concrete failing tests.

Mechanism. For a merge candidate M produced from writers $W_1, \ldots, W_n$, the gate materializes $N+2$ trees (each writer's own tree, the base tree, and M), runs the relevant test suites on each, and *diffs the per-test outcome vectors*. Outcomes are `passed`/`failed`/`errored`/`absent`, with `absent` distinct from `failed` because a test that disappears on M (deleted, or never compiled) is itself a loss signal. The silent-loss predicate, per test t and writer i:

$$\text{out}_{W_i}(t) = \text{passed} \;\wedge\; \text{out}_M(t) \in \{\text{failed}, \text{errored}, \text{absent}\} \;\Rightarrow\; \text{surface a silent-loss conflict for } (i, t),$$

unless a conflict for that unit was already surfaced structurally. A test not passing on W_i carries no preserved-intent obligation. The base vector never suppresses a loss; it only classifies whether a green-on-W_i test is *distinguishing* (witnesses the writer's new behavior: not passed, or absent, on base). Under sequential landing, "base" for the k-th lander is the moving canonical the splice targeted, not the original lineage base; using the original would both re-litigate already-landed work and mask regressions that exist only relative to the post-landing state. Figure 3 shows the vector-diff shape.

Fail-closed posture. The verdict is a structured object, not a boolean; its type makes "I ran the suite on M and it was green" unrepresentable as a pass. Specific rules: a whole-suite compile failure or unparseable runner output on M is *indeterminate* and surfaces; the same on a witness tree W_i surfaces an indeterminate conflict for that writer (an early review found erroring witnesses being swallowed as vacuous, letting a different writer's green test certify the merge; the fix is pinned). A writer with no distinguishing suite is a vacuous slice; if *no* distinguishing test ran anywhere, the verdict is `vacuous` and never certifies: the only positive signal (`earned: clean`) requires at least one distinguishing test to have run, no loss events, and a non-errored M. A candidate loss is cleared as a flake only by a stable green on *both* M and the witness across identical-tree reruns; a flaky witness is indeterminate and surfaces (the exact opposite polarity of the throughput-oriented land gate, whose retries exist to rescue greens). Test quarantine never rescues a C2 loss. Finally, monotonicity is mechanical: the output conflict set is initialized from the structural conflict set and every code path can only add to it, so the gate may add conflicts and can never clear a structural one.

C2 is necessary-not-sufficient by construction: it certifies that no *tested* writer behavior regressed, nothing more. It is implemented (534 lines of TypeScript plus a runner, 26 dedicated unit tests within a 108-test

module suite), dark behind its own flag, and requires a runnable per-language toolchain at land time (Section 8).

5 Implementation

Chorus is implemented as two Rust crates plus a TypeScript integration layer inside the Loom Conductor application. `chorus-core` (about 14,500 lines: the two-way symbol-fenced splice, the import union, the N-ary batch merge, the member tier, the concurrent intent-set path, and a conservative source canonicalizer) carries 416 library tests and 3 integration tests; `chorus-fuzz` (about 10,500 lines: generators, layered oracles, and a long-running campaign binary) carries 144 tests plus 2 pinned regression tests. All 565 pass at the time of writing. The crate is deliberately free of application dependencies (tree-sitter, serde, blake3 only) so the fuzz harness links the real merge code directly.

Twelve languages, one grammar graph. The structural tier covers TypeScript/TSX, Rust, Python, Go, Java, C, C++, C#, Swift, Kotlin, PHP, and Ruby at symbol granularity; the member tier covers TypeScript/TSX, Rust, and Python only (Section 3.6). Determinism across the fleet (C3) requires that every worker parse identically, so grammar versions are pinned to a single tree-sitter 0.24 core with all thirteen grammar crates at ABI 14, resolved so that exactly one `tree-sitter` version exists in the dependency graph. The pin discipline is finicky in practice: the C# grammar is pinned exactly (`=0.23.1`) because a later patch release crossed to ABI 15 *inside the patch line*; Swift is pinned exactly (`=0.7.0`) for the same reason, verified by loading each candidate; Kotlin uses the `kotlin-ng` crate because the older binding would drag a second tree-sitter into the graph; Ruby's ABI was verified by a runtime probe. The engine version itself is folded into a merge-config hash alongside grammar versions so mismatched workers refuse to merge rather than diverge.

Language quirks. Multi-language soundness is dominated by per-language sharp edges, several of which were found adversarially (ledger IDs in parentheses; Section 6.1):

- **C++/C# multi-segment keys (NN).** Nested-namespace symbols key as `a::b` / `A.B`, but the dangle scan matched single identifier leaves, so a qualified use like `a::b::f()` could never match: removals shipped dangling qualified references with a clean status. The scan now matches the trailing segment, which every qualified usage contains as a leaf under the pinned grammars. Separately, C# *file-scoped* namespaces (`namespace N;`) are treated as Other regions: renaming one surfaces rather than merges (a pinned test).
- **PHP name-kind leaves and the `<?php` shim (OO).** Every PHP identifier leaf has node kind `name`, not `identifier`, so the reference scan was definitively blind for the entire language; and a tag-less PHP fragment parses as one inert `text` node with zero identifiers and no error flag. The engine now accepts the `name` kind and shims `<?php` onto fragment parses in both the body-defines-key check and the reference scan; without the shim every PHP concurrent edit would fail closed and every dangle scan would return a false "no reference."
- **Kotlin own-line annotations (PP).** The grammar parses an own-line annotation *with arguments* (`@Deprecated("old")`) as a root-level sibling of the declaration it annotates. Removing the function silently re-attached the annotation to the next symbol, deprecating an innocent neighbor, with a clean reparse. The removal range now takes the annotation with the symbol, under a strict structural guard so a genuine annotated expression is never swallowed.
- **Ruby magic comments (QQ, RR).** The comment-run attachment that removes a symbol's documentation block with it extended over *every* contiguous comment, including `# frozen_string_literal: true`, whose semantics are file-wide: a removal silently flipped string mutability for the whole surviving file. The run now stops at magic comments, and the fence recognizes all spellings MRI honors (no-space variants, the Emacs `-*-` wrapper, `coding:`/`coding=`), with Python's PEP-263 header fenced as a byproduct.
- **Go directive comments (R).** `//go:build`, `//go:generate`, `//go:embed`, and cgo preambles are semantically load-bearing trivia; dropping them on splice was silent loss. They are fenced, and Go additionally stays out of the member tier entirely.

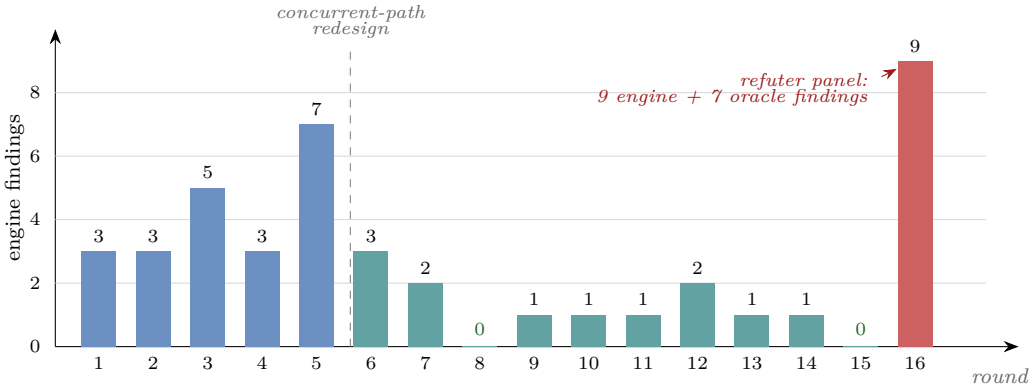

Figure 4: Engine findings per adversarial round, as counted by the regression ledger. Rounds 1–5 drove a full redesign of the concurrent path; rounds 8 and 15 were clean; round 16 was a refuter panel that probed language classes and replay semantics the campaign could not reach, and additionally audited the fuzz oracle itself (Section 6.2). Round 4's count includes two findings later disproven by mechanical reproduction.

The concurrent path is wired to the application through thin Tauri commands and a TypeScript adapter, all dark behind three independent fail-closed flags (engine, concurrent-land, C2), each with a runtime kill-switch that overrides even a build with flags baked on. A flip-everything build with the runtime kill-switch retained was produced on 2026-07-04; the merge queue of Section 2 remains the default path (Section 8).

6 Evaluation

Three instruments, all real and all reported including their failures: an adversarial review panel (the primary bug-finder to date), a mutation-kill audit of the fuzz oracle (a deliberate attack on our own evidence), and the generative fuzzing campaign (designed, partially run, and honestly voided once; re-run pending).

6.1 Sixteen rounds of adversarial panels

Method. After every build increment, independent review agents (LLM agents, run as subagents of the build orchestrator) are dispatched with *lenses*: narrow adversarial briefs such as "attack land-order independence," "attack the import union," "what metadata does the concurrent path emit," or "deep-dive Python and Go." Rounds 1–5 ran 20 lenses (four per round). A finding counts only when mechanically reproduced against the real engine: since round 10 the discipline is that every claim is reproduced by a standalone probe crate *before* any fix, and re-run green after; this matters because the ledger records two early claims (C attributed-declarator mis-keys) that compile-level reproduction disproved. Every confirmed finding becomes a permanent entry in a regression ledger with a repro, a root cause, a fail-closed fix, and a pinning test; the halt criterion is K *consecutive* fully-clean rounds, with the counter reset to zero on any escape.

Results. Sixteen rounds (2026-06-30 through 2026-07-04) produced 42 counted findings: 38 engine defects fixed fail-closed with pinning tests, two accepted as sound over-surfacing (fail-closed completeness costs, consistent with the prime directive), and two refuted by mechanical reproduction. Additional accepted-limitation entries (for example Go removals over-surfacing because a bare Go body does not parse standalone) are recorded outside the escape counts. Figure 4 shows the per-round trajectory. The signal is not a clean decay: rounds 1–5 found $3, 3, 5, 3, 7$ findings, and 11 of those 21 concentrated in the concurrent-land path, which was then re-derived once against all 11 reproductions as its test suite rather than patched 11 times. Post-redesign rounds found $3, 2, 0$ (round 8 the first clean round), then $1, 1, 1, 2, 1, 1, 0$, characterized in the ledger as "narrow symmetry/parity gaps (a protection one op/tier/language has that a sibling lacked), not structural." Then round 16 found nine more.

A taxonomy of silent loss. The ledger's 38 fixed defects cluster into recurring classes, which we offer as a checklist for any structural merge tool:

- *Attachment orphaning* (V, CC, FF, PP, QQ): an attribute, annotation, or doc comment re-attaches to the wrong neighbor across a removal or insertion (`#[derive]` orphaned onto the next Rust item; a Kotlin annotation deprecating an innocent function; a Ruby magic comment swallowed as documentation).
- *Dangle-scan blindness* (E, H, NN, OO): the reference scan that guards removals cannot see whole reference classes (kind-namespaced keys never matching identifier leaves; multi-segment namespace keys; an entire language whose identifier leaves have a different node kind). A blind scan returns a *definitive-looking* "no reference."
- *Re-dangle past the fixpoint* (S, W, W2, II): a later phase (edit revert, render fallback) reintroduces a reference whose removal an earlier phase had validly cleared; closed by unifying the phases into one fixpoint and enforcing parity at render.
- *Invisible-trivia drop* (A/G, B, D, R, RR): semantically load-bearing text outside modeled node ranges vanishes (license headers, TypeScript import attributes, side-effect imports, Go directives, magic-comment spellings).
- *Identity and canonicalization gaps* (C, GG, KK, LL): two spellings of one runtime entity key as distinct (`1` vs. `0x1` member names, escape and NFKC identifier forms, `def x` vs. `x = ...`), so a union double-declares or one writer's binding is silently dead.
- *Silent no-op with false credit* (J, K, X, DD, JJ): the engine applies nothing but reports the writer landed (edits to absent keys, the whole-file sentinel, a whitespace-normalizer classifying a real edit as no change).
- *Replay staleness* (I, L, MM): stale lineage silently reverts landed work (the blind-overwrite delete path; replaying a landed request set against a re-pinned origin), closed by the no-blind-overwrite invariant and StaleBase.
- *Fail-open gates* (O, Z): a verifier that swallows an error certifies what it could not evaluate (an erroring C2 witness suite treated as vacuous; a surfaced cell rendered anyway).

Round 16 as a case study. The final round (2026-07-04) re-ran the owed member-merge lens as a *refuter* panel: its brief was to disprove the engine's claims, with every claim probed against the real binary before being ledgered. It produced 16 findings: nine confirmed engine silent losses (JJ through RR: the whitespace-collapse edit drop, identifier-form and namespace-cell identity gaps, the stale-replay member reversion, the C++/C#/PHP dangle-scan blindness, and the Kotlin/Ruby attachment hazards of Section 5) and seven findings against the fuzz oracle itself (Section 6.2). All nine engine defects were fixed fail-closed the same day across four commits, adding 19 pinning tests (the library suite grew from 397 to 416), and the clean-round counter was reset to zero. The methodological point generalizes: adversarial panels of LLM agents with narrow lenses, a mechanical-reproduction bar, and a consecutive-clean-rounds halt criterion found 38 real silent-loss defects in an engine that a 94.9-million-case fuzz campaign had passed clean. We believe this panel-plus-probe loop is a reusable assurance method for merge tools generally.

6.2 The mutation-kill audit of the oracle

A fuzz campaign is evidence only if its oracle can detect the failures it claims to exclude. Round 16 tested this directly by planting silent-loss mutants in the engine and asking whether the campaign's judges would catch them. The control arm (M0), a mutant of a class the oracle explicitly models, was killed at a high rate (658 detections in 50,000 cases), confirming the harness kills what it is designed to see. The three adversarial mutants were not: M1, a `Remove` that lands as a silent no-op, rode 200,000 cases with zero detections; M2, a deterministic symbol reorder correlated with conflict handling, rode 200,000 cases with zero detections *and* was invisible to all 397 then-current unit tests; M3, deletion of a file header line (an Other region), was invisible to the oracle for every non-Go language. The commit that records the audit states the verdict plainly: the storm and batch oracles were *mutation-blind*; three planted silent-loss bugs rode 200k cases clean.

The root causes are instructive. The permutation-identity judge is structurally blind to any *order-independent wrong output*: an engine that ships the same wrong bytes under every permutation passes it by construction, which is precisely the class the panel's late-round bugs occupied. The dangle judge called the engine's own reference-scan primitive, the very function that was blind for PHP and multi-segment keys: a blind engine graded itself clean. And no judge asserted that a reported-landed operation had any *effect* at all.

Five oracle floors were built in response, each shipped with its own mutation-kill self-test: an *ops-effect floor* (every reported-landed operation's effect is asserted against the converged state: a landed remove leaves the key absent, a landed add/edit leaves the writer's fingerprint present, a surfaced cell pins to origin); a *no-blind-overwrite floor* (two distinct raw intents on one cell that converge clean is an unsound-clean escape); a *placement floor* (origin symbol order preserved for non-removed symbols in the storm judges); an *Other-region line floor* (every untouched non-symbol origin line survives verbatim, closing the header-deletion class for all languages); and an engine-independent *dangle text net* (a second, text-level scan for calls and qualified uses of removed keys that shares no code with the engine primitive it checks). The hardened harness passes its 144 tests, including the self-tests that verify each floor kills its mutant.

We regard this audit, and its initial negative result, as a feature of the evaluation rather than an embarrassment to it: the alternative was continuing to cite a large zero-escape number that measured the oracle's blindness, not the engine's correctness.

6.3 The generative fuzzing campaign

The campaign machinery is a generator suite (74 weighted scenario classes, including per-language disjoint and same-symbol generators for all twelve languages, order-sensitive construct generators whose *clean merge* is the escape, conflict-must-surface generators carrying a constructed witness, and concurrent storms that cycle all structural languages) judged by layered oracles: the C1 membership floor (exact symbol-set equality after normalization; no drop, duplication, or spurious content), the placement judge (base-relative order preserved), import-set equality with removal-wins semantics, the member floor (member name *and* body fingerprint equality), the cross-permutation identity judge of Section 3.4, the dangle judges (including the new text net), a differential judge against `git merge-file` restricted to the safe disjoint class, and a batch-equivalence judge (folding `try_land` equals the batch `merge_n` on the no-contention class), now extended by the five floors of Section 6.2. Every case is seed-deterministic and every escape shrinks to a pinned regression.

The ship gate is stated as a risk bound, not a proof: by the rule of three, zero escapes over K cases bounds the escape probability at roughly $3/K$ (95% confidence), and the target is 10^{-9}, reached at approximately 3×10^9 cases (about two days at the measured ∼16k cases/second on the development machine). We state plainly where this stands: a prior campaign was stopped at 94.9 million cases with zero escapes (nominal risk bound 3.16×10^{-8}) and then *voided*, because the round-16 audit proved those zeros were partly vacuous; the ledger's own wording is that the campaign "ran the blind oracle and does not count toward the gate." The full campaign on the hardened oracle and the current binary had not been re-run from zero at the time of writing; the number we can honestly report today is not 3×10^9 but the audit that explains why we discarded 9.49×10^7.

7 Related Work

Textual and structured merge. Khanna et al. [12] formalized `diff3` and proved it lacks the properties users assume, motivating merge tools that operate above lines. Semistructured merge [2] and structured merge with auto-tuning (JDime) [3] merge on program structure for Java, with large-scale evaluations and improvements by Cavalcanti et al. [6] and conflict characterization by Accioly et al. [1]. Spork brought structured merge with formatting preservation to Java tooling [15]; Mergiraf is a contemporary syntax-aware git merge driver across many languages [8]; and SemanticMerge was an earlier commercial language-aware merge tool [7]. Chorus shares the structural substrate but differs in posture and setting: these tools maximize automatic resolution for two-variant merges of uncontrolled human edits and are explicitly heuristic (the commutative-parent assumption is a documented silent-wrong-merge vector), whereas Chorus is N-ary, default-deny (commutativity must be proven per construct or the merge surfaces), and receives declared intents from writers it controls.

Operation-based merging and patch theory. Lippe and van Oosterom [16] argued for merging operations rather than states; Chorus's typed intents are an operation-based model made honest by engine-side certification. Darcs [20] and Pijul [23] build version control on patch commutation, where merge order-independence is the foundational law; Chorus imports that law as its headline invariant but enforces it by recomputation from a fixed origin rather than by a commutation theory over patches, and treats conflicts as first-class surfaced data.

CRDTs. Conflict-free replicated data types [21], sequence CRDTs such as RGA [19], JSON [13] and tree CRDTs [14], rich-text CRDTs [17], and interleaving-minimal designs [24] guarantee convergence without coordination. Chorus shares the goal of order-independence but rejects convergence as the bar: a CRDT converges on *some* state, whereas a merge engine must converge on a *correct* state or refuse. The CodeCRDT measurement (100% convergence, 5–10% semantic breakage for concurrent LLM writers) [18] is the cleanest empirical statement of the gap, in exactly Chorus's setting; Chorus instead borrows CRDT-adjacent machinery as oracle predicates and keeps a conflict channel.

Semantic merge and conflict detection. Horwitz et al. [10] gave the first sound semantic integration via program dependence graphs; Sousa et al. [22] defined semantic conflict-freedom and verified it with relational reasoning over edit-local product programs. Chorus adopts their Definition 4.4 as the written C2 criterion and operationalizes it with test execution over the $N{+}2$ tree quadruple, trading completeness for language-agnostic executability and concrete counterexamples. Proactive conflict detection in collaborative development (Crystal) [5] anticipates conflicts by speculatively merging early; Chorus's standing recompute over accumulated intents is a per-file analogue with a sound refusal channel.

Merge queues and coding agents. Merge queues [4, 9] serialize integration and gate the post-merge tree; Chorus keeps that outer loop and attacks the intra-file serialization it forces. Agent benchmarks and systems [11, 25] established that LLM agents produce mergeable-quality patches at scale; fleets of them are the workload Chorus exists for, and, in this paper's evaluation, also the adversaries that audited it.

8 Limitations and Future Work

Member tier coverage. Member-level merging is live for TypeScript/TSX, Rust, and Python only, and for Rust/Python only on the provably order-irrelevant subset; the other eight languages surface all same-symbol contention. Go's exclusion is a decision (field order is reflection- and layout-significant), but Java, C#, Kotlin, and Swift member tiers are future work gated on per-construct commutativity proofs.

Symbol granularity. Two writers editing the same function body to different texts surface SameUnit even when a statement-level merge would be safe; the design deliberately floors leasing at named declarations and members. The routing layer above the engine (structured tier, then author re-run, then human) absorbs this cost; quantifying it on real fleet workloads is open.

StaleBase is advisory. The engine deliberately has no per-symbol CAS; staleness is caught only for absence-as-removal member folds, and broader replay safety rests on a documented, asserted orchestration invariant (all outstanding intents stay in the resolved set while the origin is pinned). A host that violates the invariant can still construct reversion scenarios; hardening this seam mechanically is future work.

The campaign is incomplete. The 10^{-9} ship gate has not been reached on the hardened oracle; the prior 94.9M-case run is void (Section 6.3), and the clean-round counter stands at zero after round 16. The honest current claim is "565 tests green, 38 adversarial findings fixed and pinned, oracle hardened with mutation-kill self-tests," not "fuzz-proven."

C2 requires runnable suites. The semantic gate certifies nothing without at least one distinguishing test that runs; a vacuous verdict never certifies, but it also catches nothing. Writers that ship no tests degrade C2 to the structural floor plus surfacing. The gate also multiplies gate cost by roughly $N{+}2$ and needs a per-language toolchain at land time; it is implemented and dark, with zero live callers today.

Single-file scope. Chorus reconciles one file. Cross-file coupling (a rename whose uses span files, interface drift between leased files) is owned by the layers above: write-set admission, wave partitioning, the compile-and-test gate on the post-merge tree, and the merge queue's CAS landing.

Deployment status. The engine, concurrent path, and C2 gate ship dark behind three independent fail-closed flags with runtime kill-switches; a flip-everything build with flags baked on (and the kill-switch retained) was produced on 2026-07-04. Live fleet-scale operation, and with it any throughput evaluation against the serializing queue, is the immediate next step.

9 Conclusion

Chorus demonstrates that for fleets of coding agents, the two classic answers to concurrent same-file editing (serialize, or merge text) can be replaced by a third: decompose the file into keyed cells, collect typed intents

against a fixed origin, and make the merged file a pure function of the intent *set*, so that arrival order is unobservable by construction and every ambiguity fails closed into an explicit, writer-attributed conflict. The engine's evaluation is deliberately dominated by attempts to destroy it: sixteen adversarial rounds that found and pinned 38 silent-loss defects, a mutation audit that invalidated our own 94.9-million-case campaign and rebuilt the oracle that let it happen, and a permutation judge that catches any reintroduction of order dependence at specific seeds. We offer the intent-set model, the silent-loss taxonomy, and the panel-plus-probe assurance loop as reusable pieces for anyone building merge infrastructure for the many-writer era.

References

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering*, 23(4):2051–2085, 2018. doi: 10.1007/s10664-017-9586-1.

[2] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*, pages 190–200. ACM, 2011. doi: 10.1145/2025113.2025141.

[3] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 120–129. ACM, 2012. doi: 10.1145/2351676.2351694.

[4] bors-ng developers. bors-ng: A merge bot for GitHub pull requests. Software. URL `https://github.com/bors-ng/bors-ng`. `https://bors.tech/`. Accessed 2026-07-04.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*, pages 168–178. ACM, 2011. doi: 10.1145/2025113.2025139.

[6] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):59:1–59:27, 2017. doi: 10.1145/3133883.

[7] Codice Software. SemanticMerge: The diff and merge tool that understands C#, Java, and C. Software, 2013. URL `https://www.semanticmerge.com/`. Language-aware merge tool by the makers of Plastic SCM. Accessed 2026-07-04.

[8] Antonin Delpeuch and contributors. Mergiraf: A syntax-aware merge driver for Git. Software, 2024. URL `https://mergiraf.org/`. Source code at `https://codeberg.org/mergiraf/mergiraf`. Accessed 2026-07-04.

[9] GitHub, Inc. Managing a merge queue. GitHub Docs. URL `https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/configuring-pull-request-merges/managing-a-merge-queue`. Accessed 2026-07-04.

[10] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989. doi: 10.1145/65979.65980.

[11] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL `https://openreview.net/forum?id=VTF8yNQM66`.

[12] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of Diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007)*, volume 4855 of *Lecture Notes in Computer Science*, pages 485–496, Berlin, Heidelberg, 2007. Springer. doi: 10.1007/978-3-540-77050-3_40.

[13] Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017. doi: 10.1109/TPDS.2017.2697382.

[14] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. A highly-

available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, 2022. doi: 10.1109/TPDS.2021.3118603.

[15] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. Spork: Structured merge for Java with formatting preservation. *IEEE Transactions on Software Engineering*, 49(1):64–83, 2023. doi: 10.1109/TSE.2022.3143766.

[16] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE 5)*, pages 78–87. ACM, 1992. doi: 10.1145/142868.143753.

[17] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A CRDT for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2), 2022. doi: 10.1145/3555644.

[18] Sergey Pugachev. CodeCRDT: Observation-driven coordination for multi-agent LLM code generation, 2025. URL `https://arxiv.org/abs/2510.18893`. arXiv preprint.

[19] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3): 354–368, 2011. doi: 10.1016/j.jpdc.2010.12.006.

[20] David Roundy. Darcs: Distributed version management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*, pages 1–4, Tallinn, Estonia, 2005. ACM. doi: 10.1145/1088348.1088349.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Berlin, Heidelberg, 2011. Springer. doi: 10.1007/978-3-642-24550-3_29.

[22] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):165:1–165:29, 2018. doi: 10.1145/3276535.

[23] The Pijul developers. Pijul: A distributed version control system based on a sound theory of patches. Software. URL `https://pijul.org/`. Patch theory and merge model described at `https://pijul.org/manual/theory.html`. Accessed 2026-07-04.

[24] Matthew Weidner and Martin Kleppmann. The art of the fugue: Minimizing interleaving in collaborative text editing. *IEEE Transactions on Parallel and Distributed Systems*, 36(11):2425–2437, 2025. doi: 10.1109/TPDS.2025.3611880. Preprint: arXiv:2305.00583.

[25] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, pages 1592–1604, Vienna, Austria, 2024. ACM. doi: 10.1145/3650212.3680384.