Coordination After Storage: Git, Agent Concurrency, and the Boundary of Admissibility

Flyxion
Independent Researcher

November 6, 2026

Abstract

GitHub's Rubicon architecture lets very large numbers of agents write to a single Git repository by dividing its state into immutable objects that need no agreement, a small replicated region of references that does, and disposable compute that needs neither. This design secures the persistence and atomic acceptance of candidate changes. It does not, and was not built to, establish that independently produced changes are compatible and worth accepting, a problem that grows as the cost of producing changes falls. The argument separates five predicates that are easily conflated (durability, connectivity, commitment, validation, and admissibility), shows that the architecture is designed to secure only the first three, and examines where contention migrates once independent writes become cheap, including a proof that independently admissible changes need not compose. It also considers what physical compaction does and does not preserve, in contrast with the semantic approximations common in language model context compaction.

1 Introduction

The headline figures in the announcement of GitHub's new storage architecture, which the company calls Rubicon, are easy to repeat and easy to misread. A repository that accepts 7,200 pushes per second while 5,000 agents work on it has solved a storage problem. It has not, by that fact alone, solved the problem of producing a coherent piece of software from the work of 5,000 agents. The distance between those two statements is the subject of what follows.

The discussion is organized around five predicates that coverage of infrastructure announcements tends to blur, namely durability, connectivity, commitment, validation, and admissibility, which are defined in Section 5. The mechanical sections that precede it should be read as an account of what secures the first three. The architecture itself rests on an observation that Git has always made available and that earlier hosting designs did not fully exploit as a boundary of distribution. Git objects are immutable and named by their contents, so two

writers can store distinct objects at the same time without agreeing on anything. References are mutable, small, and contested, and a reference update is the one part of a push that requires agreement [1]. Rubicon takes that distinction and makes it the principal seam of the system. Objects and packs go to Azure Blob Storage, references and the pack manifest go to a replicated compare-and-swap service called Ref-CAS, and request serving and maintenance run on disposable compute that holds nothing authoritative. The design continues a series in which earlier posts set out the two principles of minimizing coordination and decoupling storage from compute [3] and described the migration of nearly all repositories to Azure, with per-reference quorums and virtual replicas [2].

The interest of the design for anyone thinking about agents is that it makes a particular kind of work cheap, namely the production and publication of candidate changes, and leaves visible what remains expensive. The sections below describe the mechanism, state precisely what it guarantees, and then turn to what it leaves open.

2 Three Regions of State

Traditional Git hosting couples several responsibilities: receiving objects, validating their connectivity, updating references, maintaining packfiles, and serving reads. Each can be optimized separately, but they compete for the same machines. Rubicon separates them into three regions that scale independently.

The first region is immutable object storage. Packs, their indexes, and read accelerators such as multi-pack indexes and reachability bitmaps are written once to distinct keys and never modified, which is the access pattern at which object stores are strongest. Because identity is derived from content, concurrent writers need not coordinate.

The second region is Ref-CAS, a replicated log that holds references and the pack manifest. A transaction locks only the references it modifies, checks that each still holds the value the writer expects, and becomes visible when a majority of the log's replicas have recorded it. Each commit receives a log sequence number, and the repository version is the number of the latest commit. A reader presents the version it already holds, and if that version is current the reply carries no references, so an unchanged repository costs a single comparison [1].

The third region is the fleet of virtual replicas and the independent compaction and garbage collection workers. A replica's local disk is a cache that can be rebuilt, and compaction runs against durable storage directly instead of on request-serving machines.

The ordering inside a push matters more than any individual component. A replica indexes the incoming pack and uploads it to Blob Storage, and only then commits one transaction to Ref-CAS that lists the pack in the manifest and moves the references. Written as a sequence,

$$\text{construct objects} \longrightarrow \text{persist objects} \longrightarrow \text{commit references}.$$

A failure between the second and third steps leaves unreferenced objects behind, which later collection can reclaim. The reverse failure, a committed reference to an object that was never stored, cannot occur if the order is respected. The asymmetry is the whole point: unreferenced durable information is a cost, while an authoritative reference to nonexistent information is a corruption.

3 Why the Reference Layer Looks the Way It Does

The authors describe two earlier prototypes that kept reference state in Blob Storage, and the reasons they failed are instructive because they exhibit the trade-off that Ref-CAS removes. A single write-ahead log indexed by one blob made reads cheap, since a replica need only ask whether the blob had changed, but only one conditional write to a blob can be in flight at once, which held write throughput to the mid-hundreds per second. Splitting references across several segment blobs raised the write ceiling, because writes to different segments no longer waited on each other, but a read then had to inspect every segment and was only as fast as the slowest. Since reads outnumber writes by orders of magnitude, the second design traded the scarce resource for the plentiful one [1].

Ref-CAS replaces the choice between a global serialization point and a partitioned read path with a localized conflict relation. Let T_i be a transaction and $R(T_i)$ the set of references it modifies. Two transactions can proceed without reference-level contention when

$$R(T_i) \cap R(T_j) = \varnothing.$$

When the sets intersect, compare-and-swap ensures that an update based on a stale expected value cannot silently overwrite a competing one, and only one of the two appends to the log. The condition is sufficient to remove direct conflicts on references and no more. Transactions also update the pack manifest, and the published account does not give enough detail to derive the full concurrency protocol for those updates, so independence is claimed here only at the level of references. The manifest is shared state that every push updates, so disjoint reference sets do not imply disjoint manifest entries, and the account does not say whether manifest writes are per-pack additions, which would compose, or per-transaction updates, which would contend.

It would be a mistake to say that global ordering has been eliminated. Every committed transaction receives a log sequence number, so the repository still has a single total order of commits. What has been reduced is the amount of work that must be serialized behind that order, particularly for independent reference updates. The published description speaks of ordering each reference independently while also exposing a repository-wide version. Since the repository version is a position in the log, the total order is definitional, and what per-reference

treatment buys is the ability to check conflicts locally without serializing unrelated work. That arrangement is affordable because the log is small and carries pointers, never object data. RefCAS stores no objects and validates none [1]. This is what permits the reported figure of tens of thousands of reference operations per second on the reference state alone.

Compare-and-swap as a coordination primitive has a long pedigree. Herlihy showed that it is universal for wait-free synchronization [4], and quorum-replicated logs of the kind the authors describe belong to a family exemplified by Raft [5], although the published account names no particular consensus algorithm and nothing here implies which one GitHub uses. The novelty here is not the primitive but the decision to confine it to a region small enough that its cost does not dominate.

4 Compaction as the Second Achievement

Every push adds a pack. At 7,200 pushes per second, with each push introducing its own pack and nothing consolidating them, a repository would accumulate $7200 \times 86400 = 622{,}080{,}000$ packs per day. Supporting a high write rate without a comparable maintenance rate would only move the bottleneck forward in time, and lookup cost grows with pack count even when storage is cheap.

Rubicon moves this work off the serving path. Compaction nodes claim shards of a repository’s packs through leases held in Blob Storage, so one node compacts a shard at a time. Each pack’s hash maps to a shard, which means adding shards and nodes adds compaction throughput. A node gathers new packs from the manifest into an open segment, seals the segment when it reaches a limit on count, size, or age, and merges its packs into fewer larger ones. Before publishing, it checks that the output contains exactly the objects of the inputs. It then adds the new packs to the manifest, removes the inputs in a second transaction, and leaves deletion to later cleanup. Compacted packs below the size limit re-enter a shard by their own hash and merge again, so packs grow in tiers, and the structure resembles a log-structured merge tree [6] applied to a content-addressed object graph.

The invariant that makes this safe can be stated as set equality,

$$\mathrm{Objects}(P_{\mathrm{input}}) = \mathrm{Objects}(P_{\mathrm{output}}),$$

and it is set equality, not byte equality of packfiles. Git objects may be reorganized and recompressed without changing identity. The architecture also separates compaction, which preserves objects while changing their physical representation, from garbage collection, which changes which objects remain retained subject to reachability. Keeping those two operations distinct is what allows each to be audited against a single obvious condition.

Set equality establishes that two representations are logically equivalent. It does not by

itself establish that replacing one with the other is safe under concurrent access, which additionally requires a publication and retirement protocol that keeps readers from losing access to objects while the manifest changes. The account describes adding the replacement packs to the manifest in one transaction and removing the inputs in a second, which creates an interval in which both representations are published and so provides a basis for a safe transition. The complete concurrency and reclamation argument is not supplied, and object-set equivalence should not be mistaken for it.

5 Five Predicates That Storage Cannot Collapse

A large part of the confusion around announcements of this kind comes from using one word, such as “accepted” or “valid,” for several distinct conditions. The architecture makes it possible to separate them cleanly. For a change t consisting of a set of new objects $O(t)$ and a reference transition, define the following.

Definition 1. $D(t)$ holds when every object in the incoming pack $O(t)$ associated with the transaction is durable in object storage. Whether each of those objects is needed by the proposed references is a separate matter, which $N(t)$ addresses. $N(t)$ holds when the objects reachable from the proposed reference values are all present, either in $O(t)$ or already stored. $C(t)$ holds when the reference transaction has been committed by a quorum of the log under its expected-value conditions. $V(t)$ holds when t passes the validation procedures the repository requires, such as protections and continuous integration. $A(t \mid s)$ holds when t is semantically admissible relative to a repository state s, meaning that accepting it at s is correct with respect to what the software is supposed to do. Likewise $V(t \mid s)$ holds when t passes the validation run against s. Both are written $A(t)$ and $V(t)$ where the state is clear from context.

The published write protocol is designed so that $C(t) \Rightarrow D(t)$, since the pack is uploaded before the reference transaction is issued, and the connectivity check performed during indexing, which the authors identify, in the requirements section of the same post, as among the most CPU-expensive parts of a write [1], is intended to support $C(t) \Rightarrow N(t)$. Both implications are relative to the protocol. They hold if the protocol is followed. Connectivity checking at push time establishes N only relative to the object set visible at that moment, so the second implication holds modulo a reclamation-safety invariant, namely that no object reachable from a committed reference is ever reclaimed, which the maintenance machinery must establish and which Section 7 identifies as unresolved in the public record. They are intended safety invariants of the described design and not results that the engineering account proves, and they are treated as such here. No comparable implication holds toward the remaining two:

$$C(t) \not\Rightarrow V(t), \qquad V(t) \not\Rightarrow A(t).$$

A change can be committed, durable, and connected while failing its tests, and it can pass every test while remaining wrong in a way the tests do not express. Compare-and-swap establishes that a reference transition satisfied its expected-state condition. It says nothing about whether the transition was a good idea.

These predicates are connected by protocol-dependent implications and do not form a linear chain, since validation may precede commitment when repository protections require it. The separation is not a criticism of the design, since a storage layer should not be asked to decide A. It does determine what the benchmark can be taken to show. The result concerns D, N, and C under high concurrency, and it should be cited as a result about those three.

6 Where Contention Goes

The performance of Ref-CAS depends on most updates touching different references. The authors note that agents mostly push to their own branches, so most updates do not contend. That is the favorable case, and it is the case in which the concurrency number is least informative about integration, because disjoint branches have not yet been combined.

Consider what happens when k agents all target one protected integration reference. Each push is a compare-and-swap against the value the agent last read. In any round, one succeeds and the remaining $k - 1$ observe a stale expected value. Under an adversarial model of synchronized retries, in which each round begins with every remaining agent obtaining the current reference value and submitting a competing update based on that same value, so that exactly one succeeds per round, the number of attempts needed to land all k changes is

$$k + (k - 1) + \cdots + 1 = \frac{k(k + 1)}{2},$$

which is quadratic in the number of contenders. The model also assumes that each successful update leaves the remaining proposals meaningful and capable of being rebased or regenerated, which is not guaranteed for arbitrary software changes. It counts attempts, not computational work, elapsed time, or network round trips, and it is not a prediction of Rubicon's behavior: an implementation that detects stale expectations early, serializes integration requests, or lets agents abandon obsolete proposals can behave quite differently. Contention for one reference can be mitigated by scheduling, whereas the semantic cost of integration discussed below cannot in general be. The relevant distinction is between delaying attempts and refreshing their expected state: backoff alone does not reduce stale attempts if the remaining participants resubmit proposals based on the same obsolete value, whereas a queue or coordinated scheduler that evaluates each proposal against current state does. The model does identify the shape of the problem. The cost does not disappear when storage becomes fast. It moves to the one reference everyone wants and to the validation and integration processes around it. A merge

queue serializes integration deliberately, and the throughput of that queue then depends on validation and integration costs, including the available validation parallelism, and not simply on reference-update throughput.

There is a second and more important sense in which independent branches defer the difficulty. Disjoint write sets $R(T_i) \cap R(T_j) = \varnothing$ guarantee that two transactions do not conflict at the reference level. They do not guarantee that the changes compose. Two agents may each produce a change satisfying A on its own branch, while their combination violates an invariant neither touched, such as an interface one removed and the other began to use. Admissibility is relative to the state against which a change is evaluated. Let s be a repository state and write $t(s)$ for the result of applying a transition t where it is defined, so that $t_2 \circ t_1$ applied at s means $t_2(t_1(s))$.

Proposition 1 (Non-composition)**.** *There exist a state s and transitions t_1, t_2 such that $A(t_1 \mid s)$ and $A(t_2 \mid s)$ hold while $A(t_2 \circ t_1 \mid s)$ fails.*

Proof. Take states $s = (x, y) \in \{0, 1\}^2$, where $x = 1$ means that a function f exists and $y = 1$ means that a caller of f exists. Let the requirement be the invariant $I(x, y) \equiv (y \Rightarrow x)$, that every caller resolves. For the purposes of this witness, take $A(t \mid s)$ to mean $I(t(s))$, a choice that is the most favorable one for composition, as the discussion after the proof explains. Take $s_0 = (1, 0)$, $t_1(x, y) = (0, y)$ and $t_2(x, y) = (x, 1)$. Then $I(t_1(s_0)) = I(0, 0)$ and $I(t_2(s_0)) = I(1, 1)$ both hold, so $A(t_1 \mid s_0)$ and $A(t_2 \mid s_0)$ hold. But $t_2(t_1(s_0)) = (0, 1)$ and $I(0, 1)$ is false, so $A(t_2 \circ t_1 \mid s_0)$ fails. □

The proposition is existential, and the model is a witness: even when admissibility is expressible as a state predicate, which is the most favorable case for composition, pointwise admissibility does not suffice. The cause is that t_2 was evaluated against s and not against $t_1(s)$. Validation results are conditional on the state in which validation occurred: $V(t_1 \mid s)$ and $V(t_2 \mid s)$ may both hold even though t_2 is never validated against $t_1(s)$, the state in which it will actually be applied. This is the feature of parallel agent development that scheduling cannot remove. The two transitions write different coordinates, one deleting the function and the other adding a caller, so the failure is not attributable to overlapping writes, and they could sit on different branches with disjoint reference sets, so nothing in the storage layer would register a conflict.

A positive counterpart is available when an invariant can be stated.

Proposition 2. *Let I be a predicate on repository states, and suppose each of t_1 and t_2 preserves I at every state in its domain, meaning every state at which that transition is defined. If $I(s)$ holds and $t_2 \circ t_1$ is defined at s, then $I\big(t_2(t_1(s))\big)$ holds.*

Proof. Since $I(s)$ holds and t_1 preserves I, $I(t_1(s))$ holds. Definedness of the composition places $t_1(s)$ in the domain of t_2, and t_2 preserves I there, so $I(t_2(t_1(s)))$ holds. □

The hypothesis is much stronger than observing that both changes passed tests against one shared starting state. Pointwise validation at s constrains behavior at a single state, whereas invariant preservation constrains it over a domain, and only the latter composes. In the example above, t_1 preserves the requirement that calls resolve at s but not at states where f has callers, which is exactly why the hypothesis of the second proposition fails. Taken together, the results say the following. Pointwise admissibility of two transitions against a common initial state does not guarantee admissibility of their composition. Preservation of a specified invariant over the relevant transition domains is sufficient to guarantee that the invariant survives defined composition, and whether this amounts to full semantic admissibility depends on whether the invariant exhausts the requirements that admissibility represents. Absence of a demonstrated invariant does not show that composition fails, only that the available premises do not guarantee it. Nor does the positive result make the problem tractable, since the invariant may be difficult or impossible to discover or check automatically for arbitrary programs, and no general procedure is claimed to decide admissibility. This is a stronger reason for an enduring integration bottleneck than the observation that merge queues take time.

Text-level merging, which Git performs well, detects overlapping edits. Semantic incompatibility between non-overlapping edits is a different phenomenon and is not detected by any mechanism in the storage layer. The benchmark measures storage concurrency and cannot, by construction, measure semantic compatibility among changes.

7 Open Questions in the Published Account

Several properties are not established by what has been published, and the authors are candid that some belong to future posts.

The aggregate push rate is reported without a characterization of push size, object counts per push, branch contention, or validation cost. Throughput of small independent pushes need not predict throughput for large, heavily interdependent agent-generated changes, where indexing and connectivity checking dominate.

The interaction among compaction, garbage collection, and concurrent reference updates needs a careful correctness argument. Leases coordinate who performs maintenance, but a lease does not by itself prove that reclamation is safe. A complete argument must state when an object may be deleted without invalidating an in-flight read, a recently committed reference, or a lagging replica. The authors say garbage collection has special behaviors for safely removing data and defer the discussion, so this question is open in the public record, not answered. It is also the invariant on which the connectivity implication of Section 5 depends, so the two discussions are one problem seen from the write side and the maintenance side.

A monotonically increasing log sequence number makes stale-cache detection cheap, but it does not by itself fix every read-consistency guarantee a client might expect. The roadmap

includes global replicas whose reads remain within bounded staleness of the latest write, and bounded staleness requires visibility semantics to be stated precisely, including what a client that has just pushed is guaranteed to see on a different replica.

Finally, the planned atomic commit API, which would land a multi-file change in a single operation that is either committed or cleanly rejected on conflict, is a step toward the transactional treatment of C at the level of the user's intent. It still concerns acceptance of a transition and not its semantic merit.

8 Physical Preservation and the Record of Decisions

The compaction guarantee deserves separate attention because it is unusually strong. When compaction merges packs and verifies that the output holds the same objects as the input, it preserves a precisely defined set. This is a far stronger preservation condition than the approximations that are common when the context of a language model agent is compacted into a summary, where the relation between the original and the compacted form is semantic, lossy, and usually unverifiable.

The strength is bounded by what the guarantee ranges over. It preserves the Git object set. It does not preserve operational facts around a push: the failed attempts that preceded it, the intermediate reasoning of the agent that produced it, or the external validation evidence that justified accepting it. The object graph does preserve some provenance, since commits carry authorship, parent relationships, messages, and optionally cryptographic signatures. The sharper claim is that it preserves only the provenance and justification explicitly encoded in its objects, not the decision process that produced them. A commit message may state a rationale but is not a reproducible validation record, and a signature authenticates a statement without establishing that the statement is correct. A system that preserves the artifact exactly while discarding the unencoded remainder retains the artifact without the full grounds for trusting it, and as agents author a growing share of changes, those grounds become more valuable and not less.

This is a place where the distinction between working state and evidentiary record is concrete. Compaction of packs is safe because the pack is a representation of an object set and the object set is the thing of record. Nothing analogous makes a summary safe as a replacement for the trace that produced a change. The Rubicon design is therefore best read as a model of what faithful compaction requires, namely a stated equivalence relation, verified before the inputs are retired, together with the honest acknowledgment that the relation covers only what it was defined over.

9 Conclusion

Rubicon makes independent writes cheap by concentrating coordination at narrow, explicit boundaries: content-addressed objects need none, references need a compare-and-swap in a small replicated log, and everything else is disposable. The design honors an ordering, durable objects before authoritative references, that makes one class of corruption unrepresentable, and it keeps maintenance from competing with service.

What it does not do, and was not built to do, is establish that thousands of independently produced changes are mutually compatible, causally attributable, reproducible, and worth accepting. Those are properties V and A, and the cost of establishing them does not fall when the cost of producing candidates falls. If anything it rises in relative terms, because the supply of candidates grows while the capacity to adjudicate them does not. Scalable software development in the presence of many agents therefore appears to require three independently specified guarantees: persistence of candidate artifacts, atomic acceptance of state transitions, and verification of their semantic consequences. The published architecture substantially redesigns the first two. The third is where the next bottleneck lies, and it is a problem of a different kind from the ones storage engineering has so far solved.

The same gap appears in verification, where evidence must be bound to the exact state evaluated: a successful result against an earlier state neither establishes admissibility after integration nor supplies a durable record of why the resulting state was accepted. Preserving the code without preserving that evidence retains the artifact and loses the grounds for accepting it. Physical compaction does not discard such evidence, but it does not guarantee its preservation either, because its guarantee ranges over the object set alone.

References

[1] Karthik Raman and Brian Celenza. Inside GitHub's New Git Storage Architecture. GitHub Blog, Engineering, Architecture and Optimization, November 5, 2026. The most recent post in a series with the two entries below. `https://github.blog/engineering/architecture-optimization/inside-githubs-new-git-storage-architecture/`.

[2] Eleanor White and Victoria Dye. The Bridge to Agent-Scale Git Infrastructure. GitHub Blog, Architecture and Optimization, 2026. Predecessor post in the same series. `https://github.blog/engineering/architecture-optimization/the-bridge-to-agent-scale-git-infrastructure/`.

[3] Brian Celenza. Building Git Infrastructure for Agent-Scale Development. GitHub Blog, 2026. First post in the same series. `https://github.blog/engineering/architecture-optimization/building-git-infrastructure-for-agent-scale-development/`.

[4] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[5] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, pages 305–319, 2014.

[6] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.