GrokRxiv:2026.10.serverless-git-on-cloudflare [cs.DC] 20 Oct 2026

A Cloud-Portable Architecture for Serverless Git

Part III 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

Abstract

Source code and source control can now be treated as transactional abstractions: a repository is a database, a change is a transaction with read and write sets, and landing is a commit under optimistic concurrency. This Part measures what that transaction system costs on cloud primitives and where its serialized authority saturates. The architecture needs an object store, one serialized authority per repository, a durable queue, and stateless and container compute; the reference realization runs on Cloudflare. In six bursts of 1,000 writers on one repository, batched landing, which appends changesets to the log in one authority call per batch and builds Git objects afterwards, landed every change with landing p95 of 1.2 to 3.0 s and refused no commit, landing or read. Commit p95 was under 1 s in two of six runs, main's Git commit lagged by 3.2 to 5.4 s at p95, and no sustained run exists. Landing modes that built Git trees on the landing path landed 4 and 385 of 1,000 on a controlled deploy. A closed queueing model over primitive service times reproduces earlier commit tails within 18 percent and bounds the measured landing rates. The evidence comes from one account, two clients and few runs per revision.

1 Introduction

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, a version is a snapshot, and derived data are views kept consistent with a snapshot. This series lays foundations for that view. This Part supplies the systems layer: it builds the transaction system on cloud-portable primitives and asks what its serialized authority costs and where it saturates.

We argue that the next evolution in code is transactional, and that data went through the same change 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 [1], with partitioned computation over those snapshots [2]. Source control is now under the same pressure. Coding agents are always online, they write small changes against a head that moves while they work, and there can be thousands of them per repository. Anthropic's account of Scion reports that merges per day grew from about 2,500 in May 2026 to more than 30,000 in September 2026, and that source control became the largest bottleneck in development [3]. Scion's published design separates commit creation from ref updates, lands changesets through a write-ahead log, and keeps a ref compare-and-swap on one database row [3], which is the move data

made. This Part supplies one piece of evidence for the argument: what such a transaction system costs on cloud primitives, measured on one realization.

Part I of this series [4] formalizes a design in the same family as a Delta-style transaction log over Git-compatible objects [1]: a hash-chained log of ref moves, checkpoints, snapshot reads pinned to a log version, and landing by backward validation against declared read and write sets [5]. Part II [6] treats blame, search, symbols and codemods as partitioned jobs over immutable snapshots. Both Parts state their results over abstract primitives, and neither says what the design costs in a deployment. This Part states the primitives' contracts, maps them onto Cloudflare's serverless products as the reference realization, separates the limits of those products from the limits of the architecture, reports what was measured, and builds a queueing model over primitive service times that extrapolates the measurements to larger fleets of agents.

The author designed and built the system under study, YonedaRepo; terms such as "authority", "push queue" and "land combiner" are this project's names, not industry usage. The implementation is in Rust (compiled to WebAssembly) with a thin TypeScript adapter, and every claim about it in this paper is tied to a file and commit of its repository in Appendix A [7].

The principal result is measured. The landing redesign, merged into the development branch (fef94e7, merge f9d9c14), appends path-disjoint changesets to the log in one authority call per batch of up to 256 landings and builds main's Git objects in the background. In each of six bursts of 1,000 writers on one repository it landed all 1,000 changes, refused no commit, landing or read, and kept landing p95 between 1.21 s and 3.03 s. Commit p95 was under 1 s in two of the three runs in which each writer had its own token, and in none of the three in which writers shared one login. On an earlier controlled deploy, the two landing modes that built Git trees and merge commits on the landing path landed 4 and 385 of 1,000 changes under the same burst, the first refused as overloaded and the second timed out. The redesign has costs and gaps: main's Git commit arrived 3.2 to 5.4 s after landing at p95, reads succeed only because they are served from a cached head rather than the authority, and the measured landing rates are averages over bursts of a few seconds, not sustained rates.

The path to that result includes a negative one. In the early revisions, commit acceptance p95 was between 2.8 s and 3.7 s across four runs with 100 concurrent writers, against a design target of 1 s. Per-phase timings attribute the miss to an unloaded commit path of about 1.5 s, more than half of it an R2 write, and to queueing under the burst at the serialized authority, a Durable Object in this realization. Later committed revisions took those costs off the acknowledged path, and in 28 of 39 runs commit p95 was then under 1 s, with a weaker meaning of acceptance.

The second result is a capacity model over the primitives' service times. It reproduces the early 100-writer commit tails within 18 percent, using parameters taken from local and per-phase timings rather than from the tails themselves. It projects that the land combiner's cycle sets each branch's ceiling, and that configurations in which waiting landings poll the single-threaded authority collapse past that point. The controlled runs confirmed the second projection in direction, and showed that the realization refuses requests rather than letting them queue; the measured batch call of the redesign bounds its landing rate as the model's batch-capacity relation says it should. Section 8 sets each projection next to the measurement that tests it.

2 System model

A repository in this series has four components: immutable objects, workspace changesets (commits that no protected ref points at yet), protected refs, and stateless compute [4]. The protected refs are governed by a hash-chained transaction log. Each log entry moves one ref and records the paths the

change touched; the snapshot at version v is the ref map that replaying the first v entries produces; and checkpoints let a reader start a replay part way. A writer lands a change by submitting a proposal $P = (r, b, o, W, R)$: a ref, the base version it read, the ref's expected old value, and declared write and read sets. The authority validates the proposal backwards against every entry since b and answers fast-forward, rebase, merge or conflict. It then applies the result as one atomic append that moves the ref and writes the entry in a single transaction. Part I defines these objects (its Definitions 4.6 to 4.16) and proves their properties; this Part uses them without change.

The land combiner is the part of Part I's design that this Part measures most closely (Part I, Definition 7.1). It queues landings per branch, lets one Worker at a time hold a lease with an epoch, builds one octopus merge commit for a batch $B = (t_1, \ldots, t_k)$ of at most 64 queued changes (128 from revision 03821e0), and lands the batch as one log entry. Part I proves that the combiner preserves the single-writer discipline of the append (its Theorem 7.2) and that members of one batch can exhibit write skew with each other (its Theorem 7.3). The batch size therefore has a semantic cost as well as the latency cost studied here.

Three of Part I's invariants (its Table 3) matter on this platform. I4 says that a ref move, its log entry, the projection target, any checkpoint and the landing record commit in one transaction with no network input or output. I6 says that a commit is recorded only after the object store's readback matches what the authority built, and that a hidden ref keeps it reachable. I13 says that branch refs in the object store follow the authority, are never forced, and report a refused update as drift.

Derived data enter this Part only through their deployment. Part II defines a job as a snapshot, a partitioning, a task kind and a policy, and shows that values keyed by content need recomputation only for changed partitions (its Definition 3.3 and Theorem 5.3) [6]. Here derive jobs run from a queue after landing, their segments are stored in R2, and the commit path waits for one derivation step whose cost appears in Table 3.

Two further facts from the companion paper frame the capacity question [8]. Its independence condition (its Eq. 1) decides which work items may run in parallel, and its critical-path bound (its Eq. 2) says that a set of work items needs at least the larger of total work divided by the number of workers and the longest dependent chain. A mandatory serial resource is a chain that every operation passes through. The capacity model of Section 6 is a quantitative version of that observation for the serial resources of one repository.

3 Primitives and the reference realization

The architecture is defined over six primitives, and only their contracts enter Parts I and II. Table 1 states each contract, the Cloudflare product that realizes it in YonedaRepo, the limit of that product that binds the design, and one other product whose documentation offers the same contract. The table makes no performance claim about the other products. Scion's ref update, a compare-and-swap on one Spanner row, is evidence that the serialized-authority primitive is available on another cloud [3, 9]. Every measurement in this paper comes from the Cloudflare realization, on an isolated deployment with its own Artifacts namespace, R2 bucket, D1 database and queues, so no measurement shares state with any other workload.

Three Cloudflare products appear in the reference realization without being primitives. The Cache API is an edge tier in front of the object store; D1 holds a rebuildable index of repositories and is on no measured path; and Basin, Cloudflare's Iceberg-based analytics service, receives a change-log stream on an unmerged branch, off the commit and landing paths [24]. The push queue of Section 4 is a fourth realization-specific component, needed because Artifacts accepts one receive-pack at a time; in later revisions it serves only protected-ref projection.

Table 1: The six primitives, the contract each must meet, its Cloudflare realization with the limit that binds it, and another documented realization. Limits marked as observed are not documented.

Primitive	Contract	Cloudflare realization and binding limit	Another realization
Object store	Immutable, content-addressed puts and gets	R2 as the shared tier and Artifacts for Git objects, with one write per key per second on R2 [10] and 1 GB per Artifacts repository [11]	Amazon S3 with conditional writes [12]
Serialized authority	Durable state per key, with linearizable compare-and-set on refs and log append	One Durable Object with SQLite per repository, single-threaded, with a soft limit of 1,000 requests per second and 200 to 500 for complex work [13, 14]	A Spanner row per ref [9], or DynamoDB conditional writes [15]
Durable queue and workflow	At-least-once delivery with retries	Queues at 5,000 messages per second per queue [16], and Workflows at 300 creations per second per account [17]	Amazon SQS [18]
Stateless compute	Request handlers and map tasks	Workers, with six outgoing connections and 10,000 subrequests per invocation [19]	AWS Lambda [20]
Container compute	Checks and heavy builds	Containers, whose standard-1 type has half a virtual CPU and 4 GiB and no documented cold start [21]	AWS Fargate [22]
Git projection	Serves the Git protocol from the log, optionally	Artifacts, which accepts one receive-pack per repository at a time (observed)	Any Git server fed by an outbox [23]

3.1 The serialized authority

The serialized authority must hold durable state per key and execute a compare-and-set on a ref together with a log append, linearizably. In the reference realization each repository has exactly one Durable Object of class `SourceRepository`, addressed by the repository's name. It holds the transaction log, the refs table, checkpoints every 64 versions, the commit records, the landing queue and lease, approvals, check runs and access grants, all in its SQLite database. A Durable Object is single-threaded and its storage is co-located with its code [25]. SQLite writes inside it are synchronous, a transaction is all-or-nothing, and an output gate holds a response until the write it depends on is durable [26]. Cloudflare's description of the storage layer states that a write is confirmed once at least three of five followers acknowledge it [27]. These properties give Part I's atomic append its meaning on this platform: the compare-and-swap on a ref, the log entry and the landing record are one SQLite transaction, and no other request to the repository can observe a state between them.

From 5bac6b3 on, commit records leave the authority: each commit is recorded with its staged objects in one of 16 commit-shard Durable Objects chosen by the commit's identifier (64 from fef94e7), and the authority adopts a commit the first time an operation needs it. This is the per-workspace direction of the design notes, realized with shards keyed by commit rather than by workspace.

The price of the design is that the authority remains a serial resource for its whole repository. A Spanner row in Scion holds one ref [3, 9]; the authority here holds every ref, every commit record and every landing of the repository. Requests to two branches of one repository queue behind each other, while requests to two repositories do not interact. The architecture needs a total order only for each repository's log; that commit preparation and reads share the same single thread is a

property of this realization.

3.2 The object store and the Git projection

The object-store primitive needs only immutable puts and gets keyed by content. Objects are real Git blobs, trees and commits whose identifiers are the SHA-1 of the type, length and content [28]. In the early revisions measured here, Artifacts is both the Git projection and the authoritative object store. Its Workers binding exposes reads of commits, trees and blobs and the minting of tokens, but no object write or ref compare-and-swap, so the platform writes objects by speaking Git's receive-pack protocol over HTTP [11, 23]. Every recorded commit is kept reachable by a hidden ref named after the commit, and branch refs in Artifacts follow the authority through an outbox with an expected old value, which is Part I's invariant I13. People clone the Artifacts repository read-only with a short-lived token; agents never clone.

The architecture treats the Git projection as optional and asynchronous. The early revisions measured here did not: every commit was pushed to Artifacts and read back before it was acknowledged. Later revisions made R2 the store of record, with a projector that drains staged objects to R2 and pushes only protected refs to Artifacts. The distinction matters for capacity because it decides whether a push is on the path that acknowledges a commit.

3.3 Read tiers

Reads of immutable objects pass through three tiers: the Cache API at the serving location, then R2, then the authoritative store. Each tier asks only the next one, a miss fills the tiers above it, and keys carry the repository so that cached bytes are reachable only through that repository's authorization. Because an object never changes, no tier needs invalidation; ref values are never part of an object cache key. This is the same structure as Scion's daemon, cache and server tiers [3], with two differences. There is no per-machine daemon tier in the platform, and the edge tier is per location, so a miss at one location does not fill another.

3.4 Asynchronous and heavy work

Work that need not finish before a response goes to the durable queue: deriving search segments and symbols after a landing, projecting branch refs to Artifacts, and dispatching check runs. None of it is on the commit or landing path, so the queue's limits do not enter the capacity model. The same holds for imports, which run as Workflows.

Checks are the exception that matters. They run on container compute, Containers in this realization, on the exact commit under test, and a landing cannot proceed until its required checks pass. A run on a commit near a branch head starts from a warm-start seed in R2 and fetches only the blobs that differ, which is the platform's counterpart of Scion's daemon starting from a refreshed snapshot [3].

3.5 Request paths and serialization points

Definition 3.1 (Serialization point)**.** A serialization point is a resource that serves at most one operation at a time for a given key, such as a repository or a branch. Its service time s is the time for which one operation holds the resource. A serialization point that serves a batch of m operations in one service is a batch serialization point with service time $T(m)$.

Definition 3.2 (Commit path and land path)**.** A commit request in the early revisions reads the repository's status for authorization (two or three reads per request in the measured runs) and runs

five stages in order: (1) it reads a snapshot of the target ref from the authority and answers the authority's requests for the trees on changed paths in one or more prepare calls; (2) it starts writing the new objects to R2 and the edge cache, and at the same time pushes a single-ref receive-pack through the push queue; (3) once both have finished, it reads the commit back from Artifacts; (4) it records the commit at the authority and dispatches check runs; and (5) it classifies review classes and derives symbol writes in parallel, recording the review classes at the authority, and the symbol writes too when the changed files contain symbols. A land request first attempts a direct compare-and-swap at the authority. If the branch needs a merge, it enqueues the change and then polls its landing's status and the branch's lease until a batch lands it; a request whose claim on the lease succeeds builds and lands the next batch itself.

The later revisions shorten both paths. A commit makes one authority call, which authorizes it and names its parent, is built in the Worker, and is acknowledged once its commit shard holds it. A landing adopts the commit into the authority and lands or enqueues it. At 4407515 a waiting landing polls the lease with back-off; at 03821e0 it waits on the land combiner instead.

In the early revisions, the measured sequential commits made ten round trips to the authority in an idle repository: four prepare calls, three status reads, one snapshot, one record and one review-class record. The burst's commits made seven, because they add a file one directory below the root and need one tree round where the sequential edits need three; the extra status read of the sequential commits is unexplained. The text files of the benchmark contain no symbols, so no symbol-write call appears in either case. The visits of one commit to the authority are sequential.

Four serialization points lie on these paths: the repository authority, keyed by repository; the push queue together with Artifacts' receive-pack, keyed by Artifacts repository; the land combiner's lease, keyed by branch; and any R2 key written more than once per second. In the later revisions the push queue leaves the commit path, and the 16 commit shards add a serialization point keyed by shard. Everything else on the paths is stateless or partitioned by object: Workers isolates, the compute that builds trees and merges, R2 writes to distinct keys, and edge cache writes.

4 Realization limits

Every limit in this section belongs to a product of the reference realization, not to a primitive. Each subsection names the primitive property involved and what another realization would change.

4.1 The single-threaded authority

Cloudflare documents a soft limit of about 1,000 requests per second for one Durable Object, and advises that complex operations with storage writes sustain 200 to 500 requests per second [13, 14]. The authority executes each command atomically and serially (Part I, Assumption 4.14), and its landing transaction does no network input or output (invariant I4), so its service time is local computation and SQLite work. The service time is not measured inside the object on the hosted platform: the Server-Timing instrumentation measures waits in the Worker, and Workers clocks advance only across input and output, so a Durable Object call appears as the round trip including network and queueing. Section 6 therefore estimates the service time indirectly.

The primitive asks only for linearizable compare-and-set per key. A single thread per repository is one way to provide it, and it makes the whole repository one key. A realization with one row per ref, as in Scion, would serialize ref moves per ref instead, but the log of Part I still needs a total order per repository, so such a realization would need one log per branch or a separate sequencer. This paper analyzes neither.

A second limit is storage. An object's SQLite database is limited to 10 GB on the paid plan [13], and the log has no retention or compaction protocol yet. The log of a repository with Scion's merge rate would grow by tens of thousands of entries per day, so log compaction is a precondition for long-lived repositories on this design, independent of throughput.

4.2 The receive-pack limit

The first hosted test with 100 concurrent commits failed 91 of them with the Artifacts error `ERR artifacts_git_receive_pack_busy`. Artifacts accepts one receive-pack per repository at a time. This limit does not appear in the documentation pages consulted; it is consistent with Cloudflare's statement that each Artifacts repository is its own Durable Object [29]. The constraint belongs to Artifacts as the Git projection, not to the architecture. The object-store primitive accepts concurrent puts of distinct keys, and with the projection made asynchronous, as the architecture allows, no push would lie on the commit path. In the early revisions the projection was authoritative, and the platform's response was a push queue.

Definition 4.1 (Push queue)**.** The push queue of an Artifacts repository is a Durable Object that holds the single-ref receive-pack bodies waiting to be sent. While one combined request is in flight, new bodies wait. When it completes, the queue takes the oldest waiting bodies, at most one per ref, up to $m_{\max} = 64$ bodies and 24 MiB, combines them into one receive-pack whose pack holds the union of their objects and whose commands are their single commands, sends it, and splits Git's report so that each caller receives the status of its own ref. A busy answer is retried up to six times.

After the queue was added, 50 of 50 concurrent commits succeeded, and the 100-writer runs of Section 5 carried 47 to 63 commits in one combined receive-pack. The correctness of combining rests on two properties of the Git protocol and of the platform's pushes.

Proposition 4.2 (Combined push is equivalent to separate pushes)**.** *Let $p_1, \ldots, p_m$ be single-ref receive-pack requests with pairwise distinct refs $r_1, \ldots, r_m$, each of whose packs contains every object reachable from its new value that the repository lacks. Suppose the server applies the commands of a receive-pack without the atomic capability independently, deciding each command from its ref's old value, its new value and the presence of the new value's objects, with no hook or quota that couples one ref's update to another's. If the combined request is unpacked successfully, the status reported for r_i equals the status that p_i would receive if sent alone against the same ref state, for every i.*

Proof. The combined pack contains every object of every p_i, deduplicated by identifier. Identifiers are content hashes, so deduplication removes only byte-identical objects, and every object that p_i needs, which by hypothesis its own pack supplies, is present. The extra objects of other requests can only add to what the repository holds, and the hypothesis already makes p_i complete without them. The server's decision for the command of p_i depends on the old value of r_i, the new value, and the presence of the new value's objects. The refs are distinct, so no other command in the batch changes the old value of r_i, and without the atomic capability the failure of another command does not change the outcome of this one [23]. Each input to the decision is therefore the same as when p_i is sent alone. □

The hypotheses of a successful unpack and of independent commands both matter: a repository-wide quota such as the 1 GB limit would couple the requests of one batch. A malformed or oversized body in a batch makes the whole batch fail, so combining introduces fate sharing that separate pushes do not have. The platform bounds the damage by refusing duplicate refs, capping the batch at 24 MiB, and letting every caller retry. In the measured workload every push creates a fresh

hidden ref with an all-zero old value, so the distinctness hypothesis holds by construction, and the authority builds each pack from every new object of its commit, so the completeness hypothesis holds as well.

The push queue removes the failures but keeps the serialization. A combined receive-pack took 409 to 440 ms to send under the burst and 121 ms for a single commit. The push queue is a batch serialization point in the sense of Definition 3.1, and its capacity is $m/T(m)$ commits per second, which Proposition 6.11 relates to the batch size.

4.3 Object-store write limits

R2 is strongly consistent and limits concurrent writes to one key to one per second [10, 30]. Content-addressed objects have distinct keys, so object writes of different commits do not contend unless two commits write a byte-identical object. Yet in one instrumented 100-writer run, 7 of 100 commits failed with R2 error 10058, "Reduce your concurrent request rate for the same object." Some key is therefore written by many commits. Reading the committed commit path does not reveal it: every object the commit path writes to R2 is keyed by its content hash, and the burst's writers produce distinct blobs, trees and commits. The build that ran H4 came from a working tree whose exact contents were not recorded, so the key may lie in uncommitted code. Until it is identified, this limit is a potential ceiling of one commit per second for whatever path writes that key, and the projections of Section 8 assume that the key can be removed from the commit path. The per-key rate and the latency of R2 puts, which Section 5 shows to have dominated an unloaded commit in the early revisions, are properties of this object store. The primitive's contract says nothing about latency, and another store would have its own, which this paper does not measure.

4.4 Worker connections and subrequests

A Worker invocation keeps at most six outgoing connections open and may make 10,000 subrequests on the paid plan [19]. These are limits of the stateless-compute realization. They do not bind a one-file commit, but they bound bulk operations. Creating a 3,000-file repository took 183 s while every object was written through the R2 and edge tiers, and 6.9 s once large commits wrote only their commit object through; a 10,000-file creation exceeded the subrequest limit before that change and took 11.6 s after it. An earlier change that wrote the cache in bounded chunks rather than all at once cut a 5,000-file import from 129 s to 16 s.

4.5 Other limits

Artifacts limits a repository to 1 GB and a blob to 32 MB [11]. Scion's main branch history is almost 100 GB [3], so a repository of that size cannot be hosted in one Artifacts repository. That limit of the Git projection, not throughput, is the first obstacle to a Scion-sized monorepo in this realization, and it is one reason later revisions made R2 the store of record. Queues and Workflows have limits far above the rates projected here (Table 1). Containers have no documented cold-start time; Section 5 reports the measured one.

5 Measured evaluation

5.1 Environment and workloads

All hosted measurements use the isolated platform-only Worker on Cloudflare's paid Workers plan, with real Artifacts, Durable Objects, R2 and Cache API. Two clients drove the benchmarks. The

live benchmark ran on one developer machine, so its latencies include that machine's network path. The fan-out benchmark ran in a client Worker placed in the AWS us-east-1 region, which can drive a thousand writers; it reports client-side latency and the server-side Server-Timing total separately. Latencies from the two clients are not comparable as if they came from one client.

The live benchmark seeds a repository with 100 files, starts 100 writers at once, each of which commits one new file on `main` and then lands it, and then reads the 100 new blobs concurrently. It then times sequential one-file edits on a small and a large repository and compares a `git clone` of the Artifacts projection with a first read through the API. The fan-out benchmark runs the same commit-then-land cycle per writer at 20, 100 or 1,000 writers. Percentiles are computed over the successful requests of one run, so a p95 over 100 requests is set by the five slowest. The deployed tree's cleanliness is not recorded by the hosted benchmarks, so each run is attributed to the revision its output names.

The code changed between runs, and the runs fall into three groups. The early group (H1 to H4) ran revisions from 535d9ab to the Server-Timing instrumentation of ebbee09, in which every commit made seven authority calls under load, wrote its objects to R2 and pushed them to Artifacts before it was acknowledged. From 734c8a9 on, a small commit is acknowledged once a Durable Object holds its objects, and from 5bac6b3 that object is one of 16 commit shards per repository; R2 later became the store of record and Artifacts receives only protected-ref projections. The later group ran these revisions with polling landers, and the last group ran 03821e0, which replaced polling with a land combiner that waiting landings do not poll.

5.2 Concurrent writers across revisions

Table 2 gives every hosted burst run that the paper uses.

The early group missed the commit target. Commit acceptance p95 was between 2.8 s and 3.7 s across four runs, against a design target of 1 s with 100 concurrent writers, and every commit and landing succeeded only in H1 and H2. H3 lost 9 commits to an input-validation error whose message the run did not keep, and H4 lost 7 to the R2 per-key limit. The benchmark's inputs are fixed, so the H3 failures arose from server-side state under load rather than from the inputs, but the cause is unknown. The deployed revision of H3 and H4 was not recorded, and the two runs were not repeated for this paper.

The later group met the commit target, with a changed meaning of acceptance. In 28 of 39 runs on twelve committed revisions from 6dac346 to 4407515, commit p95 was under 1 s, ranging from 437 ms to 2,356 ms; every one of the 39 runs committed and landed all 100 writers. On 4407515 all four runs fell between 437 ms and 753 ms client-side, and between 198 ms and 606 ms server-side. An accepted commit in these revisions is durable in a Durable Object, the authority's staged copy or a commit shard, but not yet in R2 or in Artifacts. That is a different durability point from the early group's, and Section 11 treats it as such. The outputs of these runs exist as uncommitted files only.

Landing did not improve with the non-polling combiner. At 100 writers from the same us-east-1 client, the polling path landed with p95 3.65 s on the run attributed to a1766af, while the combiner at 03821e0 landed with p95 25.7 to 30.1 s. The run on the polling path survives only as a tool result in a session transcript, and its revision is inferred, so the comparison rests on one run on each side.

At 1,000 writers, both landing paths overloaded the repository authority. With polling landers at a1766af, every commit succeeded but only 20 to 306 of 1,000 landings did; in one run 980 landings were refused with the platform's error "Durable Object is overloaded. Requests queued for too long." With the combiner at 03821e0, 990 commits succeeded, 288 landings succeeded, 547 were refused as overloaded, 155 timed out, and the landing p95 was 105 s. Until the redesigned landing below, these were the only hosted runs in which a serialization point visibly saturated, and Section 7 compares

Table 2: Hosted burst runs on one repository. Latencies are client-side p95 in milliseconds over successful requests. Revisions marked inferred are not recorded in the output. Runs S1 to S6 come from one deploy of revision dd6c9a8 and differ only in landing mode; runs B1 to B8 come from one deploy of fef94e7, where tokens means one token per writer and login means one shared login, and their entries are ranges over the runs listed.

Run	Revision	Client	Landing	Writers	Commits	Commit p95	Landing p95
H1	535d9ab	developer	polling	100	100	3,731	6,722
H2	778f76a	developer	polling	100	100	2,802	4,976
H3	ebbee09 build	developer	polling	100	91	3,691	7,745
H4	ebbee09 build	developer	polling	100	93	3,316	7,505
L1 to L4	4407515	developer	polling	100	100	437 to 753	4,234 to 5,423
F1	a1766af, inferred	us-east-1	polling	100	100	465	3,650
F2	03821e0	us-east-1	combiner	100	100	638	30,089
F3	03821e0	us-east-1	combiner	100	100	805	25,671
F4	a1766af	us-east-1	polling	1,000	1,000	2,945	43,063
F5	a1766af	us-east-1	polling	1,000	1,000	2,177	63,077
F6	a1766af	us-east-1	polling	1,000	1,000	1,801	24,033
F7	03821e0	us-east-1	combiner	1,000	990	2,643	105,496
S1	dd6c9a8	us-east-1	polling	100	100	255	2,389
S2	dd6c9a8	us-east-1	combiner	100	100	1,220	13,028
S3	dd6c9a8	us-east-1	changeset	100	100	653	1,202
S4	dd6c9a8	us-east-1	polling	1,000	1,000	2,221	104,034
S5	dd6c9a8	us-east-1	combiner	1,000	1,000	2,704	64,756
S6	dd6c9a8	us-east-1	changeset	1,000	1,000	1,698	18,728
B1 to B3	fef94e7	us-east-1	batch, tokens	1,000	1,000	714 to 1,673	1,212 to 1,865
B4 to B6	fef94e7	us-east-1	batch, login	1,000	1,000	1,140 to 1,614	2,167 to 3,027
B7, B8	fef94e7	us-east-1	batch	100	100	263, 328	956, 568

them with the model.

5.3 Batched landing on the development branch, measured

The landing redesign, merged into the development branch (fef94e7, merge f9d9c14), lands a burst of 1,000 writers without refusals. In every one of six 1,000-writer runs on a deploy of fef94e7, whose deployed platform was a clean build of that revision while the bench client carried an uncommitted change that reports commits per shard, all 1,000 changes landed, no commit or landing was refused as overloaded, and all 1,000 reads returned the correct content. The measured mode, configured as `batch`, submits every waiting landing on a branch to the authority in one call of up to 256 landings. Path-disjoint landings append changesets to the log, landings on the same files are queued for the merge path, and a member whose reads or writes meet an earlier member's writes waits for the next batch. Commits are recorded in 64 commit shards, between 7 and 30 per shard in these runs, and reads of main are served from the published head that the Worker caches rather than from the authority.

The results depend on whether each writer has its own token or all writers share one login, and Table 2 gives both sets as ranges over three runs each. With separate tokens, commit p95 was 714 to 1,673 ms, under 1 s in two of the three runs, and landing p95 was 1.21 to 1.87 s. With one shared login, commit p95 was 1,140 to 1,614 ms, missing 1 s in all three runs, and landing p95 was 2.17 to 3.03 s; the shared login's authorization phase alone took 381 ms at the median in the run that

kept its detail. At 100 writers both runs met every target, with commit p95 of 263 and 328 ms and landing p95 of 956 and 568 ms.

The rate is a burst rate, and no sustained run exists. Over the span from the first landing request to the last answer, the runs landed 404 to 491 changes per second with separate tokens and 235 to 284 with one login; each burst lasted 2.7 s and 4.3 s end to end in the runs that recorded it. A burst that short says nothing about the rate the authority can hold, which Section 9 plans to measure.

The delay before main's Git commit exists fell with the redesign but did not vanish. Its p95 was 3.5 to 4.6 s with separate tokens and 3.2 to 5.4 s with one login at 1,000 writers, and 2.7 and 2.4 s at 100 writers. A reader that needs main's Git commit, such as a person cloning it, waits that long; a reader of the log or of the published head does not.

The reads succeed because they avoid the authority, which is a trade-off rather than a removal of the limit. On an earlier revision of the same branch, the same mode with reads served by the authority returned 540 of 1,000 reads correctly and refused 460 as overloaded, and a changeset mode that also began commits at the authority refused all 1,000 reads. The authority still cannot serve a 1,000-writer burst of commits, landings and reads at once; the design moves the reads elsewhere.

Two qualifications apply to all six runs. Only the third run of each set kept its full output; the first two survive as one-line summaries that the benchmark script printed from their output before overwriting it. The workload writes one new file per writer and declares no reads, so it never exercises the within-batch write-skew check, which fef94e7 contains (Part I, Proposition 7.4).

5.4 Controlled comparison of landing modes

Because fef94e7 removed the older landing modes, the controlled comparison comes from an earlier deploy, dd6c9a8 on branch `claude/native-p95e`, in which the benchmark chose the landing mode per run on one deploy from one client (runs S1 to S6 in Table 2). In its changeset mode a landing whose paths are disjoint from everything landed on its branch since its base appends its changeset to the log with one indexed query, without building Git objects or a merge commit, and a deterministic projector builds the branch's Git commit afterwards.

At 1,000 writers on that deploy, the changeset mode landed all 1,000 changes with a landing p95 of 18.7 s and a commit p95 of 1,698 ms, and no commit or landing was refused. On the same deploy the polling mode landed 4 of 1,000, with 994 refused as overloaded and a landing p95 of 104.0 s, and the combiner landed 385 of 1,000, with 615 timeouts and a landing p95 of 64.8 s. At 100 writers the changeset mode landed all 100 with a landing p95 of 1.2 s, against 2.4 s with polling and 13.0 s with the combiner. The landings that built Git trees and merge commits on the landing path failed under the 1,000-writer burst; the landing that appends changesets and builds Git objects in the background did not. That deploy's changeset mode still had the costs that fef94e7 later reduced: a burst-span rate of 49.7 landings per second, a Git projection delay p95 of 17.3 s, and 422 of 1,000 reads refused as overloaded, which the benchmark's overload count omits because it covers commits and landings only. The deploy was attributed to dd6c9a8 from the session record; two of the six output files name a later commit because the checkout moved during the runs, and no deploy happened in between.

5.5 Commit time by phase

Table 3 gives the median of each phase in the two instrumented runs of the early group, for the burst and for the 12 sequential commits measured afterwards in the same run. A phase value is the sum of the waits of its calls within one request, and the median is taken over requests. Phases inside the parallel classification and derivation step overlap, and the cache write overlaps the push,

Table 3: Median phase waits per commit request in milliseconds for the two instrumented runs. Burst columns are the 100-writer commits; sequential columns are the 12 one-file commits after the burst. Calls per request are given where a phase makes several authority calls.

Phase	H3 sequential	H3 burst	H4 sequential	H4 burst
Authority snapshot (1 call)	45	202	45	153
Authority prepare (4 calls sequential, 2 burst)	214	313	215	470
Authority status (3 calls sequential, 2 burst)	145	348	146	400
Authority record (1 call)	74	89	77	115
Authority review classes (1 call, inside derive)	74	107	75	118
Cache write (R2 and edge)	859	1,141	892	1,157
Push queue round trip (wait and send)	170	790	169	620
Readback from Artifacts	105	246	113	219
Derive (classes and symbols in parallel)	126	240	129	248
Tree reads (3 rounds sequential, 1 burst)	9	30	8	20
Push step as a whole	962	1,456	1,005	1,442
Commit acceptance, measured	1,543	2,820	1,632	3,059

so the phases do not add to the total. The critical path does add up. For a sequential commit in H4 it is snapshot, prepare and status calls (406 ms), push including the wait for the cache write and the readback (1,005 ms), record (77 ms), derive (129 ms) and tree reads (8 ms), a total of 1,625 ms against a measured median of 1,632 ms.

The R2 write was the largest single cost of an unloaded commit in the early group. The push step waits for the cache write before it reads the commit back, and the R2 part of that write took 892 ms at the median in sequential H4 commits, while the edge part took 4 ms. The push queue's own round trip, 169 ms, finished long before the cache write did, so in an idle repository the push to Artifacts was hidden under the R2 write. An unloaded commit spent about 55 percent of its time waiting for R2, about a third in ten authority round trips of roughly 50 ms each, and the rest in readback and derivation.

The burst added most of its delay at the authority. Between sequential and burst commits in H4, the summed waits of the four authority phases before the derive step grew from 483 ms to 1,138 ms at the median, although the burst made fewer calls. Per call, the early-path calls (snapshot, prepare, status) grew from about 50 ms to 150 to 235 ms, while the record call, which arrived after the push had spread the commits out in time, grew only from 77 ms to 115 ms. This pattern is what a single-threaded server shows when 100 requests arrive together and fewer arrive later.

At the tail, the push queue could join the critical path. In the burst its round trip had a p95 of 1,070 to 1,271 ms against a cache-write p95 of 1,411 to 1,433 ms, and a fresh isolate also had to mint a write token, whose p95 was 512 to 575 ms. A commit that needed a new token and waited behind a full batch could therefore spend longer in the push than in the cache write. The median wait behind an in-flight combined push was 45 to 183 ms, and its p95, 407 to 419 ms, is about one send time, which is the behavior Definition 4.1 predicts.

The later revisions removed exactly the costs that the table identifies. On 4407515 a burst commit makes one authority call, which authorizes it and names its parent; the Worker builds the commit and records it with its objects in a commit shard. Its server-side median was 222 ms, of which the authority call took 116 ms and the shard record 50 ms; a sequential commit took 145 ms, with an authority call of 20 ms. The R2 write and the Artifacts push no longer delay the acknowledgment, which is the source of both the speedup and the change in what acceptance means.

5.6 Landing under a burst

Landing p95 under the burst ranged from 5.0 s to 7.7 s in the early group, while the direct compare-and-swap call to the authority took 86 to 87 ms at the median. Most of the landing time was waiting on the land combiner. A landing request made 13.6 to 16.0 lease claims and 14.6 to 17.0 status reads on average, summing to 2.5 to 2.6 s and 2.6 to 2.9 s of waiting respectively. Three to four batch landings served each burst of 91 to 100 landings, which puts the mean batch near 25 to 33 changes if every landing went through a batch. The change-log stream recorded the same pattern on later revisions: across 18 repositories with 101 accepted changes each, a 100-landing burst became two to four octopus batches.

The landing request that held the lease spent about 2.4 to 2.6 s on one cycle: building and pushing the octopus merge commit, recording it, and the batch landing call itself (487 to 579 ms). A landing that arrives during a cycle waits for it and then for its own, and a landing in the last of three or four batches waits for more. The measured landing p95 of 5.0 to 7.7 s corresponds to two to three such cycles.

Polling was a property of the adapter's landing loop in the Worker, not of the combiner's design, and it dominated the authority's load: on average an early-group landing made about 30 authority calls, of which 28 were polls. Revision 4407515 folded the status read into the claim and backed off between polls from 20 ms to 320 ms, and a landing then made 6.5 to 8.2 claims on average.

The non-polling combiner at 03821e0 holds waiting landings in memory and lands batches of up to 128 in steps driven by Durable Object alarms. Two of its alarm steps recorded wall times of 10.5 s and 12.5 s during the 20-writer and 1,000-writer runs, and the developers' message attributes the long steps to reading every fresh tree and blob serially through five storage layers. Only the two wall times are measured; the cause is the developers' reading of the code path. A step of that length, together with waits that the landing loop renews in rounds of 20 s, is consistent with the measured 100-writer landing p95 of 25.7 to 30.1 s, though that consistency is an inference.

5.7 Reads, commit cost, clones and checks

Warm reads met their own target of p95 under 1 s in every run (Table 4), and the edge tier served every warm read. The one measurement that reached R2 came from the reads of a cold Container check: R2 served 121 of 123 reads, a hit rate of 0.984, below the platform's own target of 99 percent per tier, which it adopted from Scion's published design. The run's revision was not recorded, and it is a single run.

Commit cost did not grow with repository size in the early group. In H1 and H2 a one-file commit on the 3,000-file repository was no slower than on the 100-file repository, which is what Part I's bound on the cost of an incremental tree rewrite predicts (its Proposition 4.3). On 4407515 the comparison used a 10,000-file repository and passed in two of four runs; in the last run the medians were 248 ms and 261 ms. The sequential commit median of 1.06 s reported in the commit message of the early latency changes is not confirmed by raw output, which shows 1,302 ms after those changes.

Reading one file through the API was about seven times faster than cloning the repository: a full `git clone` of a small projected repository took 2.0 s, while the first API read of one file took 278 ms at the median. Scion's published design rests on the same contrast between fetching what a job needs and materializing the repository [3], but one 3,000-file repository and one client support no general claim.

A seeded check run started an order of magnitude faster than a cold one in both measured pairs, 7.0 s against 86.4 s and 8.1 s against 109 s. Most of the cold time is starting the container, and

Table 4: Other hosted and local measurements. Values marked as from a commit message have no raw output on disk.

Quantity	Value	Source
Warm blob reads, 100 concurrent, p95	666 to 906 ms in the early group and 443 to 599 ms on 4407515	Raw output
Tier serving the warm reads	Edge 101 of 101 in H1, H2 and the last 4407515 run	Raw output
Reads of a cold Container check, 121-file repository	R2 121, store 2 and edge 0, an R2 hit rate of 0.984, and a seeded run took 120 of 121 files from its seed	Raw output, revision not recorded
Container check wait, cold and from a warm-start seed	86.4 s and 7.0 s in one pair, 109 s and 8.1 s in an earlier pair	Raw output and commit message
One-file commit p50, small vs large repository	1,725 vs 1,627 ms (H1, 3,000 files), 1,302 vs 1,085 ms (H2), 248 vs 261 ms (4407515, 10,000 files)	Raw output
`git clone` of a 3,000-file projection, full and depth 1	2,023 ms and 1,540 ms for a 216,064-byte pack	Raw output
First API read of one blob in the same repository, p50 and p95	278 ms and 370 ms over 10 reads	Raw output
Clone of the Artifacts repository after a landing	Landed commit, content and `git fsck -strict` verified	Commit message
Local workerd, 100 writers, revision ba19372	All 100 landed, with commit p95 303 ms, landing p95 453 ms and read p95 85 ms	Raw output
Local one-file commit p50, 100 vs 10,000 files	5 vs 5 ms (ba19372) and about 13 vs 14 ms earlier	Raw output and commit message

each pair is a single run.

5.8 Local measurements

Local measurements ran in workerd, the open-source Workers runtime, on one developer machine with simulated Durable Object and R2 latency. The one local run with raw output, on revision ba19372 with a clean working tree, landed all 100 writers with commit p95 303 ms, landing p95 453 ms and read p95 85 ms. A one-file commit took about 5 ms at the median whether the repository had 100 or 10,000 files, which shows that the authority's logic is not the cost. These figures are not evidence about hosted latency, because the hosted path is dominated by network round trips that workerd does not reproduce.

A separate local benchmark of the earlier agent-layer authority, a different Durable Object class with a different schema, measured 662 commands per second with four concurrent clients on one repository, 676 per second with 32 clients over 8 repositories, and 646 per second with 64 clients over 16 repositories. Throughput did not rise with the number of repositories, so these figures measure the cost of a command to one workerd process rather than a per-object limit. Section 6 uses the four-client figure as an independent estimate of per-call service time.

6 Capacity model

The measurements describe one repository with 100 writers, but the question that motivates the system concerns thousands. The model below is stated over the primitives and parameterized by

their service times: the authority's service time per call s, the push queue's send time, the land combiner's cycle and its batch size k. The Cloudflare measurements supply one instantiation, and the model can extrapolate them only after it reproduces the 100-writer runs, which Section 7 checks.

6.1 Definitions and assumptions

Definition 6.1 (Capacity model)**.** A repository is a set of serialization points $j = 1, \ldots, J$ (Definition 3.1). One agent cycle consists of a think time Z, one commit and one landing. During one cycle the agent's operations visit point j a mean of c_j times with service time s_j per visit, so the demand of a cycle at point j is $D_j = c_j s_j$. A batch point whose service of k operations takes $T(k)$, with batches capped at $k_{\max}$, has demand $D_j = T(k_{\max})/k_{\max}$ at saturation. The unloaded cycle time R_0 is the time one cycle takes when no other agent is active. With N agents per repository, $X(N)$ is the throughput in landings per second and $R(N)$ is the mean time of a cycle excluding think time.

The landing protocol adds one feature that a plain demand model misses. A landing that finds the branch's lease held does not wait in a queue; its Worker runs a loop that asks the authority for the landing's status and for the lease, sleeps for 5 to 25 ms, and asks again; whichever waiting landing obtains the lease builds the next batch. Each waiting landing is therefore a source of authority calls for as long as it waits, and its iteration time is set mostly by the two round trips.

Definition 6.2 (Poll loop)**.** A waiting landing polls the authority in iterations of two authority calls followed by a short sleep. Its unloaded iteration time τ is the sleep plus two round trips in an idle repository, so the iteration takes $\tau + 2W$ when each call waits W at the authority. Q denotes the number of waiting landings. A landing whose enqueue call is still queued at the authority also has exactly one call outstanding, so it is counted among the Q pollers.

Assumption 6.3 (Closed population)**.** The N agents of a repository form a closed population: each agent waits for its landing before it starts its next think period, and the think time Z is the same for all agents and independent of load.

Assumption 6.4 (Work-conserving points)**.** Each serialization point serves waiting operations in arrival order and is never idle while an operation waits.

Assumption 6.5 (Uniform call cost)**.** Every call to the repository authority has the same service time s, whether it is a poll, a prepare or a batch landing.

Assumption 6.6 (Combiner cycle)**.** One cycle of the land combiner takes a time T_{c0} plus the authority waits of the n_c authority calls it makes, $T_c' = T_{c0} + n_c W$, independent of the batch size k; with T_{c0} set to a cycle measured under load, the waits are counted twice, which makes the ceiling conservative. A landing needs a merge with probability $f_m = 1 - e^{-XL_c}$, the chance that the branch moved while its commit was being made, where L_c is the commit's latency. A landing that needs no merge lands with one direct compare-and-set. A landing that needs a merge waits for its own cycle and, with probability $\rho_c = \min(1, XT_c')$ that the combiner is busy, for half of the current one: $T_c'(1 + \rho_c/2)$ on average.

Assumption 6.3 matches how agents work, since an agent whose change has not landed usually waits or rebases rather than piling up commits. Assumption 6.4 holds for the authority's input queue and the push queue by construction. Assumption 6.5 is the weakest: a poll is cheaper than a batch landing that verifies an octopus merge and inserts k landing records. Assumption 6.6 ignores growth of the merge with k, which Section 9 measures. In a single-agent repository every landing is a fast-forward, so R_0 is about 1.56 s, the commit plus one direct compare-and-set; with many agents nearly every landing needs a merge.

6.2 Bounds

The model uses five results. The first bounds waiting at a work-conserving point when many operations arrive together, which is the situation of the benchmark.

Lemma 6.7 (Burst waiting bound)**.** *Let N jobs share a work-conserving single server. Let each job make c visits one after another, each of service time s, with arbitrary delays between visits spent away from the server. The total time any one job spends waiting for the server is at most $(N-1)cs$.*

Proof. While a job waits for the server, the server is busy (Assumption 6.4). Because the job's visits are sequential, the visit in service then belongs to another job. Each unit of the other jobs' work is served exactly once. The waiting time of the job, summed over its visits, is therefore at most the total work of the other jobs, which is $(N-1)cs$. □

The second result is the standard pair of asymptotic bounds for a closed population, which locates the knee. It is Denning and Buzen's bottleneck analysis [31], also given as the asymptotic bounds of Lazowska et al. [32]; we state it in this paper's notation and do not reprove it.

Proposition 6.8 (Asymptotic bounds [31, Section 4])**.** *Under Assumption 6.3, with $D_{\max} = \max_j D_j$,*

$$X(N) \le \min\Big\{\frac{N}{R_0+Z},\ \frac{1}{D_{\max}}\Big\}, \qquad R(N) \ge \max\Big\{R_0,\ ND_{\max} - Z\Big\}. \tag{1}$$

Corollary 6.9 (Knee [31, Section 4])**.** *The two throughput bounds cross at $N^* = (R_0+Z)/D_{\max}$. Below N^* throughput is bounded by the population, and above it by the bottleneck point, the index j that attains $D_{\max}$.*

The bounds need only job flow balance and visit counts and service times that do not change with load, which are the hypotheses Denning and Buzen state. The model meets the first in steady state. It meets the second only approximately: polling makes the number of authority visits per landing grow with the wait, which is why Corollary 6.12 treats waiting landings separately. The unloaded cycle R_0 here includes network round trips and object-store writes that are delays rather than queueing demands; in Denning and Buzen's terms they are delay centers, and they enter R_0 without entering $D_{\max}$.

The bounds say nothing about latency below the knee. For that the model uses the mean waiting time of a single server with Poisson arrivals and deterministic service.

Proposition 6.10 (Waiting at a deterministic point [33, Sections 7.6 and 7.7])**.** *If operations arrive at a single first-come, first-served server as a Poisson process of rate λ and each needs service time exactly s, with $\rho = \lambda s < 1$, the mean waiting time before service is*

$$W = \frac{\rho\, s}{2(1-\rho)}. \tag{2}$$

This is the Pollaczek-Khinchine mean-value formula $W = \rho\,\mathbb{E}[R]/(1-\rho)$ with mean residual service $\mathbb{E}[R] = \mathbb{E}[S^2]/(2\,\mathbb{E}[S]) = s/2$ for deterministic service; its derivation relies on Poisson arrivals seeing time averages [33].

At the authority one commit makes several visits, so visit arrivals are not Poisson even if commits are, and waiting landings are a closed population rather than an open stream. The model uses Eq. (2) as a mean-value approximation for the open part of the load. Below saturation it shows that queueing is negligible next to the unloaded path, because s is a millisecond or two while R_0 is seconds. Above saturation the polls dominate, and the approximation's solution approaches the bound of Corollary 6.12 (Appendix B).

The fourth result describes batch points.

Proposition 6.11 (Batch capacity)**.** *A batch point whose service of a batch of k operations takes $T(k) = a + bk$ with $a > 0$ and $b \geq 0$ has capacity $k/T(k)$ operations per second. The capacity increases strictly with k and tends to $1/b$ as k grows, or without bound if $b = 0$.*

Proof. The derivative of $k/(a + bk)$ in k is $a/(a + bk)^2 > 0$, and $k/(a + bk) \to 1/b$ as $k \to \infty$ when $b > 0$. □

Batching to amortize a serial commit step is old database practice: DeWitt et al. describe group commit as part of the field's folklore [34], and the Delta Lake paper names batching as the way to raise a log's commit rate past the latency of its atomic step [1]. The early revisions cap both batch points at 64 and the combiner at 03821e0 caps its batches at 128, so the capacity at the cap, $k_{\max}/T(k_{\max})$, is the relevant ceiling.

The fifth result explains why polling matters.

Corollary 6.12 (Poll saturation)**.** *Let each of the Q waiting landings of Definition 6.2 make p authority calls per iteration. Treated as a closed population with think time $\tau - ps$ and demand ps each at the authority, they alone saturate the authority once Q exceeds $Q^* = \tau/(ps)$, and above Q^* one poll iteration takes at least pQs seconds.*

Proof. Apply Proposition 6.8 and Corollary 6.9 to the pollers, with population Q, think time $\tau - ps$, unloaded iteration τ and bottleneck demand $D_{\max} = ps$. The knee is $(\tau - ps + ps)/(ps)$, and the response bound gives an iteration of at least $Q\,ps$. □

Above Q^* every other caller of the authority shares the queue that the polls build. The land combiner's own n_c calls wait in it, so its cycle lengthens, its throughput $64/T_c'$ falls, more landings wait, and Q grows further. In a closed population this feedback has no stable point near the combiner's ceiling, and the mean-value solution of Section 8 shows throughput collapsing once it starts. The corollary gives the threshold; the size of the collapse depends on Assumptions 6.5 and 6.6 and is a model result, not a bound.

6.3 Model inputs

Table 5 lists every input and its source. Three inputs are assumptions with no source: the think time Z, the peak-to-average factor of the merge rate, and the configurations of Section 8 that do not exist yet.

Local and hosted estimates agree that the authority's service time is about 1.5 ms per call. The value used, 1.51 ms, is the inverse of the throughput of one local process that ran both the Worker and an earlier authority class, so it includes some Worker work. The hosted estimate takes the growth of the authority's p95 waits between sequential and burst commits, after removing the three calls that only sequential commits make, and divides it by $(N_{\text{ok}} - 1)c$. It equals s only if the tail commit waits for the full bound of Lemma 6.7, and it gives 1.40 ms for H3 and 1.58 ms for H4.

In the early revisions a waiting landing polled about every 0.11 s in an idle repository: its loop sleeps 5 to 25 ms between iterations, and an idle round trip to the authority takes 45 to 49 ms. In the burst the measured iteration was about four times longer, 0.40 to 0.45 s, the median landing time of 6.3 to 6.6 s divided by 14.1 to 16.5 claim and status pairs. Section 7 uses that difference as a check on Corollary 6.12. At 4407515 one claim per iteration replaced the claim and status pair, and the back-off between 20 ms and 320 ms puts the unloaded iteration near 0.29 s once the back-off has grown, the mean pause of 240 ms plus one round trip; in the burst a landing made 8.2 claims in a median landing of 4.5 s, about 0.55 s per iteration.

One combiner cycle took about 2.6 s in the burst and made nine authority calls: three prepares, two status reads, a record, a review-class record, a claim and the batch landing. The cycle is the sum of the median phases of the request that held the lease, three or four such requests per run, and it was measured while polls loaded the authority, so it includes some queueing. The model uses the measured cycle as it stands; it has no measurement of an unloaded cycle.

The push send time is nearly flat between batches of 47 and 63, which suggests that a combined push costs mostly a fixed overhead. A line through the single-commit send and the larger batches gives a send time at batch 64 between 0.41 and 0.55 s, and therefore a push ceiling of 116 to 156 commits per second by Proposition 6.11.

7 Calibration

The model can be trusted to extrapolate only if it reproduces what was measured. The 100-writer benchmark is a single closed cycle with $Z = 0$ in which every writer starts at once, so Lemma 6.7 applies to its commits. Adding the largest possible authority waiting to the unloaded commit tail gives the predictor

$$\widehat{T}_{95} = \left(L_0^{95} - \delta\right) + \left(N_{\mathrm{ok}} - 1\right) c\, s + \Delta_T. \tag{3}$$

Each term comes from a measurement other than the run's commit p95. The unloaded tail L_0^{95} is the p95 of sequential one-file commits in the same run. Those commits edit a file two directories below the root, while burst commits add a file one directory down, so a sequential commit needs three tree-read rounds instead of one, two more prepare calls and one more status read; $\delta = 161$ ms removes their unloaded cost. The authority term uses the local service time and the number of commits that succeeded, N_{ok}, because failed commits stopped before making all their calls. The tier term Δ_T is the mean growth of the push step's p95 between sequential and burst commits in H3 and H4; it covers the slower R2 write, the token mints of fresh isolates and the wait behind an in-flight combined push. The bound is approached by the commits that finish last, and those are the commits that set the p95. The tier term stays constant in N only while the authority releases commits to the push queue no faster than the queue drains them. Five authority calls precede the push, so the condition is $1/(5s) \le m_{\max}/T_{\mathrm{push}}(64)$; at $s = 1.51$ ms the release rate is 132 commits per second against a drain rate of 145, and the condition holds.

The gate, fixed before any projection was computed, requires the predictor to reproduce each run's measured commit p95 within 25 percent. Table 6 applies it.

The predictor passes the gate on every run, with errors between -17.8 and -0.4 percent. H1 and H2 are out of sample in the strict sense: they have no phase data, and none of the predictor's parameters was estimated from them. H3 and H4 supplied δ and Δ_T from their phases, but not their measured commit p95.

Every error is negative, and three omissions explain the direction. The burst's writers land as soon as their own commits finish, so landing calls and polls from early finishers join the authority's queue while the slowest commits are still waiting, and the predictor counts only commit calls. The p95 of a sum of phases exceeds the sum of the phases' typical values, and the predictor adds an unloaded p95 to bounds and means. And the largest error belongs to H3, whose nine failed commits left no error message and whose maximum commit latency, 5.5 s, is the largest of the four runs.

The measured landing times agree with Assumption 6.6. With the measured cycle of about 2.6 s, a landing that finds the combiner busy waits $1.5\,T_c = 3.9$ s on average. The measured landing median was 3.5 s in H2 and 5.3 to 6.6 s in the other runs, whose bursts needed three to four cycles back to back.

The burst's landing phase also matches Corollary 6.12. About 90 landings waited at once, far

more than the 36 at which polls alone saturate the authority. The measured poll iteration of 0.40 to 0.45 s was four times the unloaded 0.11 s, and the lease holder's calls took about 90 to 155 ms each against 45 to 49 ms in an idle repository. The corollary's bound with two calls per iteration, an iteration of at least $2Qs$, then limits the service time to at most $0.45/(2 \times 90)$, or 2.5 ms, which the value used respects.

The later revisions offer three further checks, none of which was used to set a parameter. On 4407515 a burst commit makes one authority call, and that call's p95 grew from 29 ms in sequential commits to 138 ms in the burst, a growth of 109 ms. Lemma 6.7 bounds the growth at $99 \times 1.51 = 150$ ms, and the measured growth lies within it while implying $s \geq 1.1$ ms. At 1,000 writers the same call's p95 was 1,067 ms with polling landers and 2,113 ms with the combiner. The commit calls alone bound the first at $999 \times 1.51 = 1{,}508$ ms; the second exceeds that bound, which the lemma explains only if landing calls overlap the commits, as they do when early writers adopt and land while late writers still commit. Finally, both 1,000-writer landing paths overloaded the authority. With polling landers about 1,000 landings waited at once, far above the Q^* of about 190 that Corollary 6.12 gives for 4407515's polling loop. With the combiner the polls were gone, but about 3,000 authority calls arrived within seconds by the developers' count, which is about 4.5 s of serial work at 1.51 ms per call; the platform refused requests that had queued too long. That count is arithmetic, not a measurement, and the platform's queueing limit is not documented.

The model is therefore calibrated in the sense the gate requires: it predicts commit tails within 18 percent from parameters that do not use those tails. It is not calibrated for landing throughput at sustained saturation, which no run reached; the failure it predicts for polling configurations has been observed only in bursts, and as refusals (Section 8.4).

8 Projected capacity

Every number in this section, except those labelled measured, is projected, estimated from measured per-operation cost of the Cloudflare realization through the model of Section 6. Another realization would enter the same formulas with its own service times, which this paper does not estimate. The projections describe a steady state in which agents commit and land repeatedly; the measured comparison points are bursts, and Section 8.4 keeps the two apart.

8.1 Workload shape

Scion's published design reports more than 30,000 merges per day, an average of about 0.35 merges per second [3]. The post reports no peak-to-average ratio, no count of agents, no per-agent rate and no landing latency under bursts. This paper therefore assumes a think time Z of 60 s between an agent's landing and its next commit, varied from 10 s to 600 s, and makes no claim that its runs or projections are comparable with Scion's operation.

8.2 Configurations

The three modelled configurations follow the code. Each takes its parameters from Table 5.

C1, early revisions. Seven authority calls per commit, 2.24 per landing besides polls, polls of two calls with an unloaded iteration of 0.11 s, a land combiner with the measured cycle of 2.6 s and batches of up to 64, and an unloaded commit p95 of 1.54 s.

C2, revision 4407515. Commits in shards with one authority call each, an unloaded commit p95 of 345 ms, polls of one call with back-off and an iteration of 0.29 s, and otherwise C1's landing.

C3, revision 03821e0. C2's commit path, a land combiner that waiting landings do not poll, batches of up to 128, and the measured step of about 11.5 s. Each landing costs the authority its 2.24 calls plus one for answering its wait.

The redesigned landing of Sections 5.3 and 5.4 is not modelled as a configuration; Section 8.4 compares its measured batch call with the batch-capacity relation instead. C3 assumes that the authority can hold one waiting call per landing at no service cost beyond answering it. The model is solved as a fixed-point approximation for each N (Appendix B); it is not the exact mean-value analysis of closed networks.

8.3 Ceilings and knees

Table 7 gives the ceiling of each serialization point, the combiner's knee, the peak that a polling configuration reaches before it collapses, the projected throughput and commit latency for 10 to 10,000 agents, and the measured burst rates of the same landing modes.

The land combiner is the ceiling of every modelled configuration, and its cycle sets the ceiling. In C1 and C2 the measured cycle of about 2.6 s with batches of 64 bounds one branch at about 24.6 landings per second. The step of about 11.5 s measured at 03821e0 lowers the bound to about 11 per second even with batches of 128. The authority's ceiling for real work is far higher in every configuration, and the push queue is off the commit path from C2 on. By theorem 6.11, a combiner that lands 128 changes per cycle needs a cycle of at most 2.56 s to reach 50 landings per second.

Polling decides what happens at the knee. In C1, polls of two calls with an iteration of 0.11 s saturate the authority once about 36 landings wait, which happens near 9 landings per second; the lease holder's calls then slow, the combiner's cycle lengthens, and the backlog feeds on itself. C1 is projected to peak near 16.5 landings per second at about 1,125 agents and to collapse beyond that point, to about 0.5 at 10,000 agents. C2's back-off raises the threshold to about 190 waiting landings, so C2 reaches the combiner's ceiling, about 24.5 landings per second near 1,575 agents, and collapses only when the backlog beyond the knee grows past the threshold. In C2 the commits avoid the polls until the collapse, after which every commit's single authority call waits behind them.

C3 does not collapse in the model. Without polls the authority's utilization stays below one fifth at the combiner's ceiling, and throughput levels off at about 11 landings per second from about 860 agents at 60 s think time, while landings wait longer.

8.4 Projections against measurements

The side-by-side runs on dd6c9a8 test the model's qualitative claims, though not its steady-state numbers, because they are bursts of 1,000 writers that all start at once. The model was right that polling fails first. On the same deploy, the polling mode landed 4 of 1,000 changes and the authority refused 994 as overloaded. It was wrong about the form of the failure. Theorem 6.12 describes an unbounded queue whose waits grow without limit; the platform bounds the queue of a Durable Object and refuses requests that wait too long, so saturation appears as refusals rather than as slow answers.

The model was right that the measured combiner cannot reach the rates of the other modes. Its projected ceiling of about 11 landings per second bounds the measured burst rates of 7.0 per second at 100 writers and 5.9 per second at 1,000 writers, where 615 landings timed out after the landing loop's 120 s deadline. The bound is not tight: a burst spends part of its span before the first batch fills and after the last.

The changeset mode of dd6c9a8 measured 49.7 landings per second over the span of the 1,000-writer burst, with all 1,000 landed. The batch-capacity relation says that 49.7 per second with batches of 128 corresponds to one full batch every 2.58 s, close to the 2.56 s that a rate of 50 per second requires; that deploy logged neither batch sizes nor cycle times, so the relation could not be tested there.

The batched landing of fef94e7 logs both, and it gives the relation its one direct test. Its batches held 256 landings, and the authority call that lands a batch took 308 ms at the median and 372 ms at p95 in the run that kept its detail. By theorem 6.11 that call bounds one branch at $256/0.372 \approx 690$ to $256/0.308 \approx 830$ landings per second. The measured burst-span rates of 404 to 491 per second with separate tokens and 235 to 284 with one login lie below that bound, as they must, and the gap is the time a burst spends before its first batch fills and after its last. The relation was right about what limits landing: one serial call per batch, not one per landing. It does not say what rate the branch sustains, because no sustained run exists.

8.5 Commit latency

The commit target is a property of the commit path, not of the landing path. With the early commit path, no configuration is projected to bring commit p95 under 1 s at any fleet size, because the unloaded p95 alone, about 1.5 s, exceeds it. With the sharded commit path of C2 and C3, the unloaded p95 is about 345 ms and queueing at the authority adds little until a polling configuration collapses: one call per commit at 1.51 ms is a small load even at the combiner's ceiling. In C3 commits stay near 0.35 s at every fleet size in the model, with the acceptance semantics of the later revisions. Measured bursts are slower, because landings and reads overlap the commits: at 1,000 writers on fef94e7, commit p95 was 714 to 1,673 ms with separate tokens and 1,140 to 1,614 ms with one login, under 1 s in two of six runs.

8.6 Scale against the Scion workload

The tests are not comparable with Scion's operation: Scion's published design reports a daily average and no burst latency, while these figures are a measured burst-span rate and projected steady-state ceilings for one branch of one repository. With that caveat, the projected ceilings of 11 to 24.6 landings per second are 30 to 70 times the published daily average, and the measured burst-span rates of 235 to 491 per second on fef94e7, over bursts of a few seconds, are 680 to 1,400 times it. The comparison also ignores repository size, which rules out one Artifacts repository for a Scion-sized history, the cost of checks, which this platform runs before landing, and the read traffic of tens of millions of checkouts per day.

8.7 Sensitivity

The authority's per-call cost matters most for the polling configurations. At 1.0 ms the C1 peak rises to about 19.5 landings per second near 1,300 agents; at 5 ms, the slowest rate Cloudflare's guidance allows for complex work, the C1 peak falls to about 7 near 550 agents and the C2 peak to about 12 near 800. C3 holds about 11 landings per second across the whole range of s. The measured step of 10.5 to 12.5 s at 03821e0 puts C3 between about 10.2 and 12.2 landings per second, and the early cycle's range of 2.4 to 2.6 s puts the C1 and C2 ceilings between 24.6 and 26.7. A faster poll iteration in C2 hurts: at 0.15 s its peak falls to about 22.5. The knee scales with $Z + R_0$: at $Z = 10$ s the C1 and C2 peaks fall to about 300 and 350 agents, and at $Z = 600$ s no configuration saturates below 10,000 agents. The ceilings do not depend on Z.

Two effects could make every projection optimistic. The combiner's cycle may grow with k, since a larger batch takes longer to merge and verify. And the unidentified R2 key that failed 7 of 100 commits in H4 could cap commit throughput near one per second on whatever path writes it, unless the later revisions removed it.

8.8 Validation runs

Each projection has a run that would test it, and Section 9 gives the methods. A commit-only burst of 1,000 writers on the early revisions should show a commit p95 between 12.6 s, if the push queue keeps pace with the authority, and 19.1 s, if its 16 sends of about 0.44 s add to the authority's queue; a value outside 9.5 to 23.9 s refutes the model's extrapolation in N. On the sharded commit path the same burst should show the authority call's p95 within $999s$ of its sequential value, about 1.5 s. The C2 projection is refuted if an open-loop sweep of the landing rate on 4407515 peaks below 18 or above 31 landings per second, or if throughput past the peak holds steady while the authority's poll rate rises. The C3 projection is refuted if logged combiner steps at 03821e0 put $128/T_c$ outside 8 to 15 landings per second. The changeset landing is tested by the sustained-rate run of Section 9.

9 Planned evaluation

This section states the planned evaluation for the whole series; Parts I and II cite it for the measurements they lack. Each item gives the method, the metric and the outcome that would confirm or refute the expectation. No results are reported.

Batched landing at a sustained rate. The developers' targets are 100 of 100 landings with landing p95 at most 2 s at 100 writers, which both 100-writer runs on fef94e7 meet, and 1,000 of 1,000 landings at a sustained rate of at least 50 per second at 1,000 writers, which no run has tested. The 1,000-writer runs landed every change at 235 to 491 per second over bursts of a few seconds, which is a burst average, not a sustained rate. Run open-loop arrivals at fixed rates from 50 to 800 landings per second for at least ten minutes each on a revision of the development branch, from one client, and log the batch size and duration of every authority batch call. The target is met if the sustained rate reaches 50 per second with no refused landing or read; the batch log tests the capacity relation directly, and the highest rate held without refusals locates the knee.

Reads under bursts. On fef94e7 every read succeeded because reads of main came from the cached published head; with reads served by the authority, 540 of 1,000 succeeded. Repeat the 1,000-writer burst with each read source, count refused reads with the other refusals, and treat any refused read as a failure of the run, so that the cost of reading from the authority under load is measured rather than avoided.

Commits under a shared login. With one shared login, commit p95 at 1,000 writers missed 1 s in all three runs. Time the authorization phase per request across the burst to test whether the shared session's checks, not the commit path, account for the difference.

Write skew in a batch. The benchmark cannot produce write skew, because each writer writes its own file and declares no reads. Add a workload whose writers read files that other writers in the same batch write, and measure how often the within-batch check of Part I defers a member and what it costs per batch.

From 100 to 1,000 writers. Repeat the hosted burst with 100, 250, 500 and 1,000 writers on one committed revision with Server-Timing enabled, five runs per size from one client location, and record the deployed revision and the cleanliness of its tree in the output. The series begins by repeating the early group's instrumented runs on a recorded revision. Run each size twice: once as today, with every writer landing after its commit, and once with commits only, so that landing calls do not overlap the commit tail. The metric is commit acceptance p95 and the per-phase sums. The commit-only series tests the burst predictor of Eq. (3). A second series replaces the burst with open-loop Poisson arrivals of landings at fixed rates from 2 to 30 per second, recording throughput and the authority's call rate, to locate the C1 plateau and test for the collapse that Corollary 6.12 predicts.

Per-tier hit rates. The edge served every warm read in the hosted benchmarks, and the one measurement that reached R2, the reads of a single cold check, gave an R2 hit rate of 0.984. Measure hit rates per tier against the platform's 99 percent target, taken from Scion's published design, with cold reads from several client locations, reads after an edge eviction, and the reads of many Container check runs on a recorded revision. A tier below 99 percent under a realistic read mix refutes the claim that the tier structure carries that caching property. A local test already shows that 20 concurrent cold misses of one object cause 20 origin reads, so request coalescing is a known gap.

Real-model arms. The companion paper proposes Study A, which crosses three repository layouts with file-level and symbol-level interfaces, and prediction P9, that dependency-aware scheduling reduces makespan and integration failures relative to file-disjoint scheduling [8]. The platform provides the symbol interface, declared read sets and the planner that P9 needs. The planned runs use the companion paper's metrics (closure recall, exposure precision, and cost per accepted change, its Eq. 12) and its cluster-aware inference with repositories as clusters. The scripted evaluation program exists, and the early real-model runs are too small to report.

Service time inside the authority. The model's least certain input is s. Measure it by saturating one authority with a closed loop of each call type from many Workers and applying the utilization law, $X_{\max} = 1/s$, separately for polls, prepares, records and batch landings. This replaces Assumption 6.5 with measured costs. A per-call cost above 5 ms for the common calls would put C1 and C2 outside the sensitivity range of Section 8.

Land combiner batches. Log the batch size and cycle time of every claim, and sweep the landing rate from 1 to 30 per second on one branch. The metric is the distribution of k and the fitted relation $T_c(k)$, which tests Assumption 6.6.

Push sizes. Sweep the combined push from 1 to 64 commits and from small to 24 MiB, and fit the send time as $a + bm$ in the batch size m, with byte size as a second regressor.

Derived data. Part II reports no measurement of segment build time, segment size, the cost of an R2 lookup per partition, the fraction of partitions that change per commit on a real repository, codemod duration or plan duration, and no hosted measurement of the queued derivation job. Measure each on imported public repositories of increasing size by replaying their recent history through the platform, one landing at a time, with the derivation job timed in the queue consumer. The metrics are build time and size per segment, the changed-partition fraction per landing, and

the end-to-end delay from landing to a queryable segment. Part II's claim that cost follows the number of changed partitions (its Proposition 5.6) is refuted if build time per landing grows with repository size at a fixed changed-partition count. A sweep of partition granularity on the same repositories supplies the inputs that Part II's granularity result (its Proposition 11.1) needs.

The shared R2 key. Log the key of every R2 write on the commit path under the burst and identify the one that drew error 10058. The projections assume it can be removed from the commit path; if it cannot, commit throughput is capped near one per second for that path.

10 Deviations from Scion

The platform follows Scion's principles but departs from its architecture in four ways, each with a cost that the measurements make visible.

Git objects instead of changesets. A Scion changeset lists the files changed relative to its first parent and points at content-addressed blobs; it needs no tree objects [3]. The measured revisions build real Git blobs, trees and commits on the commit and landing paths. That was a choice of the realization, not a requirement: what the design needs is Git's semantics of commit, branch, merge, diff and log, and the changeset landing of Section 5.4 builds main's Git objects in the background as a deterministic projection of the log. In the measured revisions each commit writes new tree objects along every changed path. Part I's incremental rewrite keeps the cost proportional to the change (its Proposition 4.3), which the flat one-file commit cost on repositories of 100 and 3,000 files confirms, and in the early revisions each commit wrote those objects to R2 and Artifacts before it was acknowledged. The Scion post credits Mononoke, the server of Meta's Sapling, with pioneering many of Scion's techniques, especially the Bonsai data model of flat file changes [3, 35].

One authority per repository instead of one row per ref. Scion moves a ref by a compare-and-swap on a single Spanner row, separate from changeset creation [3]. Here the compare-and-swap is a transaction in the repository's Durable Object, with the same single-row semantics, but the object holds every ref of the repository and also serves commit preparation and recording. Commit creation therefore passes through the same serial resource as ref moves, which is why the burst's delay appears at the authority and why C2 exists. Scion's ref store grows by rows; this platform grows by sharding repositories, so one hot repository is the unit of capacity.

An approval gate before the main branch. Scion lands without a global merge lock and invests in testing after merge [3]. This platform requires human approval or policy auto-approval and required checks before a landing. The gates are a fixed cost on the land path. In the one measured pair, a Container check took 8.1 s from a warm-start seed and 109 s cold, so a landing that waits for a check waits far longer than the combiner's cycle. The capacity projections exclude check time; they assume gates are already satisfied when a landing is enqueued, as the code requires.

Many repositories instead of one monorepo. The platform hosts many repositories per workspace, each with its own authority and push queue, so the authority's limits apply per repository. The Scion-scale question becomes the capacity of one hot repository, which is what Section 8 projects, and a fleet spread over many repositories does not contend at all.

Two smaller deviations follow from these. Checkpoints are written every 64 versions into the authority's SQLite database rather than to object storage, which is simpler but adds to the 10 GB

limit. And a batch of landings becomes one octopus merge commit that keeps the landed commits' identifiers, whereas Mononoke's pushrebase rewrites commits onto the new head to keep history linear [36].

11 Threats to validity

One account, two clients. All hosted runs used one throwaway Cloudflare account on the paid plan. The live benchmark ran on one developer machine and the fan-out benchmark in one client Worker in us-east-1, and every latency includes its client's network path, and the placement of the Durable Objects relative to the client was not controlled or recorded. Round trips of about 50 ms per authority call in an idle repository suggest that the network path is a first-order term, so latencies from another location could differ substantially.

Few runs and code drift. The early 100-writer burst was run four times, on three code states, and the later runs span more than a dozen revisions with few runs each. The p95 of 100 samples is set by five requests, and the spread of p95 across runs was 0.93 s. Two runs lost 7 and 9 commits, which were excluded from their percentiles; their p95 values would be higher with the failures counted as slow. The deployed revision of the two instrumented early runs was not recorded in their output, the hosted benchmarks never record whether the deployed tree was clean, and the one polling run from the us-east-1 client survives only in a session transcript with an inferred revision. No output of a later run is a committed file; every one is an uncommitted output copied into the project's evidence.

Synthetic workload. Each writer adds one tiny new file to a 100-file repository in a closed burst with no think time. There are no conflicting writes, no merges of the same file, no read-heavy mix, and no open-loop arrivals. Real agents write larger changes, so commits would carry more objects and longer prepares.

Measurement semantics. Server-Timing phases are sums of waits measured in the Worker, not CPU time, and Workers clocks do not advance during computation. A percentile of a sum is not the sum of percentiles, so the decomposition of the median in Section 5 cannot be applied directly to the tail. The batch size per claim was not logged, so the mean batch is inferred from rounded call counts.

Platform behavior. Artifacts is in beta. The one-receive-pack limit is observed, not documented, and could change. The R2 key behind the failures in H4 is unidentified.

Redesigned-landing runs. The fef94e7 results come from one deploy and one client; the deployed platform was a clean build of fef94e7, but the bench client carried an uncommitted change that reports commits per shard. There were three runs per principal mode at 1,000 writers and one per mode at 100; only the third 1,000-writer run of each mode kept its full output, and the first two survive as summary lines. Their rates count from the first landing request to the last answer of a closed burst lasting a few seconds, so they are not sustained rates. The controlled comparison on dd6c9a8 used a branch revision, one run per mode and size, and two of its six output files name a later commit than the deploy, an attribution that rests on the session record.

Acceptance semantics. The commit p95 under 1 s of the later revisions counts a commit as accepted once a Durable Object holds its objects, before R2 or Artifacts does. A commit accepted in this sense survives a Worker failure but depends on the projector to reach the store of record, and a reader of the store of record may not see it yet. The early revisions' figures use the stricter meaning, so the two groups do not measure the same event.

Local evidence. One local run has raw output; the earlier local figures exist only as approximate values in a commit message, and the simulated latencies of workerd do not resemble the hosted network path. The local per-call cost used in Section 7 comes from a different Durable Object class with a different schema.

Model assumptions. The projections rest on landing behavior that no run has observed at sustained saturation. The capacity model treats every authority call as equally expensive, treats agents as a closed population with a fixed think time, takes the combiner's cycle as independent of batch size, and assumes that configuration C2's combiner keeps the early cycle, which was measured under load. Its service time comes from a different authority class measured locally. Its calibration covers commit tails under one burst size. The 1,000-writer bursts did saturate the authority, but they are bursts, not the steady state that the projections describe, and the collapse the model predicts for polling configurations is a fixed-point result whose onset and depth depend on these assumptions. The changeset landing is measured but not modelled.

12 Related systems

The question of how fast a log-structured table can commit has a known answer for object stores. Delta Lake commits by atomically creating the next log entry, and its authors observe that the commit rate is bounded by the latency of that atomic step, which for object-store put-if-absent allows several transactions per second per table; they suggest batching and a low-latency coordination service as remedies [1]. Apache Iceberg commits by an atomic swap of a metadata pointer with optimistic retry [37]. The repository authority plays the role of Delta's low-latency coordinator, and the land combiner is the batching. The measurements here put numbers on both: the atomic step itself takes about 86 ms as a hosted round trip, and the batch cycle around it takes about 2.6 s.

Scion places its ref compare-and-swap in Spanner [3, 9], which scales by distributing rows. Mononoke's pushrebase moves a bookmark by rebasing the pushed commits on the server when they touch different files, retrying on conflict [36]. Part I (its Section 11) compares the concurrency control of these systems in detail; this Part adds the observation that on a serverless platform the cost of a ref move is dominated by the calls around it, not by the move. Google's Piper is the best-documented earlier case of one repository serving a large company [38].

13 Conclusion

The transaction system of Parts I and II needs six primitives, and their contracts are small: content-addressed puts and gets, a linearizable compare-and-set with a log append per repository, at-least-once queues, stateless and container compute, and an optional Git projection. On Cloudflare each maps onto one product. The commit path on that realization missed its 1 s target in early revisions, with p95 between 2.8 s and 3.7 s across four runs, and met it in 28 of 39 later runs once commits were acknowledged when a Durable Object held them.

Landing was the harder problem, and the measurements say where it lies. Landings that built Git trees and merge commits on the critical path failed under a burst of 1,000 writers. Landings that append changesets to the log in batches and project Git objects afterwards, merged into the development branch at fef94e7, landed all 1,000 changes in each of six bursts without a refused commit, landing or read, with landing p95 of 1.21 to 3.03 s. A queueing model over primitive service times reproduces the early commit tails and predicted that polling landings would fail first, which they did, though by refusal rather than by unbounded delay.

What the foundation enables is a source-control system whose correctness rests on a transaction log and whose cost can be stated per primitive before it is built on any particular cloud. What remains open is a sustained landing rate, which no run has measured; reads served by the serialized authority, which it still refuses under a burst; commit p95 under 1 s for writers that share one login; a delay of several seconds before main's Git commit exists; and a workload that exercises write skew within a batch.

References

[1] 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.

[2] 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, San Jose, April 2012. `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.

[3] 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`.

[4] M. Long and The YonedaAI Collaboration. A transaction log for source control. Part I of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[5] 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.

[6] 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.

[7] M. Long. YonedaRepo source repository, branch `dev` at commit `cfa4845`, 20 October 2026, with the branch commits named in Appendix A. Private repository; not publicly available.

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

[9] 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)*, Hollywood, October 2012. `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett`.

[10] Cloudflare. R2 limits. `https://developers.cloudflare.com/r2/platform/limits/`.

[11] Cloudflare. Artifacts documentation: overview, Workers binding, Git protocol and limits. `https://developers.cloudflare.com/artifacts/`.

[12] Amazon Web Services. Add preconditions to S3 operations with conditional requests. Amazon S3 User Guide. https://docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-requests.html.

[13] Cloudflare. Durable Objects limits. https://developers.cloudflare.com/durable-objects/platform/limits/.

[14] Cloudflare. Rules of Durable Objects. https://developers.cloudflare.com/durable-objects/best-practices/rules-of-durable-objects/.

[15] Amazon Web Services. DynamoDB condition expression CLI example. Amazon DynamoDB Developer Guide. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ConditionExpressions.html.

[16] Cloudflare. Queues limits. https://developers.cloudflare.com/queues/platform/limits/.

[17] Cloudflare. Workflows limits. https://developers.cloudflare.com/workflows/reference/limits/.

[18] Amazon Web Services. What is Amazon Simple Queue Service? Amazon SQS Developer Guide. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html.

[19] Cloudflare. Workers limits. https://developers.cloudflare.com/workers/platform/limits/.

[20] Amazon Web Services. What is AWS Lambda? AWS Lambda Developer Guide. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[21] Cloudflare. Containers: Limits and instance types. https://developers.cloudflare.com/containers/platform/limits/.

[22] Amazon Web Services. Architect for AWS Fargate for Amazon ECS. Amazon ECS Developer Guide. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html.

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

[24] M. Selwan and M. Wylde. Introducing Cloudflare Basin: an open, serverless data platform, now generally available. Cloudflare blog, 1 October 2026. https://blog.cloudflare.com/cloudflare-basin/.

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

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

[27] K. Varda. Zero-latency SQLite storage in every Durable Object. Cloudflare blog, 26 September 2024. https://blog.cloudflare.com/sqlite-in-durable-objects/.

[28] S. Chacon and B. Straub. Git internals: Git objects. In *Pro Git*, 2nd edition. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects.

[29] D. Mulroy, M. Carey, and M. Silverlock. Artifacts: versioned storage that speaks Git. Cloudflare blog, 16 April 2026. https://blog.cloudflare.com/artifacts-git-for-agents-beta/.

[30] Cloudflare. R2 consistency model. https://developers.cloudflare.com/r2/reference/consistency/.

[31] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225-261, 1978. doi:10.1145/356733.356735.

[32] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984. https://homes.cs.washington.edu/~lazowska/qsp/.

[33] I. Adan and J. Resing. Queueing systems. Lecture notes, Eindhoven University of Technology, 2015. https://www.win.tue.nl/~iadan/queueing.pdf.

[34] 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*, 1984. doi:10.1145/602259.602261.

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

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

[37] Apache Software Foundation. Apache Iceberg table specification. https://iceberg.apache.org/spec/.

[38] 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.

A Code evidence

Table 8 ties each claim this Part makes about the implementation to a file and commit in the YonedaRepo repository. Commits are on the development branch unless a branch is named.

Table 8: Claims about the implementation and the code that shows each. Paths are relative to the repository root.

Claim	File and commit	What the code shows
One authority per repository	`cloudflare/worker/repo-ops.ts` at cfa4845	`caller` addresses the `SOURCE_REPOS` namespace with `idFromName(repo)`
Commit path order	`cloudflare/worker/repo-ops.ts` at cfa4845	`commit()` calls snapshot, prepare in rounds with tree reads, starts the cache write, pushes, records, dispatches runs, then classifies and derives in parallel
Readback waits for the cache write	`cloudflare/worker/repo-objects.ts` at cfa4845	`pushCommit` awaits its `ready` argument before `readCommit`
Push queue per Artifacts repository	`cloudflare/worker/repo-objects.ts` at cfa4845	`ArtifactsStore.push` addresses `OBJECT_PUSHES` with `idFromName` of the repository
Push queue batching rules	`cloudflare/worker/repo-push-queue.ts` at ee0e9cd	Busy-error comment, `MAX_BATCH = 64`, a 24 MiB cap, six busy retries, and at most one body per ref
Combined receive-pack	`backend/crates/yoneda-core/src/git_object/push_batch.rs` at ee0e9cd	Refuses a ref twice, deduplicates objects by hash, and requests report-status (with delete-refs for deletions) but not the atomic capability
Landing transaction without network calls	`backend/crates/yoneda-platform/src/repo_store/land.rs` at cfa4845	`append` inserts the log entry, upserts the ref, sets the projection target, checkpoints and inserts the landing through SQL calls only
Landing poll loop	`cloudflare/worker/repo-merge.ts` at cfa4845	`landWithMerge` tries a direct land, enqueues on `MERGE_REQUIRED`, then loops over status and claim calls with a sleep of 5 to 25 ms, and builds the batch itself when its claim succeeds

Claim	File and commit	What the code shows
Commit shards	`cloudflare/worker/repo-shards.ts` at 03821e0	`SHARDS = 16`, and `shardOf` picks a shard from the commit's first hex digit
One authority call per sharded commit without check runners	`cloudflare/worker/repo-commit-shard.ts` at 03821e0	The header comment says the authority only authorizes the write and names the parent; `commitInShard` calls `repo_begin_commit` once, and `repo_adopt_commit` only when check runners are configured
Small commits acknowledged when staged	`backend/crates/yoneda-platform/src/repo_store/staging.rs` at 03821e0	`STAGE_LIMIT` of 1 MiB and `STAGE_OBJECTS = 256` bound what is staged
Poll back-off at 4407515	`cloudflare/worker/repo-merge.ts` at 4407515	`WAIT_FIRST_MS = 20`, `WAIT_MAX_MS = 320`, and one `repo_land_claim` per poll
Non-polling land combiner	`cloudflare/worker/repo-land-combiner.ts` at 03821e0	The header comment says landers wait on the combiner instead of polling the authority and that the combiner alone claims the queue
Combiner wait rounds	`cloudflare/worker/repo-merge.ts` at 03821e0	`WAIT_ROUND_MS = 20_000` and `LAND_DEADLINE_MS = 120_000`
Batch of 128 at 03821e0	`backend/crates/yoneda-platform/src/repo_store/land_queue.rs` at 03821e0	`BATCH = 128` and `LEASE_MS = 10_000`
Batched landing, one call per batch	`backend/crates/yoneda-platform/src/repo_store/submitted.rs` at fef94e7	The module comment says the combiner submits every waiting landing in one call; the submit accepts at most 256 items, and path-disjoint landings append changesets
Within-batch write-skew check	`backend/crates/yoneda-platform/src/repo_store/merging.rs` at fef94e7	`BatchWrites::admits` returns false when any of a member's reads, cut to file level, or writes overlaps a write accepted earlier in the batch, and `accept` records a landed member's writes
Deferral of a skewing member	`backend/crates/yoneda-platform/src/repo_store/submitted.rs` at fef94e7	A member that `admits` rejects is answered `deferred`, and a landed member's writes are passed to `accept`
Batch mode configured	`cloudflare/wrangler.platform.jsonc` at fef94e7	`LAND_MODE` is `batch`, with `BEGIN_MODE` and `READ_MODE` set to `head`
Sixty-four commit shards	`cloudflare/worker/repo-shards.ts` at fef94e7	`SHARDS = 64`, `shardOf` uses the commit's first byte, and older commits are still found in the 16 legacy shards
Burst-span rate and overload count at fef94e7	`tools/platform-fanout-bench.mjs` at fef94e7	`landRate` spans the first landing request to the last answer, `overloaded` counts commit and landing responses, and reads are counted separately
Changeset landing without Git objects	`backend/crates/yoneda-platform/src/repo_store/changesets.rs` at dd6c9a8, branch `claude/native-p95e`	The module comment says a disjoint landing appends its changeset with one indexed query, without Git objects or a merge commit, and the projector builds the branch's Git commit later
Changeset validation against the log	`backend/crates/yoneda-platform/src/repo_store/changesets.rs` at dd6c9a8, branch `claude/native-p95e`	`route` collects the commit's touched paths and read set and calls `touched_since`, which queries `txlog_paths` above the base version; a hit goes to the exact optimistic check
Landed paths indexed on append	`backend/crates/yoneda-platform/src/repo_store/land.rs` at dd6c9a8, branch `claude/native-p95e`	The append path calls `changesets::index_paths` after inserting the log entry
Burst-span rate and overload count	`tools/platform-fanout-bench.mjs` at dd6c9a8, branch `claude/native-p95e`	`landRate` divides successful landings by the time from the first landing request to the last answer, and `overloaded` counts commit and landing responses only

Claim	File and commit	What the code shows
Git projection lag	`tools/platform-fanout-bench.mjs` at dd6c9a8, branch `claude/native-p95e`	`projectionLag` waits up to five minutes and reports the projected time minus the committed time of each changeset
Land combiner limits	`backend/crates/yoneda-platform/src/repo_store/land_queue.rs` at cfa4845	`LEASE_MS = 30_000`, `BATCH = 64`, and `claim` answers a held lease with a retry of at most 250 ms
Checkpoints every 64 versions	`backend/crates/yoneda-platform/src/repo_store/rows.rs` at cfa4845	`CHECKPOINT_EVERY = 64`
Three read tiers	`cloudflare/worker/repo-objects.ts` at cfa4845	`RepoObjects` reads the Cache API, then R2, then the store, and counts hits per tier
Write token reused per isolate	`cloudflare/worker/repo-objects.ts` at 8682aea	`TOKEN_REUSE_MS` caches the write token per repository
Large commits write only the commit object through	`cloudflare/worker/repo-ops.ts` at 73b008e	A commit with more than `OVERLAP_CACHE_OBJECTS` objects writes only its commit object through the tiers
Read-only clone access	`cloudflare/worker/repo-objects.ts` at 662b343	`cloneAccess` mints a read token for the Artifacts remote
Clone verified by the live check	`tools/platform-live.mjs` at 662b343	Clones the projected repository and runs `git fsck --strict`
Phases are waits, not CPU	`cloudflare/worker/repo-timing.ts` at ebbee09, branch `claude/native-p95`	The header comment states that Workers clocks advance only across input and output
Queue, Container, Workflow and D1 bindings	`cloudflare/wrangler.platform.jsonc` at cfa4845	Consumer batch 1 with 3 retries and a dead-letter queue, at most 8 `standard-1` containers, the import Workflow, and the D1 index
Import through a Workflow	`cloudflare/worker/repo-import-workflow.ts` at 0fa4680	`PlatformImportWorkflow` extends `WorkflowEntrypoint`
Warm-start seeds	`cloudflare/worker/repo-seed.ts` at e1e2825	The header comment and the seed size limits for check runs
Benchmark workload	`tools/platform-live-bench.mjs` at 211de73	A 100-file seed, writers that commit `writers/<i>.txt` and then land, warm reads, and sequential edits on a small and a large repository
Basin change log, unmerged	`cloudflare/worker/repo-changelog.ts` at 6cb70fa, branch `claude/native-obs`	Streams change records for analytics outside the commit path

B Model arithmetic

Calibration. The local service time is $s = 1/662.1$ s, or 1.510 ms. With $c = 7$ the authority term is $99 \times 7 \times 1.510 = 1{,}047$ ms for H1 and H2, $90 \times 7 \times 1.510 = 952$ ms for H3 and $92 \times 7 \times 1.510 = 973$ ms for H4. The depth correction is two extra prepare calls, two extra tree rounds and one extra status read at their sequential per-call medians: $2 \times 214/4 + 2 \times 9/3 + 145/3 = 161$ ms in H3 and $2 \times 215/4 + 2 \times 8/3 + 146/3 = 162$ ms in H4. The tier term is the growth of the push step's p95, $1{,}704 - 1{,}139 = 565$ ms in H3 and $1{,}609 - 1{,}127 = 482$ ms in H4, with mean 524 ms. For H2, for example, the prediction is $1{,}382 - 161 + 1{,}047 + 524 = 2{,}791$ ms.

Hosted cross-check of the service time. The authority phases' p95 values sum to 1,320 ms in the H3 burst and 601 ms in H3's sequential commits. Removing the two extra prepare calls and the extra status read of the sequential commits at their per-call p95 ($2\times219/4+153/3 = 161$ ms) leaves 440 ms, a growth of 880 ms. The same steps for H4 give 1,607 ms against $770-(2\times236/4+187/3) = 590$ ms, a growth of 1,017 ms. Dividing by $(N_{\mathrm{ok}}-1)c$, which is 630 and 644, gives 1.40 and 1.58 ms.

Ceilings. In C1 a landing cycle makes $7 + 2.24 = 9.24$ authority calls besides polls, where 2.24 is the H4 sum of 0.99 enqueues, 1 direct land, 0.03 batch landings and 0.22 calls by the lease holder to prepare, record, classify and read status. C2 makes $1 + 2.24 = 3.24$, and C3 adds one call for answering each landing's wait, 4.24. At 1.51 ms per call the ceilings are 71.7, 204 and 156 landings per second. The land combiner's ceiling is $64/2.6 = 24.6$ in C1 and C2, or $64/2.4 = 26.7$ at the low end of the early cycle, and $128/11.5 = 11.1$ in C3, between $128/12.5 = 10.2$ and $128/10.5 = 12.2$ over the measured steps. In the early revisions the push queue's ceiling is $64/0.44 = 145$ commits per second, or $145 \times 64/65 = 143$ landings per second with one merge commit per batch; from C2 on, commits do not pass through it.

Knee. The unloaded commit median is $1.632 - 0.161 = 1.47$ s in C1 and 248 ms from C2 on. At the combiner's ceiling every landing needs a merge and the combiner is always busy, so a landing waits $1.5\,T_c$, which is 3.9 s at a 2.6 s cycle and 17.25 s at 11.5 s. The knees at $Z = 60$ s are $24.6 \times (60 + 1.47 + 3.9) = 1{,}610$ agents for C1, $24.6 \times (60 + 0.25 + 3.9) = 1{,}580$ for C2, and $11.1 \times (60 + 0.25 + 17.25) = 860$ for C3.

Polls. In C1 the unloaded poll iteration is the mean sleep of 15 ms plus two idle round trips of about 47 ms, or about 0.11 s, with two calls per iteration. In C2 it is the mean pause of 240 ms at the back-off's cap plus one round trip, about 0.29 s, with one call. By Corollary 6.12 the thresholds are $Q^* = 0.11/(2 \times 0.00151) \approx 36$ and $0.29/0.00151 \approx 192$ waiting landings. A busy combiner keeps $1.5\,XT_c'$ landings waiting, which reaches 36 at about 9 landings per second with a 2.6 s cycle.

Fixed-point approximation. For each N the solver finds four quantities together: throughput X, waiting landings Q, authority utilization U and the mean per-call authority wait W. Given W, the combiner cycle is $T_c' = T_{c0} + n_c W$, with $n_c = 9$ in C1 and C2 and $n_c = 2$ for the combiner of C3, and its ceiling is $k_{\max}/T_c'$. The landing wait is $(1 - f_m)(t_{\mathrm{ff}} + 2W) + f_m T_c'(1 + \rho_c/2)$, with f_m and ρ_c from Assumption 6.6 and a direct compare-and-set of $t_{\mathrm{ff}} = 86$ ms. If the population bound $N/(Z + L_c + \text{landing wait})$ is below the combiner's ceiling, X equals it and $Q = X f_m T_c'(1 + \rho_c/2)$; otherwise $X = k_{\max}/T_c'$ and $Q = N - X(Z + L_c)$. Here L_c is the unloaded commit median plus the commit's own authority calls times W.

Given X and Q, the authority's utilization is

$$U(W) = s\Big(Xc_{\mathrm{open}} + \frac{pQ}{\tau + pW}\Big), \tag{4}$$

where c_{open} is the configuration's calls per landing cycle besides polls and p its calls per poll iteration, with the polling term dropped in C3. U decreases in W toward $sXc_{\mathrm{open}} < 1$, so the set where $U(W) < 1$ is an interval $(W_{\min}, \infty)$, with $W_{\min} = 0$ when $U(0) < 1$. On that interval the map $W \mapsto U(W)\,s/(2(1 - U(W)))$ decreases from infinity, or from a finite value when $W_{\min} = 0$, to a finite limit, while W itself increases without bound, so the equation $W = U(W)\,s/(2(1 - U(W)))$ has exactly one root. The solver takes that root and iterates the outer quantities with damping until they stop changing.

For large Q the root satisfies $pQ/(\tau + pW) \approx 1/s - Xc_{\text{open}}$, so $pW \approx pQs/(1 - sXc_{\text{open}}) - \tau$, which matches the bound of Corollary 6.12. At $N = 1{,}000$ in C1 the solution is $X = 14.9$, $Q = 76$, $U = 0.99$ and $W = 92$ ms; at $N = 10{,}000$ it is $X = 0.5$ and $W \approx 15$ s, the collapsed state. In C2 at $N = 10{,}000$ it is likewise $X = 0.5$ and $W \approx 15$ s. Commit latency in Table 7 is the configuration's unloaded commit p95, $1.70 - 0.16 = 1.54$ s in C1 and 345 ms from C2 on, plus the commit's authority calls times W.

Scale. Scion's published daily average is $30{,}000/86{,}400 = 0.347$ merges per second. The projected ceilings of 11.1 and 24.6 landings per second are 32 and 71 times that average, and the measured burst-span rates of 235.1 to 491.2 per second on fef94e7 are 678 to 1,416 times it, ratios of quantities that Section 8 explains are not comparable.

Table 5: Inputs to the capacity model. Measured inputs come from Section 5; ranges are carried into the sensitivity analysis.

Input	Value	Source
Authority calls per commit under the burst, c	7	Calls per request in H3 and H4
Authority calls per landing, excluding polls, c_{land}	2.24	Calls per request in H4
Unloaded poll iteration time, τ, early revisions	0.11 s (0.08 to 0.14 s), two calls	Sleep of 5 to 25 ms plus two idle round trips
Unloaded poll iteration time at 4407515	0.29 s, one call	Back-off from 20 to 320 ms plus one round trip
Authority calls per commit at 4407515 and 03821e0	1	Calls per request
Unloaded commit p95 at 4407515	345 ms	Sequential one-file edits
Combiner step at 03821e0, with $k \leq 128$	10.5 to 12.5 s, central 11.5 s	Two alarm wall times
Authority service time per call, s	1.51 ms (1.0 to 5.0 ms)	Inverse of 662 commands per second in one local process
Hosted cross-check of s	1.40 to 1.58 ms	Growth of authority waits at p95, H3 and H4
Documented authority rate for complex work	200 to 500 calls per second	Cloudflare guidance [14]
Unloaded commit p95, L_0^{95}, per run	1,936, 1,382, 1,720 and 1,757 ms	Sequential commits in H1 to H4
Extra tree rounds of sequential commits, δ	161 ms	Sequential phases in H3 and H4
Growth of the push step at p95 under the burst, Δ_T	482 to 565 ms, mean 524 ms	Push phase in H3 and H4
Combined send time at batch 47 to 63	409 to 440 ms	Burst send phase
Combined send time for one commit	121 ms	Sequential send phase
Land combiner cycle in the early revisions, T_c, with n_c authority calls	2.6 s (2.4 to 2.6 s), $n_c = 9$	Lease holder's phases in H3 and H4
Batch caps, $m_{\max}$ and $k_{\max}$	64 and 64	Code
Scion merge rate	More than 30,000 per day, about 0.35 per second	Scion post [3]
Think time Z	60 s (10 to 600 s)	Assumption
Peak-to-average factor π	2, 5 or 10	Assumption

Table 6: Calibration of the burst predictor of Eq. (3) against the measured commit p95 of the four hosted runs, with $c = 7$, $s = 1.51$ ms, $\delta = 161$ ms and $\Delta_T = 524$ ms. Times are in milliseconds; errors are relative to the measured p95.

Run	N_{ok}	L_0^{95}	Authority term	Measured	Predicted	Error
H1	100	1,936	1,047	3,731	3,345	−10.4%
H2	100	1,382	1,047	2,802	2,791	−0.4%
H3	91	1,720	952	3,691	3,033	−17.8%
H4	93	1,757	973	3,316	3,092	−6.8%

Table 7: Projected (not measured), for the Cloudflare realization, at $s = 1.51$ ms and $Z = 60$ s, except the last row, which is measured. Ceilings are in landings per second. Commit latency is the configuration's unloaded commit p95 plus the mean projected authority wait, not a projected percentile; entries marked collapsed are mean values of a saturated system.

	C1	C2	C3
Authority calls per landing cycle, excluding polls	9.24	3.24	4.24
Authority ceiling without polls	71.7	204	156
Land combiner ceiling	24.6	24.6	11.1
Poll saturation threshold Q^* (waiting landings)	36	192	none
Combiner knee N^* (agents)	1,610	1,580	860
Peak before collapse (landings per second)	16.5	24.5	none
Agents at the peak	about 1,125	about 1,575	not applicable
Landings per second at $N = 100$	1.5	1.6	1.5
Landings per second at $N = 1{,}000$	14.9	15.6	11.1
Landings per second at $N = 10{,}000$	0.5 (collapsed)	0.5 (collapsed)	11.1
Commit latency at $N \leq 1{,}000$ (s)	1.5 to 2.2	0.35	0.35
Commit latency at $N = 10{,}000$ (s)	about 107 (collapsed)	about 15 (collapsed)	0.35
Measured, 1,000-writer burst on dd6c9a8: landings per second and landed	not run	0, with 4 landed	5.9, with 385 landed