Validating a High-Performance Cloud Object Store with Lightweight Formal Methods

Vimuth Fernando
Amazon Web Services

Rajeev Joshi
Amazon Web Services

Bernhard Kragl
Amazon Web Services

Julien Mascart*
Meta

Matthew Russo
Amazon Web Services

Sarek Skotåm
Amazon Web Services

Colin Walker
Amazon Web Services

Jake Wires
Amazon Web Services

Abstract

We report on our experience validating S3 Express One Zone, a high-performance Amazon S3 storage class launched in November 2023. We focused on its distributed metadata index, which implements a hierarchical namespace using distributed transactions. We check strong consistency and durability under crashes, network delays, and timeouts. Building on our prior work on *lightweight formal methods*, we combined executable reference models with property-based testing and stateless model checking. Applying these techniques to a production distributed system required three new techniques: a white-box approach to checking serializability; compositional validation of the data and control planes; and long-running canaries that use reference models to check correctness across deployments. We extended Shuttle, our open-source stateless checker, to support a production async Rust runtime and failure injection, including cancelled operations and timeouts. Our main finding is that this approach is sustainable at engineering scale. After launch, the engineering team took ownership of models and test infrastructure, increasing its share of validation commits from 41% to 84% and independently extending the models for conditional writes, object append, and atomic rename, with little involvement from the formal methods team.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: lightweight formal methods, cloud storage, distributed systems, property-based testing, stateless model checking, serializability checking

*Work done while at Amazon Web Services



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
SOSP '26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843899

ACM Reference Format:
Vimuth Fernando, Rajeev Joshi, Bernhard Kragl, Julien Mascart, Matthew Russo, Sarek Skotåm, Colin Walker, and Jake Wires. 2026. Validating a High-Performance Cloud Object Store with Lightweight Formal Methods. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830418.3843899

1 Introduction

S3 Express One Zone (S3 Express) is a high-performance Amazon S3 storage class that provides single-digit millisecond latency for data access and supports millions of operations per second per bucket, while ensuring strong read-after-write consistency. It targets data-intensive applications (AI/ML training, financial modeling, media processing, high-performance computing) that use hundreds or thousands of parallel compute nodes to access data. The system was implemented largely from scratch in async Rust and was publicly launched in November 2023. This paper reports on our experience validating the correctness of S3 Express.

Given the complexity and importance of this system, we embedded a small formal methods team with the engineering team from the beginning of development. Our goal was for formal methods to provide correctness assurance without slowing development, and for the validation approach to remain sustainable and maintainable after the formal methods experts left. Inspired by our prior work on S3 ShardStore [7] and similar efforts [8, 10, 17, 23], we adopted *lightweight formal methods*.

Our validation focused primarily on S3 Express's new distributed metadata index, since the storage layer used ShardStore, inherited from regional S3. Building on our experience with ShardStore, we adopted the ideas of writing executable reference models in Rust, using property-based testing for automatically generating tests, and using the Shuttle concurrency checker [33] to check concurrent code. These ideas carried over fairly directly. However, validating a distributed system presented several new challenges that we had not faced on ShardStore (which runs on a single storage node).

First, the distributed index must handle network delays, timeouts, and host failures while ensuring that the data path's consistency guarantees are preserved even as the control plane moves partitions between hosts. Second, the index was built using a new transaction library supporting distributed transactions involving collections of key-value maps; ensuring strong read-after-write consistency required checking transaction serializability under crashes, timeouts, and network partitions. Existing approaches to serializability checking [6, 25] do not handle our production behaviors, such as key deletions, range operations, and aborted transactions that cannot simply be dropped. Third, the index makes heavy use of async Rust, which introduced new concerns such as *cancellation safety*: operations can be dropped at intermediate `await` points, leaving partial side effects. Existing concurrency checkers such as Shuttle did not support the async Rust runtimes used in our production code. Fourth, the control plane consists of background processes running asynchronously, which does not map directly to the request/response validation approach we used in ShardStore. Finally, S3 Express was expected to keep evolving after launch, through addition of new features (such as object append and rename) as well as ongoing behind-the-scenes performance improvements, so our validation had to check correctness *across* software deployments and version upgrades. Subtle bugs from the interaction of different versions are hard to find with conventional tests, which typically run against a single version.

Our validation applies a lesson we carried over from ShardStore: rather than checking one global correctness property with a single tool, we decompose correctness into separate properties and check each with the technique that suits it best. This led us to apply separate approaches to checking the data path and the control plane. The data path has a well-defined request/response interface, so we reused the refinement approach from ShardStore: we check that the implementation's responses match those of an executable reference model on randomly generated operation sequences. The control plane has no request/response API to build a reference model against, so we instead identified *safety invariants* that hold in all states and *stability properties* that hold at quiescent points. This separation also helped with maintainability, since it let us write smaller, separate models that engineers could understand and evolve independently.

We use property-based testing to automatically generate operation sequences and check conformance against an executable reference model that acts as the oracle, a form of differential testing we call *reftesting*. We run reftests across a range of fixtures: fast in-memory tests on developer desktops, larger-scale tests that run across thousands of instances in the deployment pipeline to explore more schedules, and long-running stateful canaries that run for weeks or months on pre-production stacks. All of these run as approval steps in deployment pipelines, so any failure blocks the deployment.

This paper makes the following contributions:

- We present a white-box algorithm for efficiently checking serializability of multi-partition transactions. We build a serialization graph from *instrumentation events* emitted by the implementation, which determine graph edges and their direction. Our algorithm handles production behaviors that prior approaches do not, including key deletions, range operations, and aborted transactions (§5).
- We present a compositional approach that checks data-path and control-plane correctness independently. The data-path tests assume properties that the control-plane tests are responsible for ensuring, which lets us write smaller, separate models that are easier to understand and maintain (§6).
- We introduce *persistent reftesting*: using executable reference models to build stateful canaries that check correctness continuously across software deployments and version upgrades, catching bugs that single-version testing misses (§7).
- We report on 2.5 years of post-launch experience showing that lightweight formal methods can be sustained by a much larger and faster-moving engineering team than in our prior work on ShardStore. The engineering team's share of validation commits has grown from 41% to 84%, and it has independently shipped complex features (conditional writes, object append, and atomic rename) with little involvement from formal methods experts (§8).

2 System Overview

S3 Express consists of three main components (Fig. 1): a *frontend fleet* that performs routing, access control, and erasure-coding; a *data storage fleet* that stores customer data as erasure-coded shards; and a distributed *index subsystem* that maintains the mapping from customer keys to metadata and storage locators. Our validation effort focused on the index subsystem (the red dashed box in the figure), which was complex and newly built for S3 Express. The data storage layer is built on top of ShardStore, and the frontend is largely stateless and validated using a black-box approach [22]. Readers familiar with regional S3 will recognize this high-level architecture, but S3 Express shares only the storage node software (ShardStore) and some low-level libraries with it; the index subsystem in particular is new.

A key feature of the index is that it implements a *hierarchical namespace*: a tree whose nodes correspond to key prefixes, where each node is stored as a persistent key-value *map* managed by the index. An object with key `foo/bar/baz` is represented by internal nodes `foo/` and `bar/`, and a leaf node `baz`, where the map for `foo/` holds an entry pointing to the map for `bar/`, and the map for `baz` stores the object's metadata and the storage locators where its data resides. The

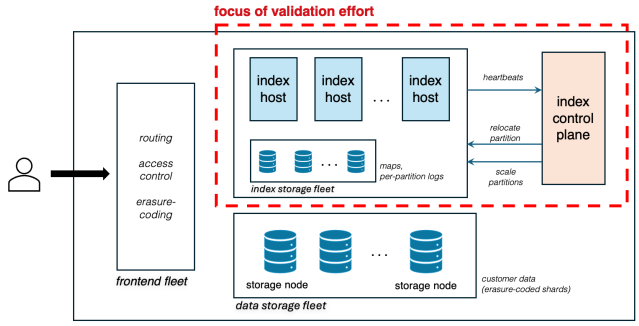


Figure 1. Overview of system architecture.

hierarchical design was chosen because it makes it easier to implement operations such as atomic rename. To spread load across hosts, the index stores maps across multiple *partitions*. Each partition is assigned to exactly one index host at a time, and a single map may itself span multiple partitions.

The index subsystem has two components: a *data path* that serves index operations, and a *control plane* that keeps partitions correctly placed and healthy across the fleet.

The data path runs on a fleet of hosts, backed by a storage fleet that persists maps and a per-partition write-ahead log. An index operation traverses the hierarchical namespace and updates the affected maps atomically, using a *transaction library* that provides serializable transactions over arbitrary sets of maps across several partitions. Transactions simplify complex operations such as conditional writes and atomic rename while ensuring strong read-after-write consistency, making the library central to correctness and performance.

The library supports three transaction modes: read-write, read-only, and lock-free, and uses fine-grained two-phase locking and distributed two-phase commit. To keep latency low, transactions write entries to per-partition write-ahead logs and return once log entries are durable; a *checkpointing* process running in the background applies these committed entries to update the actual maps. Uncommitted transactions leave no visible side effects, so recovery replays the log forward with no undo. The library also supports periodic consistent *backups* across all partitions. We defer details to later sections (§4, §5).

The control plane manages the placement and health of partitions. It monitors host health through periodic heartbeats and relocates partitions from one host to another in response to failures or load imbalances. It also supports *partition scaling* which splits partitions (to improve performance) or merges them (to reduce resource use) in response to heat. The control plane also uses a fencing mechanism to prevent *zombie hosts* (hosts that appear failed but are still processing requests) from corrupting state. Control plane operations interact with concurrent data-path transactions: a transaction in flight when its partition is relocated may need to be aborted and retried, and the system must ensure that no committed data is lost during the transition.

3 Validation Approach

We validate S3 Express using a combination of property-based testing and systematic exploration of concurrent interleavings. This section describes the validation methodology that we apply to both the data path and control plane.

3.1 Property-Based Testing

Following prior work on checking stateful systems [4, 7, 17, 21], we use property-based testing (using the proptest library [2]) to generate test scenarios over an *operation alphabet*. Each component we validate has its own alphabet, which we spell out in its section (the transaction library and the namespace in §4, the control plane in §6). Broadly, an alphabet mixes client operations that model visible API calls, background operations invisible to clients (backups, relocations, and scaling), and injected failures (crashes and network failures). We use several strategies to improve coverage: biasing toward small key sets to create more conflicts, favoring state-changing mutations, and running separate test suites for different failure combinations to see how features interact. Operations generated by proptest are initially incomplete: they specify the operation type but not all arguments. During test execution, these arguments are *concretized* on the fly based on current state (e.g., a delete picks from among the existing keys so it does not trivially fail).

3.2 Concurrency Testing

Rust's type system eliminates data races at compile time, but it does not rule out higher-level concurrency bugs. S3 Express is implemented in async Rust using the Tokio [1] runtime. Rust's async model differs from those in other languages in two ways that matter for testing. An async function in Rust compiles to a state machine (a *future*) that makes progress only when the runtime *polls* it; the future may yield at any intermediate `await` point (e.g., while waiting for a lock or a message), and the runtime is then free to poll a different ready task. Tasks can therefore interleave at any `await` point, creating a large space of schedules that traditional testing does not systematically explore.

A second difference concerns cancellation. A future is cancelled simply by *dropping* it: the runtime stops polling it and drops whatever is in scope, but no code past the current `await` point ever runs. There is no cancellation exception (as in Python's `asyncio` or Kotlin coroutines) that would run `finally`-style cleanup. Cancellation arises routinely through Tokio's `select!` macro, which races several operations and drops the losers as soon as the first completes. Because a future can be dropped at any `await` point, often deep inside a call path where the author never anticipated interruption, operations must satisfy *cancellation safety*: they should

be correct even when dropped mid-update. Cleanup happens only where the author added a `Drop` implementation, so an unguarded multi-step update can be left half-finished: a lock released while the data it guards is half-written, or a request sent but its response never handled.

To systematically explore interleavings, we use Shuttle [33], a stateless model checker for Rust that controls the scheduler and replaces async I/O with in-memory mocks. Because safe Rust (which S3 Express uses throughout) is free of data races, it suffices to interleave tasks only at `await` points: every execution on a multi-threaded runtime is equivalent to some sequential interleaving at these points, which is exactly what Shuttle explores. For each test, Shuttle runs many iterations with different schedules and returns a replayable trace in case of failure. Shuttle did not support Tokio when we started, so we reimplemented Tokio's concurrency primitives so they can be controlled by the Shuttle runtime. (This Tokio support is now publicly available in Shuttle on GitHub.) We also extended our test framework to inject task drops at intermediate `await` points (to check cancellation safety) and to inject timeouts on demand. Injecting timeouts this way lets Shuttle explore timeout-driven failures without modeling time, a hard problem in stateless model checking.

Concurrency testing must deal with a behavior specific to asynchrony: an operation can return success while work it started is still ongoing. For instance, a transaction commit returns once its log entries are durable, but checkpointing continues to update maps in the background; similarly, a two-phase commit returns once all participants have prepared, while the commit protocol continues to run to completion. Testing correctness means exercising how this background work interleaves with operations issued afterward.

We therefore organize concurrent reftests as a sequence of *rounds* (Fig. 2 shows an example for the data path; see §4.2 for details). Each round spawns multiple concurrent *clients*, each executing a sequence of *jobs*, and we wait for all clients to finish before starting the next round. What constitutes a job depends on the component: for the data path, jobs include transactions, backups, partition relocations, or host crashes; for the control plane, starting or stopping workers or initiating partition scaling operations. Because a client's background work can spill past the end of its round, Shuttle explores interleavings both within a round and across round boundaries.

Because all operations in round N *visibly* complete before any operation in round $N+1$ begins, rounds impose ordering constraints that the system must satisfy. For the data path, any valid serialization of transactions must be consistent with this client-visible ordering across rounds. For the control plane, rounds enable *quiescence detection*: we can detect when the system has finished reacting to a round of perturbations before checking that it has reached a stable state.

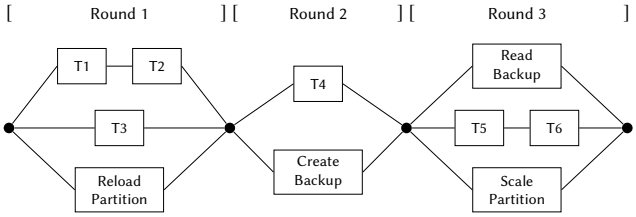


Figure 2. Example scenario for the transaction library.

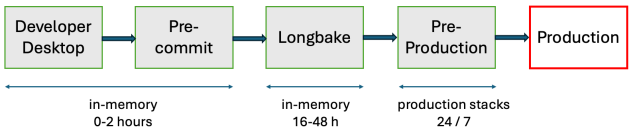


Figure 3. Deployment pipeline showing where reftests run. Each stage gates promotion to the next; any failure blocks deployment.

3.3 Deployment Pipeline

Fig. 3 shows how our validation steps are integrated into the deployment pipeline. We run tests at multiple levels of fidelity, trading off between execution speed, debugging ease, and coverage.

Sequential reftests run operations one at a time against an in-memory stack with mocked storage and network layers. Because there is no concurrency, validation is straightforward: each operation's result is compared directly against the reference model. Sequential reftests are fast, run on developer desktops, and provide immediate feedback during development.

Concurrent reftests use Shuttle to explore different interleavings of concurrent operations. For each test case, Shuttle executes many iterations with different random schedules. If a failure is found, the test seed enables deterministic replay. Concurrent reftests typically run as part of a pre-commit hook rather than interactively during development.

Longbake reftests run at scale: we spin up thousands of instances executing for many hours, each with different random test cases and schedules. We exploit the fact that random exploration is "pay-as-you-go", so more instances and longer tests means we cover more of the state space. Longbake runs in the deployment pipeline before deployment to pre-production and production stages.

Finally, **persistent reftests** run against a real deployment on pre-production infrastructure, rather than the in-memory stack used by the previous tests. A long-lived reference model runs alongside the deployment for weeks or months as an oracle, driving operations and checking the results as the system is upgraded and redeployed underneath it. This catches bugs that arise only from the interaction of different software versions.

The next sections describe how we apply this validation methodology to the data path (§4, §5), the control plane (§6), and the composed system across deployments (§7).

4 Validating the Data Path

For the data path, the key property is behavioral equivalence: for every sequence of operations, the implementation must produce the same responses and reach an equivalent final state as a reference model specifying correct behavior. This section describes how we specify correct behavior and check it for both sequential and concurrent executions. The one part we defer is checking that concurrent transactions are *serializable*, which is involved enough to warrant its own section (§5).

4.1 Reference Models

Each reference model implements the same interface as its associated component (using the same Rust types and traits), but with a much simpler implementation. Reference models are simpler in several ways: (1) while they model all non-error behaviors faithfully, they do not model many error cases (e.g., resource exhaustion, handling of device failures), (2) the models are sequential even though the implementation is concurrent, and (3) the models use simple data structures (hash maps and arrays) because efficiency is not a concern.

Simplifications (1) and (2) deserve some explanation. For (1), we primarily focus on correctness of successful operations: if the system processes a request and returns success, the result must be correct; if it returns an error, we only check that system state did not change. We do not attempt comprehensive availability testing, since availability bugs are generally best caught by targeted load testing and careful monitoring of metrics. That said, if an availability property is easy to check with our approach, we add a targeted check for it (see the cache-miss bug below). Simplification (2) is justified because the index is designed to ensure strong consistency. Thus any concurrent execution must be equivalent to *some* sequential execution, so we can check correctness by verifying that the responses and final state match some execution of the sequential reference model. (Finding the matching execution of the reference model is nontrivial: we explain in §5 how we do this efficiently.)

We wrote separate reference models for the two main data path components: the namespace and the transaction library. This separation was driven by two considerations. First, the namespace and library were developed by different teams, and separate models allowed each team to work more independently. In particular, early versions of the namespace used the library's reference model as a mock implementation, since the reference model exposes the same Rust API and is directly executable—a benefit of writing models in the same language as the implementation. Second, having the library as a separately testable component allowed us to exercise its full API, including transaction patterns that the namespace did not yet use, but might need for future features.

4.2 Concurrent Data-Plane Validation

As described earlier (§3), our concurrency tests are organized as a sequence of rounds with concurrent clients each executing a sequence of jobs under Shuttle. The alphabet for the transaction library consists of *transactions* plus other operations and injected failures that run concurrently with them and must not violate serializability. A transaction has one of three types (read-write, read-only, or lock-free) and runs a sequence of map operations before committing or aborting. The other operations are reloading a partition, creating a backup, reading a backup (which must return a consistent snapshot), and scaling a bucket (by either splitting or merging partitions); the injected failures are host crashes and network failures. The namespace alphabet is simpler: a job is either an object operation (put, get, delete, or multipart upload) or a partition relocation.

Fig. 2 shows a typical scenario. In Round 1, three clients run concurrently: one executes transaction T1 followed by T2, another executes T3, and a third reloads a partition. Depending on the schedule, T3 may observe state before, between, or after T1 and T2; the concurrent reload can cause transactions to abort part-way through the commit protocol. Round 2 has two clients: one creates a backup, while the other performs transaction T4; the backup must capture state before or after T4. Round 3 has three clients: one reads the backup from Round 2, another performs transactions T5 and T6, and a third scales the bucket. Reading the backup must return a consistent snapshot: it reflects T1–T3 (completed before the backup), may or may not reflect T4 (concurrent with it), and does not reflect T5 or T6 (run afterward). At the same time, T5 and T6 must remain consistent with the concurrent scaling operation.

The transaction library must satisfy *serializability*: any concurrent execution must be equivalent to *some* sequential execution of the same transactions. The index must satisfy *per-key linearizability*: concurrent operations on any given key must appear to take effect atomically in an order satisfying read-after-write consistency. In both cases, checking involves finding a valid sequential ordering and replaying it against the sequential reference model (§5).

Because all operations in round N visibly complete before any operation in round $N+1$ begins, we impose additional ordering constraints: any valid serialization must be consistent with this client-visible ordering. The serializability checker uses these ordering constraints to prune the search for a valid serial order (§5).

Transactions. After all rounds complete, we check that the run's transactions are serializable: that *some* serial order of the transactions, replayed against the reference model,

matches what the implementation did during the concurrent run. Concretely, the reference model must return the same result for each operation, and must end in the same state. We compare final state at the end of a run by enumerating all maps and their key/value pairs in the implementation and checking that they match the reference model exactly.

Concurrent reftest scenarios must cover not just committed read-write transactions but also read-only transactions, lock-free transactions, and transactions that are aborted by the client. One additional twist for us is that aborted transactions in S3 Express have an unusual requirement: unlike textbook definitions, where aborted transactions can simply be dropped when checking serializability, we need to account for the reads they performed. This is because the namespace may abort a transaction based on values it has read, and those reads must be consistent with the serial order of committed transactions for the index to satisfy per-key linearizability.

To see why aborted transactions cannot be ignored: consider two concurrent `Delete`s of the same key, creating transactions T1 and T2. One (say T2) aborts because the key is already gone—it read the key's absence. T2 must serialize *after* T1, and cannot simply be dropped from the serializability check. If it were dropped, we would miss bugs where both transactions abort and report failure to the client (a violation of per-key linearizability, since at least one should succeed).

Partition relocation and scaling. Partition relocation and scaling run concurrently with data-path transactions. A partition relocation racing with an in-flight transaction can cause the transaction to fail at different points in the commit protocol (before prepare, during prepare, after prepare but before commit), each requiring different handling by the recovery logic. Note that serializability must still hold for all transactions that finish successfully even in the presence of relocation and scaling operations.

Crashes and network failures. We inject host crashes and network failures concurrently with transactions and partition operations. We model network failures as dropped messages: an RPC can time out and return an error to its caller, which may result in a retry. After a crash, recovery replays the log to restore committed state: all transactions that returned a successful commit before the crash must be reflected in the recovered state, and uncommitted transactions must have no visible side effects. The recovered state must be consistent with the same serial order that was valid before the crash. During recovery, the system may have to play forward transactions that successfully prepared in order to complete the two-phase commit protocol.

Partition backups. Partition backups capture a consistent point-in-time snapshot of all partitions. We include backup creation as a job type in concurrent reftests, allowing them to race with user transactions and failures. A correct backup represents a single point in the serialization order: it must reflect all transactions that serialized before it and none that serialized after. We check this by including backups in the serialization graph alongside transactions (§5).

Example bug: cache-miss handler. As described in §3, we check cancellation safety by injecting cancellations at `await` points. For the transaction library, this uncovered a subtle bug in its block cache. The library stores each map as a set of blocks and caches them in memory to avoid re-reading from storage; on a cache miss, a handler reads the missing block from storage. The handler registered an ongoing read in a `pending_reads` table, awaited the storage read, and cleared the entry. If the caller was cancelled while the read was in progress (e.g., because another `select!` branch completed), the future was dropped and the `pending_reads` entry was left uncleared. Subsequent reads to the same block found the stale entry, concluded that a read was still in progress, and blocked indefinitely. The block became permanently unavailable until the partition was reloaded.

This bug was originally not caught by our concurrent reftests, and was only uncovered during pre-production testing. Upon investigation, we noticed that the reftests had in fact exercised the code path. However, our specification was too weak—it treated all errors as acceptable when error injection was enabled, masking the permanent unavailability. To fix, we added an *availability check*: after running with error injection for several rounds, we disable error injection and continue executing. In this "error-free" phase, the system should behave as if no errors were ever injected. Any persistent error, such as the permanently unavailable block, surfaces as a test failure. The fix was to implement a `Drop` guard that clears the `pending_reads` entry when the future is dropped, restoring cancellation safety.

5 Checking Serializability

After a concurrent reftest execution with Shuttle, we have a collection of transactions that ran concurrently on the implementation and must check that the execution is serializable—that is, that the results are consistent with *some* sequential execution. Unfortunately, checking serializability is NP-complete in general [32], so naive searching over possible orderings is intractable even for modest test cases. Existing serializability checkers [6, 37] make assumptions that do not hold in our setting: they lack support for deletes, or they assume each transaction's commit-or-abort outcome is immediately known, whereas failure injection can cause a transaction to return an error yet still apply its changes durably. Black-box inference approaches like Elle [25] infer each transaction's true outcome from observed data rather than trusting the returned status. Elle recovers write order by tracing the sequence of values each key has held, which is why it requires constrained test generation (append-only lists with unique values); our operations overwrite and delete

keys, destroying the value history Elle relies on. We cannot use lock acquisition order since lock-free transactions do not acquire locks. Nor do we have a global totally-ordered commit log: each partition maintains its own log, and read-only transactions do not appear in logs. We overcome these challenges with a white-box approach. The implementation already assigns each write a persistent sequence number to order operations within a partition, so we instrumented the code to emit these numbers to our checker. Using them, we can efficiently check serializability for multi-partition transactions even in the presence of host crashes, timeouts, and task cancellation.

5.1 Algorithm Overview

During each concurrent execution, the implementation emits a stream of *instrumentation events* that record how each transaction accessed shared state and the sequence numbers assigned to those accesses. The serializability checker consumes this event stream to construct a *serialization graph* that precisely captures all ordering constraints between transactions. The graph can be constructed in time polynomial in the number of transactions, and has the property that any topological ordering yields a valid serialization, so we can replay an arbitrary ordering against the reference model.

The algorithm proceeds in three steps. First, we compute *read and write sets* that capture how each transaction accessed shared resources (§5.2). Second, we use these sets to identify conflicting transactions and add directed edges to a *serialization graph*, oriented according to the implementation's instrumentation events (§5.3). Third, we compute a topological order of the graph and check the transactions in that order against the sequential reference model (§5.4).

5.2 Read and Write Sets

For each transaction, we compute a *read set* and a *write set* based solely on the operations the transaction performed and their results. These sets contain two types of entries:

- KeyAccess(M, K): a set of keys K in map M. A single-key access uses a singleton set; a range operation uses the set of keys in the range.
- MapPresence(M): the existence of map M itself. This captures conflicts on a map's existence without any specific key being involved. For example, if T_1 creates map M and T_2 successfully deletes M, then T_1 must be ordered before T_2 (otherwise T_2's delete would have failed), even though neither transaction accesses any keys in M. The treatment of MapPresence is subtle, particularly when aborted transactions are involved. Consider: T_1: `CreateMap`(M_1); `Abort` and T_2: `CreateMap`(M_1); `Commit`. These transactions share no keys, yet T_1 must be serialized before T_2: if T_2 went first, T_1's `CreateMap` would have observed M_1 already existing and failed with `MapExists`, in contradiction with T_1's observed execution.

Rule	Edge	Condition
E1: write/write	$T_1 \to T_2$	$C(T_1, P) < C(T_2, P)$
E2: write/read	$T_w \to T_r$	$C(T_w, P) \le \mathcal{R}(T_r, P)$
	$T_r \to T_w$	$\mathcal{R}(T_r, P) < C(T_w, P)$
E3: backup	$T \to B$	if $\forall\, i:\ C(T, P_i) \le \mathcal{B}(B, P_i)$
	$B \to T$	otherwise
E4: happens-before	$T_1 \to T_2$	$\text{round}(T_1) < \text{round}(T_2)$
	$T_1 \to T_2$	same client, T_1 before T_2

Figure 4. Serialization graph edge rules (§5.3).

Two entries *overlap* if they refer to the same map and either (a) at least one is a MapPresence entry, or (b) both are KeyAccess entries and their key sets overlap. Two transactions *conflict* if they contain overlapping entries, at least one of which is a write. Computing precise read and write sets is important: missing a conflict can admit an incorrect serialization, while a spurious conflict can create a cycle in the graph, which the checker would report as a failure even though the execution is correct. We next describe how we order each pair of conflicting transactions.

5.3 The Serialization Graph

The serialization graph contains one node per transaction and one node per backup. Aborted transactions are treated as read-only: their writes are excluded (since they had no visible effect), but their reads are retained.

For each pair of conflicting transactions, we add a directed edge that records which transaction came first. We determine the direction from the partition that holds the map or key they conflict on. Each partition maintains its own write-ahead log whose entries are identified by a *partition sequence number* (PSN). Three PSN-derived values are relevant for the ordering: the *commit PSN* $C(T, P)$ is the PSN at which transaction T's writes are recorded on partition P (a multi-partition transaction receives a separate PSN for each participant); the *read PSN* $\mathcal{R}(T, P)$ is the highest PSN on P whose effects T observed; and the *backup PSN* $\mathcal{B}(B, P)$ is the PSN up to which backup B includes entries on P.

Fig. 4 shows how we use PSNs to orient the edges for each type of conflict. Rule E1 orders writers by their commit PSNs. Rule E2 handles both write–read and read–write conflicts: comparing the writer's commit PSN against the reader's read PSN determines whether the reader saw the write, which dictates the edge direction. Rule E3 states that a transaction is ordered before a backup only if *all* its writes were captured; it is ordered afterwards otherwise. The two conditions are mutually exclusive, so each committed transaction is ordered either before or after a given backup. Rule E4 encodes the happens-before ordering from the test structure (§4.2):

transactions in earlier rounds precede those in later rounds, and a client's jobs are ordered sequentially.

Correctness. Rules E1 and E2 add an edge for every pair of conflicting transactions, oriented by the implementation's instrumentation events. If the resulting conflict graph is acyclic, any topological sort yields a serial schedule that is *conflict-equivalent* to the concurrent execution: for every pair of conflicting operations, the serial order agrees with the concurrent execution on which came first. Non-conflicting transactions may appear in any relative order without affecting correctness. Because conflict-equivalence comes entirely from E1 and E2, the happens-before edges (E4) only restricts the allowed topological orders. A cycle would indicate either a serializability violation in the implementation or an over-constrained checker; in practice, implementation bugs have manifested as replay mismatches rather than graph cycles.

Our graph is essentially the conflict graph from concurrency control theory, extended to handle aborted and lock-free transactions, and backups, as described above. When the order of conflicting operations is known, checking is efficient. The PSNs provide exactly the information needed to determine this order.

5.4 Replay and Validation

We pick any topological order of the graph and replay the transactions in that order, comparing each operation's result against the result observed during the concurrent run. Aborted transactions are replayed only to check that their reads match the reference model's state. After replay, we compare the reference model's final state against the implementation to catch discrepancies not exposed by the operation results.

Partition backups. Backup nodes are ordered in the graph via rule E3. During replay, when a backup node is reached in the topological order, we verify that the backup's contents reflect all writes ordered before it and none ordered after, validating that the backup captures a consistent snapshot across all partitions.

Handling failures. Concurrent reftests inject failures (host crashes, partition relocations, timeouts), which may cause a transaction to return an error to the client yet still commit. For example, a transaction may fail after all participants have durably prepared but before the client learns the outcome: the client receives an error, yet the transaction has logically committed and will be driven to completion in the background. Such transactions must be included in the serialization graph, since their writes are visible to later transactions. We determine a transaction's true outcome by inspecting the durability of its preparation log entries: if all participants durably prepared, the transaction is included; otherwise it is excluded.

Example bug: inconsistent commits. During early development, the serializability checker caught a bug in the two-phase commit protocol implementation. In this protocol, one partition acts as the *coordinator* and drives the prepare and commit phases across the other *participant* partitions. Consider a distributed transaction T coordinated by P_1 with participant P_2. If P_2 prepared successfully but was then reloaded, it would query P_1 for the transaction's status. If P_1 had not yet finished its own preparation, it would consult only its on-disk state (which is populated only after preparation completes) and respond with `Unknown`. P_2 would then abort T. Meanwhile, P_1 would finish preparing, receive P_2's earlier *prepared* response, and commit. The result: T committed on P_1 but aborted on P_2. Because P_1 responded to the client with *committed*, the reference model applied T's writes to both partitions during replay, producing a mismatch against the implementation's state. The fix was to consult the in-memory transaction state when answering status queries, so that a coordinator that has started a transaction but not yet persisted it responds with `InProgress` rather than `Unknown`.

5.5 Lifting to Per-Key Linearizability

Serializability of individual transactions is necessary but not sufficient for correctness of object-level operations. The namespace needs to ensure *per-key linearizability*: for each object key, the operations on that key appear to take effect atomically in an order satisfying read-after-write consistency. The complication is that a single object-level operation (e.g., a `PutObject` that creates internal nodes and a leaf node) may span multiple transactions, so we must show that the *composite* operation linearizes correctly against other operations on the same key. We check this in two phases. First, we build the serialization graph over the individual transactions as above (tagging each with its parent index operation) and find a candidate topological order. Second, we lift this to the object level: each index operation is placed at a *linearization point* derived from its constituent transactions (for example, mutating operations are typically linearized by their last read-write transaction), yielding a sequence of object operations consistent with the transaction ordering. We then replay the operations in this sequence against the namespace reference model, doing a final state comparison between reference and implementation to check correctness.

6 Validating the Control Plane

The control plane manages partition placement across the fleet of index hosts, called *workers*: it monitors worker health via periodic heartbeats, detects failures, relocates partitions from failed workers to healthy ones, and orchestrates graceful deployments and partition scaling. Unlike data path components, this work happens asynchronously in background loops—heartbeat monitors poll workers and submit failover

tasks, bucket monitors scan partitions and submit repair tasks, and a task manager executes those tasks concurrently. In the absence of a request/response boundary to check against, we specify correctness via properties over the states reached during execution.

6.1 Specification

We specify correctness with two classes of properties, both concerning which worker is responsible for each partition.

Safety invariants must hold continuously during execution. The most critical is *no double-loading*: every partition must be loaded on at most one worker at any given time. A violation would allow two workers to independently serve requests for the same partition, leading to data corruption. The implementation enforces this through a permit mechanism: before loading a partition, a worker must acquire a permit from the control plane, which grants one only if no other worker holds a valid permit. Additionally, to prevent a worker that has been declared failed but is still running (a *zombie*) from continuing to serve requests, the control plane *fences* such a worker before granting the permit elsewhere, invalidating its permit so it can no longer serve the partition.

Stability properties hold only once the system quiesces, so unlike safety invariants they need not hold in transient states reached while the control plane is still reacting. Technically they are still safety properties: each asserts that *if* quiescence is reached, the state is correct, not that quiescence is eventually reached (a system that never quiesces is caught separately, see below). Our two stability properties are that every partition is loaded on *exactly* one healthy worker, and that any initiated scaling operation has completed and updated the bucket's partition count.

6.2 Validation Approach

We adapt the methodology from §3—property-based testing with proptest and Shuttle for exploring interleavings. Rather than checking request/response traces against reference models, test scenarios consist of running the control plane under Shuttle with *control operations*, while checking both classes of properties described above. The alphabet of control operations consists of three environment stimuli that change worker state (starting, shutting down, and crashing a worker) and an API call requesting partition scaling. The control plane observes these operations through its monitoring loops and reacts asynchronously.

Checking safety invariants. To check the "no double-loading" property continuously, we implemented a *partition tracker*: a runtime monitor that observes every partition state transition on every worker. The tracker drives a state machine for each partition and validates that every transition is legal.

Checking stability properties. To check stability properties, we first detect when the system has quiesced. We instrument the production code to emit *progress events*, notifying our test harness each time a monitor loop completes a full iteration. We then wait until the task manager has no outstanding tasks, and run one more iteration of each monitor to confirm that no new tasks are submitted. If there are any, the system has not yet settled and we repeat. Once quiescent, we check that the stability properties hold.

There are two challenges we need to address. A system that never quiesces would make this check repeat forever (e.g., if monitors keep generating tasks in a cycle). Shuttle runs each execution with a *depth bound* on the number of scheduling decisions and fails any run that hits the bound, so we do not silently miss livelock or divergence (at the risk of false alarms if the bound is too low). Reaching quiescence is not sufficient for a scaling operation: after the operation completes, we check that the final partition count matches the expected value, so that a scaling operation that quiesced without actually taking effect still fails the check.

Our approach exploits compositionality between control-plane and data path validation. The data path validation described earlier *assumes* the properties established by control plane validation. For example, the data path test harness explicitly unloads a partition before reloading it on a different worker, ensuring no double-loading. In production, the worker holding the partition may crash before it unloads, so the control plane cannot rely on a clean handoff. Instead, it fences the crashed worker, ensuring the partition is no longer served there before loading it on another worker. The control-plane tests validate that this guarantee holds under concurrent failures, deployments, and scaling operations.

7 Persistent Reftesting

The reftests described so far are short-lived: each test starts on a fresh in-memory stack with mocked storage and networking, runs a sequence of operations, and discards it when the test ends. These tests may miss bugs that require long-lived state or span multiple software versions. These include cross-version bugs, where version N+1 misreads data written by version N (say, after an encoding change); bugs that depend on real infrastructure or on state that accumulates over weeks of operations; and bugs in background subsystems such as billing and garbage collection, which run asynchronously from the request path. *Persistent reftests* target these bugs. These tests run continuously in pre-production, exercising the full production stack (real storage, networking, and control plane) and operating on the same test buckets for months, reaching complex states that short-lived tests never do. As with our other reftests, an executable reference model is the oracle, and the tests run as approval steps in the deployment pipeline, so any failure blocks promotion to production.

Because these tests run for months, the reference model must survive host restarts and deployments, a challenge the

ephemeral tests of §4 do not face. In this section, we describe how we address this challenge (§7.1), followed by experience from running these tests in a production pipeline (§7.2).

7.1 Design

A persistent reftest runs indefinitely on a dedicated host pool in a pre-production environment. On each host, a set of tasks shares a queue of test buckets: a task dequeues a bucket, executes one *segment* against it, and re-enqueues it. A segment is a sequence of object operations over the visible subset of the alphabet used for data path reftests (put, delete, multipart upload etc).

Segment execution proceeds as follows. Using proptest, we generate the segment's operations deterministically from a randomly chosen bucket seed and a monotonically increasing *segment counter*, then run them one at a time against the bucket. As described in §3, operations generated by proptest are first concretized using the reference model (e.g., to pick an existing key for a delete). If the operation mutates the bucket, it is first logged to a durable *operation log*; it is then executed against the production bucket, and the status and return value are compared against the reference. Any divergence is reported as a failure.

Once all operations in a segment are complete, we perform two additional checks. First, we do a complete listing of all objects in the implementation bucket, comparing keys, values and metadata (e.g., etags) against the reference model. Second, the harness queries the *ledger*—a record of mutations generated for customer billing—and compares ledger entries against the set of expected mutations recorded in the reference model. This enables us to check the additional property that the data provided to the billing system is accurate.

In addition to visible data path operations, the persistent reftesting framework also optionally triggers certain control-plane operations such as partition relocation or scaling. These operations are invoked *between* segments rather than randomly interleaved within them. Control plane operations are expected to have no impact on the reference model state, so are treated as opaque events that do not need to be tracked in the model.

Model checkpointing and recovery. The reference model is held in memory, so a host restart (e.g., from a code deployment) loses it. Because segments are generated deterministically from a (bucket seed, segment counter) pair, we could in principle rebuild the model by replaying every segment from the start, but with millions of operations per bucket per day these replays would grow without bound. Instead, after each segment we *checkpoint* the model by serializing it durably to S3 and clear the operation log. On restart, a host restores the latest checkpoint and replays the operation log, which covers any operations issued after the checkpoint (the previous run may have crashed mid-segment). Because we log a mutating operation before issuing it, the last logged operation may or may not have taken effect; we determine its outcome by reading back the affected key.

7.2 Operational Experience

Because segments are generated deterministically, the sequence of operations is reproducible, but events outside our control (e.g. timeouts due to real networking issues) are not. When an operation fails unexpectedly, the test framework records the expected and actual responses, the state of the reference model, and recent control-plane events, and a developer uses these to reproduce the issue in an in-memory test, or, when that is infeasible, to root-cause the failure from the logs.

Over time, we also need to support changes to the model representation. For this, we added migration logic so that old checkpoints still load, and when we add a new feature (e.g., object append), we restart the framework and, in some cases, re-checkpoint from a known-good state. Bugs in the model itself do occasionally cause false positives, but we have found these easy to identify, since they tend to cause failures across most test buckets at once, and are typically easy to reproduce on the in-memory stack. For example, when we extended reftests to cover *hybrid* nodes (keys that are both an object and a directory prefix, so that inserting objects `X` and `X/Y` makes `X` a hybrid node), we hit exactly such a false positive. The reference model's delete logic was too aggressive: deleting a hybrid key also removed the in-progress multipart uploads nested beneath it, which the implementation correctly kept. The failure appeared across buckets at once and reproduced immediately on the in-memory stack; the fix was to stop the model from discarding a deleted key's subtree while it still held uploads.

Each bucket carries a *health state* that governs whether it is eligible for testing. A successful segment marks a bucket as *healthy*; a transient error (such as a network timeout or throttling) marks it *unavailable*, and the harness retries it later. Once failures exceed a threshold, the bucket is marked *corrupted*, a terminal state that removes it from testing until an engineer investigates. This keeps a single bad bucket from blocking the suite or raising spurious alerts, while still ensuring that genuine data-integrity failures are not missed.

The operation log was a later addition. Initially we relied on segments being *idempotent* under replay and kept no log; conditional operations (added after launch) broke that assumption, so we now log each operation before performing it.

Example bug: incomplete listing under throttling. Persistent reftests caught a bug in `ListObject`, the operation that returns a bucket's keys one page at a time. Each page ends with a *continuation token* that tells the next call where to resume. Because the namespace is a tree, a listing traverses it in order, and the token records how far the traversal has reached, including whether it has finished descending into a

given directory. A corner case arises when the last traversed node is a hybrid node, so a listing must both return the object and descend into its children. A listing exits early either when it fills a page or when it is throttled partway through. On a throttled exit, we had a bug in how we generated the token: it could mark a hybrid node's subtree as already listed when it was not, so the next call skipped listing all the children under that subdirectory. Finding this bug required both a throttle in the middle of a listing and enough accumulated objects for a page boundary to fall exactly on a hybrid node. The in-memory reftests, including longbake, did not inject throttling errors, so they never exercised the throttled-exit path that triggers this bug. The fix was to mark the hybrid node as unexplored on early exit, forcing the resumed listing to explore its children.

8 Experience

This section describes our experience validating S3 Express during development, launch and after 2.5 years of being in production. We report on how ownership of the validation artifacts transferred from the formal methods team to the engineering team (§8.1), quantify the validation effort and bugs found (§8.2), summarize lessons learned (§8.3), and discuss limitations of our approach (§8.4).

8.1 Ownership Handoff

The formal methods team joined during the early design phase of S3 Express, before implementation began, and developed the reference models, serializability checker, test harnesses, and persistent reftest framework alongside the implementation. After the launch of S3 Express in November 2023, the formal methods team transitioned ownership of the reference models and test infrastructure to the engineering team, while continuing to maintain Shuttle and the serializability checker. Engineers had peer-reviewed all models and test code during development, so they were familiar with the codebase. For the first post-launch features, the formal methods team helped with code reviews, but within a year the engineers became self-sufficient. Since launch, engineers have released several new customer-facing features (conditional writes, object append, and atomic object rename) as well as performance optimizations such as finer-granularity locking and improved lock ordering to eliminate the overhead of deadlock detection.

Most of the framework transferred without difficulty, but we ran into two points of friction. First, the constraints used to build the serialization graph (§5.2) have to strike just the right balance: too few admit an incorrect serialization, and too many create spurious cycles. This is less of a concern going forward, since the transaction library API is stable. Second, injecting timeout failures with Shuttle requires a mechanism for cancelling tasks in a runtime with no notion

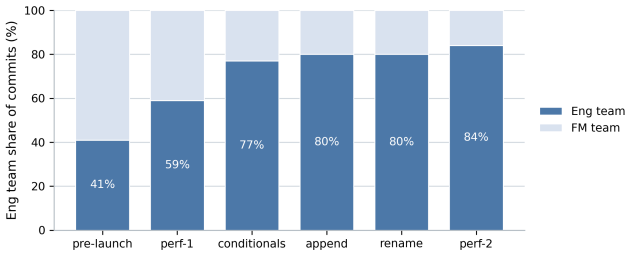


Figure 5. Validation framework authorship over time.

Implementation		Validation	
Production code	259K	Reference models	4.8K
Unit/Integration tests	214K	Reftest harness	35.6K

Figure 6. Lines of code by component.

of wall-clock time, and the exact logic for triggering cancellations has been tricky to get right. We are currently exploring extensions to Shuttle to model time more directly, which would allow us to simplify the timeout injection machinery.

Figure 5 quantifies the ownership transfer over time for the namespace validation, measured by commits to the validation framework (reference models and test infrastructure only, not implementation code). Each bar represents a development phase, showing the split between engineering team (dark) and formal methods team (light) contributions. The engineering team's share of commits grew from 41% pre-launch to 59% for the first performance improvements, then to 77–80% for feature work (conditional writes, append, atomic rename), and finally to 84% for recent performance work, demonstrating successful ownership transfer. The transaction library's specification has remained stable, so its validation framework has not required significant updates.

8.2 Metrics

Code size. Figure 6 shows the lines of code for the implementation and the reftesting framework. The reference models total just 4.8K lines, less than 2% of the 259K-line implementation. The test harness and reftesting infrastructure (including the serializability checker and test orchestration) add 35.6K lines, roughly 14% of the implementation. Together, the entire validation framework—models, test harness, and reftests—is 40.4K lines, or 16% of the implementation code. This modest overhead validates the system's most complex concurrent behaviors. The reference models stay compact by using simple, sequential data structures (§4.1).

Bugs found. Figure 7 summarizes 76 bugs caught by reftesting before reaching production. This excludes bugs found during local development, which developers routinely catch before committing. The left table shows bugs by testing method. Because tests run in stages, these counts show the

Testing Method	Bugs
In-memory reftests	20
Shuttle tests	46
Persistent reftests	10
Total	**76**

Bug Type	Bugs
Transaction Correctness	12
Concurrency	15
Error & Edge Cases	16
State Management	3
Total	**46**

Figure 7. Bugs caught.

marginal value of each layer: in-memory sequential reftests caught 20 bugs that survived existing unit and integration tests, Shuttle-based concurrent reftests found 46 that survived the in-memory reftests, and persistent reftests found 10 that survived both prior layers. The table on the right categorizes the 46 Shuttle bugs by type: transaction correctness (12), including the inconsistent commit bug from §5; concurrency issues (15) such as races, deadlocks, and resource leaks; error and edge cases (16); and state management (3).

8.3 Lessons Learned

A key contributor to our success was that the formal methods team was deeply involved in S3 Express from the very beginning, a common pattern across automated reasoning projects at AWS. This early involvement let us shape specifications before they were enshrined in code, and develop reference models as the code was written rather than after the fact. Because our reference models are executable, they could also serve as mocks for components that did not yet exist. For instance, the namespace team could start testing code before the transaction library was ready, using the library's reference model. Making tests serve as promotion blockers in the deployment pipeline then ensured models and implementation stayed in sync as the code evolved.

We deployed our tests in stages: sequential reftests on developer desktops for fast feedback, in-memory Shuttle tests before merge, overnight longbake tests before pre-production deployment, and persistent reftests running continuously for weeks in pre-production. This staging helped both velocity and maintainability. Shallow bugs are caught early and cheaply, so the slower, more expensive stages mostly run against code with fewer bugs. And because we have separate tests in each stage (rather than one monolithic test framework), tests are smaller and easier to understand and maintain.

A surprise for us has been that engineers have come to regard formal methods as an accelerator rather than a source of friction. Formal methods are often assumed to work the other way, buying assurance at the cost of development speed. While our reftests do add resources and time during testing, engineers have found that catching bugs before commit gives them the confidence to move quickly.

8.4 Limitations and Open Challenges

Our approach has three main limitations. The first limitation concerns the *properties* we check. We check functional correctness, not quantitative properties such as latency and throughput. Even within functional correctness, we deliberately leave some behavior unspecified: the reference models do not cover error cases such as resource exhaustion or device failures (§4.1), checking only that a failed operation leaves state unchanged. Our validation also targets internal APIs rather than the external translation layer, so bugs in request parsing, authentication, or authorization fall outside its scope.

The second limitation concerns *fidelity* to the production environment. Our test environments trade realism for control: in-memory reftests mock storage and networking, so they miss bugs that depend on real hardware behavior, such as bit corruption. Persistent reftests run on real infrastructure, but in exchange cannot systematically explore concurrency and injected failures the way in-memory tests can.

The third limitation is that our testing is *incomplete*. Shuttle executions are bounded in length, and although persistent reftests accumulate state over months, the reachable state space is far too large to explore completely. More fundamentally, randomized testing offers no completeness guarantee: we cannot tell which states and interleavings remain unexplored. Bugs have in fact escaped our checks. In the cache-miss handler bug (§4), our reftests exercised the buggy code path, but our specification was too weak to catch the failure, treating all errors as acceptable under injection and so masking a permanently unavailable block.

Two challenges remain where we see opportunities for future research. The first challenge is measuring test coverage: randomized testing cannot easily tell us which states remain unexplored. To address this, we are building tools that collect domain-specific coverage metrics: developers annotate interesting code paths (e.g., which two-phase-commit states a transaction visits), and after a test run we summarize which were reached. Unlike generic metrics like line coverage, this tracks interesting *semantic states*, such as whether we exercised a scenario where one participant prepared while another aborted. These metrics help engineers identify gaps in test generation, but they remain a proxy for test adequacy; techniques that bound the probability of missing a bug for a given testing budget would be valuable. The second challenge is gaining confidence that the models are correct: a reference model is only a useful oracle if it faithfully captures the intended behavior, yet today we establish this only through manual peer review. Ideally we'd like proofs that the models satisfy their intended properties, but proofs have historically been too labor-intensive to be practical. AI-assisted theorem proving is changing this, and we are currently exploring the use of the Lean prover to formalize our models and prove them correct.

9 Related Work

Formal verification of systems. Formal verification has addressed storage correctness at several layers, from file-system crash consistency in FSCQ [13] and Yggdrasil [34], through distributed protocols and concurrent crash-safe systems in IronFleet [20] and Perennial [11], to the concurrent journal and key-value store verified by GoJournal [12] and VeriBetrKV [19]. SquirrelFS [27] and PoWER [26] apply crash-consistency verification to Rust systems. These approaches provide stronger, machine-checked guarantees than our tests, but require annotation and proof maintenance as code evolves [3]. We trade completeness for reference models the development team can maintain.

Formal methods in industry. Formal methods have long been used in industry. At Amazon, TLA+ has been used to model-check distributed-system designs [31], and P, a language for modeling asynchronous distributed systems [16], was used to validate S3's strong read-after-write consistency [30]. Brooker and Desai survey the range of techniques used at AWS: model checking, property-based testing, and deterministic simulation [8]; Cauli et al. report a similar range used at Huawei Cloud [10]. A recurring challenge is keeping specifications connected to evolving code, which can otherwise go stale [10, 31]. Our reference models are instead executable and share the implementation's API, and reftests in the deployment pipeline prevent model-code divergence.

Lightweight formal methods. Property-based testing generates random inputs and checks the resulting behavior against executable properties [14]. Quviq QuickCheck extends this approach to stateful models, demonstrated on an industrial telecom implementation [5]. Jaber et al. use predicate abstraction to systematically explore requests and states for black-box conformance testing of S3 APIs [22]. In controlled concurrency testing, the same test runs under many execution schedules, and any failing schedule is recorded for replay. CHESS searches this space systematically, while Coyote and Shuttle use randomized strategies to scale this approach to production code [15, 29, 33]. Our prior work on ShardStore combined both techniques with executable reference models to validate a single storage node [7]. We extend that work to a large distributed system, which must tolerate failures, coordinate through an asynchronous control plane, and preserve correctness as it evolves across versions.

Consistency checking. Jepsen exercises distributed systems under faults and analyzes client-observed histories; Elle infers isolation anomalies from such histories, and Cobra uses an SMT solver to search for an acyclic dependency graph consistent with the observed history [24, 25, 37]. Our work instruments the implementation to orient conflict edges directly, yielding a polynomial-time check without constraining workloads. It also handles production behaviors these black-box tools exclude: range reads, deletes, and reads from aborted transactions (Cobra drops aborted transactions and range queries; Elle recovers write order from append-only workloads, but our operations overwrite and delete keys). Line-Up similarly checks concurrent executions against sequential ones with stateless model checking, but targets deterministic linearizability of a single component rather than distributed transaction serializability [9].

Control-plane testing and cross-version correctness. Sieve perturbs controllers' state views [35], while Acto drives sequences of desired states [18]. Anvil formally verifies that controllers implement eventually stable reconciliation: under its environment assumptions, a stable desired state is eventually reached and maintained [36]. Our bounded tests instead explore process interleavings, check safety continuously and stability at quiescence, and reject runs that do not quiesce within Shuttle's depth bound; finite runs do not prove liveness. T2C derives runtime semantic checkers from tests [28], while Zhang et al. target upgrade failures with testing and static analysis [38]. Our persistent reftests instead use independently maintained reference models as oracles across successive deployments.

10 Conclusion

A central question when we began this work was whether our validation approach would survive the departure of the formal methods team. After 2.5 years in production, the engineering team actively maintains and extends the validation approach, with their share of framework commits growing from 41% to 84%. They have independently delivered atomic rename, append, and conditional operations while maintaining all models and test infrastructure. Three design choices enabled this sustained ownership: reference models written in Rust against the same APIs as the implementation, reftests wired in as mandatory pipeline approval steps, and validation structured as composable layers. Along the way, this approach caught 76 documented bugs before they reached production. We are now exploring the use of Generative AI to reduce the effort required to apply these techniques to new systems while moving toward full functional verification of both reference models and production code.

Acknowledgments

We thank the anonymous SOSP reviewers and our shepherd for their constructive feedback. We are grateful to James Bornholt for many technical discussions that shaped our approach, and to both James and Don Porter for their detailed comments on earlier drafts. We thank S3 leadership, in particular Andy Warfield, Seth Markle, Sidharth Bajaj, Rishabh Animesh, Edward Naim, and Mustafa Hakim, for their support of this work. Finally, we thank the entire S3 Express team for their collaboration throughout.

References

[1] 2020. Tokio: An asynchronous runtime for Rust. https://tokio.rs

[2] 2023. proptest. https://crates.io/crates/proptest

[3] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. 2012. Large-scale formal verification in practice: A process perspective. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 1002–1011.

[4] Thomas Arts and Laura M. Castro. 2011. Model-based Testing of Data Types with Side Effects. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang* (Tokyo, Japan) *(Erlang '11)*. Association for Computing Machinery, New York, NY, USA, 30–38. doi:10.1145/2034654.2034662

[5] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang* (Portland, Oregon, USA) *(ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. doi:10.1145/1159789.1159792

[6] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (Oct. 2019), 28 pages. doi:10.1145/3360591

[7] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. doi:10.1145/3477132.3483540

[8] Marc Brooker and Ankush Desai. 2025. Systems Correctness Practices at Amazon Web Services. *Commun. ACM* 68, 6 (June 2025), 38–42. doi:10.1145/3729175

[9] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 330–340. doi:10.1145/1806596.1806634

[10] Claudia Cauli, Timo Lang, Shuo Chen, Sebti Mouelhi, Xin Jin, Subhajit Bandopadhyay, Xusheng Chen, Yazhi Feng, Haoze Song, Linhua Tang, Zhenli Sheng, and Ananth Shrinivas Srinath. 2026. Lessons Learned from Incorporating Formal Methods in Huawei Cloud Reliability. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 2223–2238. doi:10.1145/3767295.3803626

[11] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. doi:10.1145/3341301.3359632

[12] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. doi:10.1145/2815400.2815402

[14] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[15] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. 2023. Industrial-Strength Controlled Concurrency Testing for C# Programs with Coyote. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 433–452. doi:10.1007/978-3-031-30820-8_26

[16] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 621–631. doi:10.1109/ICSE.2007.68

[18] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112. doi:10.1145/3600006.3613161

[19] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 6, 17 pages.

[20] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

[21] John Hughes. 2016. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane.* Springer International Publishing, Cham, 169–186. doi:10.1007/978-3-319-30936-1_9

[22] Nouraldin Jaber, Dongyun Jin, Bernhard Kragl, Enrico Magnago, Gustavo Petri, Thorsten Tarrach, and Serdar Tasiran. 2026. High Fidelity Models for Large Scale Stateful Services (Operational Systems). In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 1699–1714.

[23] Daniel Jackson and Jeannette Wing. 1996. Lightweight Formal Methods. *IEEE Computer* 29, 4 (1996). https://www.cs.cmu.edu/%7Ewing/publications/JacksonWing96.pdf

[24] Kyle Kingsbury. 2014. Jepsen. https://jepsen.io

[25] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 268–280. doi:10.14778/3430915.3430918

[26] Hayley LeBlanc, Jacob R. Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. 2025. PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 839–857.

[27] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. 2024. SquirrelFS: using the Rust compiler to check file-system crash consistency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 387–404.

[28] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. 2025. Deriving Semantic Checkers

from Tests to Detect Silent Failures in Production Distributed Systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 19–38.

[29] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[30] Vishwas Narendra, Serdar Tasiran, and Ankush Desai. 2021. Amazon S3 Strong Consistency. https://www.youtube.com/watch?v=B0yXz6EeCaA

[31] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4, 66–73. doi:10.1145/2699417

[32] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. doi:10.1145/322154.322158

[33] Amazon Web Services. 2020. Shuttle. https://github.com/awslabs/shuttle

[34] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 1–16.

[35] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159.

[36] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666.

[37] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 4, 18 pages.

[38] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 116–131. doi:10.1145/3477132.3483577