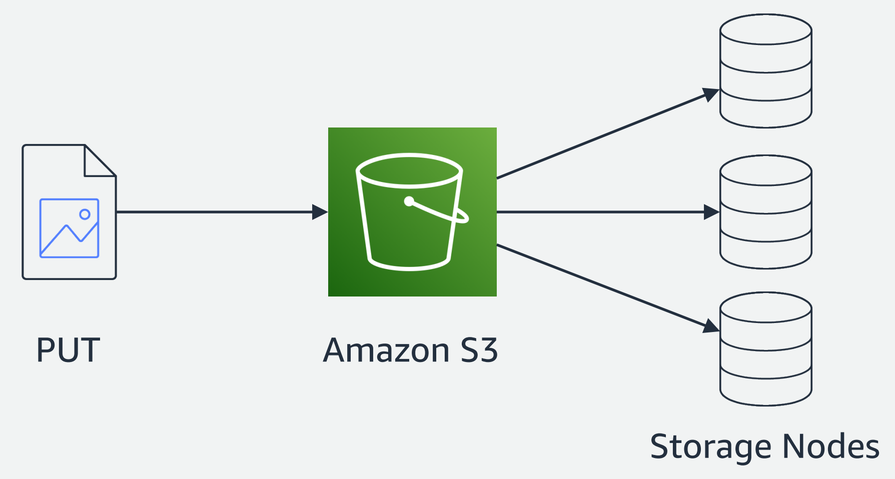
Using lightweight formal methods to validate a key-value storage node in Amazon S3

James Bornholt[1,2] Rajeev Joshi[1] Vytautas Astrauskas[3] Brendan Cully[1] Bernhard Kragl[1] Seth Markle[1]
Kyle Sauri[1] Drew Schleit[1] Grant Slatton[1] Serdar Tasiran[1] Jacob Van Geffen[4] Andrew Warfield[1]
[1]Amazon Web Services [2]The University of Texas at Austin [3]ETH Zurich [4]University of Washington

ShardStore: S3's new storage node

Amazon S3 is a cloud object storage service offering elastic storage with extremely high durability and availability. **ShardStore** is a new single-host key-value store we're building to serve as a "storage node" within S3 to durably store object data. We replicate each object's data across multiple storage nodes:



Like other production storage systems, ShardStore's Rust implementation code is difficult to get right, because it combines a number of complexities:

- A **soft-updates**-based crash consistency protocol
- Extensive **concurrency** including background operations and integrity checks
- Optimizations to support **efficient IO** (scheduling, coalescing, etc.)

Formal methods for storage systems

We've seen recent successes in using *formal methods* to validate the correctness of storage systems in the face of complexities like these:

- FSCQ [Chen et al, SOSP'15] showed it was possible to formally prove the correctness of a file system, but at a 10–20× lines of code overhead
- Yggdrasil [Sigurbjarnarson et al, OSDI'16] used an automated "push-button" verification style to reduce this overhead, but required careful co-design with the verifier to make automation tractable
- GoJournal [Chajed et al, OSDI'21] extended these verification guarantees to a concurrent journaling system, but again at 10–20× overhead

We took inspiration from these efforts, but our key goal was to **integrate formal methods into our engineering practice**, so we sought lower overheads.

Lightweight formal methods to validate ShardStore with low overhead

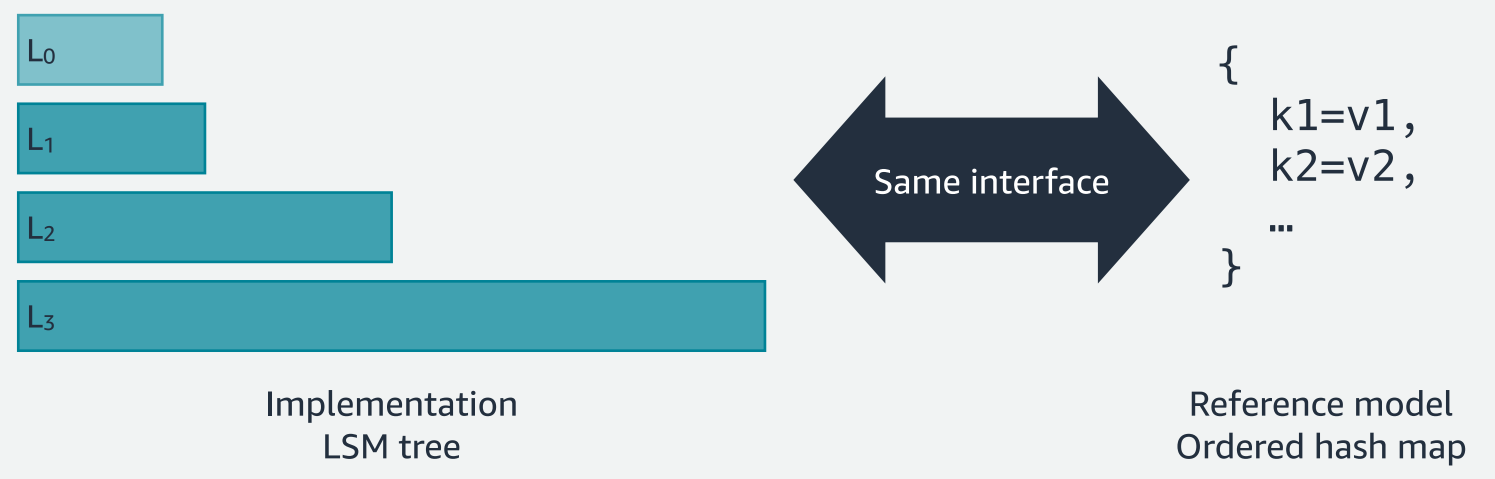
We developed a **lightweight formal methods** approach to validate deep correctness properties of ShardStore in an automated fashion. Our approach has three elements:

1. We develop executable **reference models** as specifications that live alongside the implementation code
2. We apply a suite of **automated conformance checking tools** to validate that the implementation code respects the specification
3. We implement mechanisms to **measure the effectiveness of these checks** to ensure future changes to ShardStore are still correct

In return for being lightweight and automated, we accept weaker correctness guarantees than full formal verification—we can still miss bugs. But we gain the ability for *future* engineers to validate their changes without expensive new verification work by formal methods experts.

Reference models make for simple, maintainable specifications

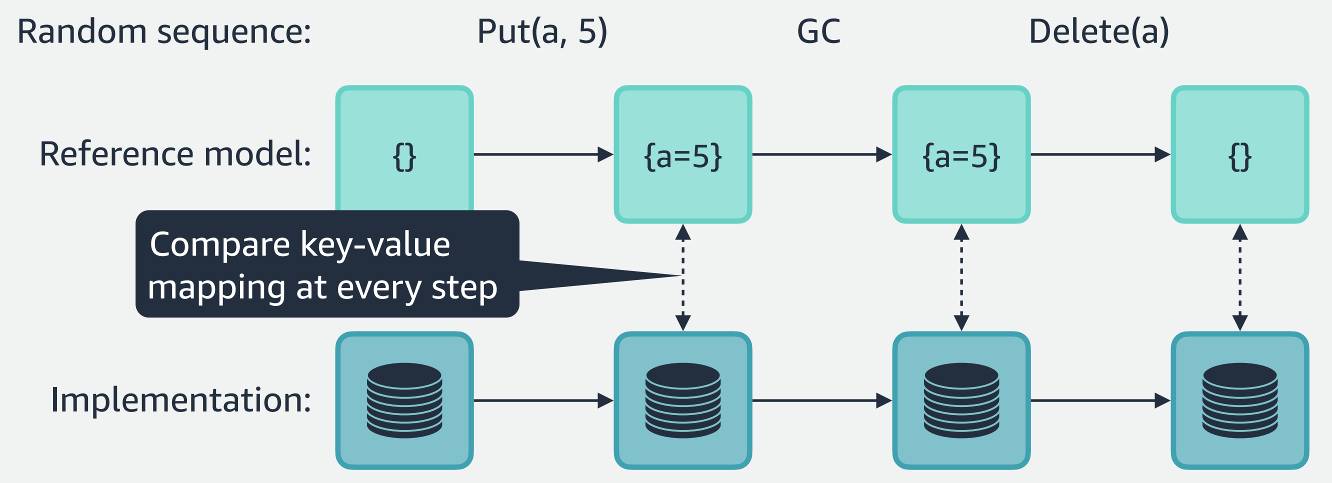
We choose to write our specifications as executable reference models—small pieces of code that offer the same interface as the component they specify, but without concern for implementation efficiencies. These specifications are easy to update over time because they're written in Rust like the implementation, and are reused for other purposes (e.g., as mocks for unit testing).



Automated "pay-as-you-go" checkers validate the code on every commit

Rather than a one-size-fits-all tool, we decompose our correctness property into smaller pieces and check each with a different tool.

For **crash consistency** we check that the implementation *refines* the reference model using property-based testing to test random traces:



For **concurrency** we check that the implementation is linearizable with respect to the reference model using stateless model checking, which executes a piece of code many times with a different thread interleaving each time.

All of these tools are "**pay-as-you-go**"—they scale with compute, so we can run them at small scale on engineer laptops to test local changes, or at massive scale in the cloud before deployments to gain confidence in correctness.

The result is that we can validate the correctness of every ShardStore commit, preventing bugs from reaching production, or even reaching code review.

We're hiring!

The S3 Automated Reasoning Group applies formal methods to build correct, secure, durable, and highly available distributed storage systems.
We're hiring for both **full-time and intern positions**.
Contact us at `s3-arg-jobs@amazon.com`.