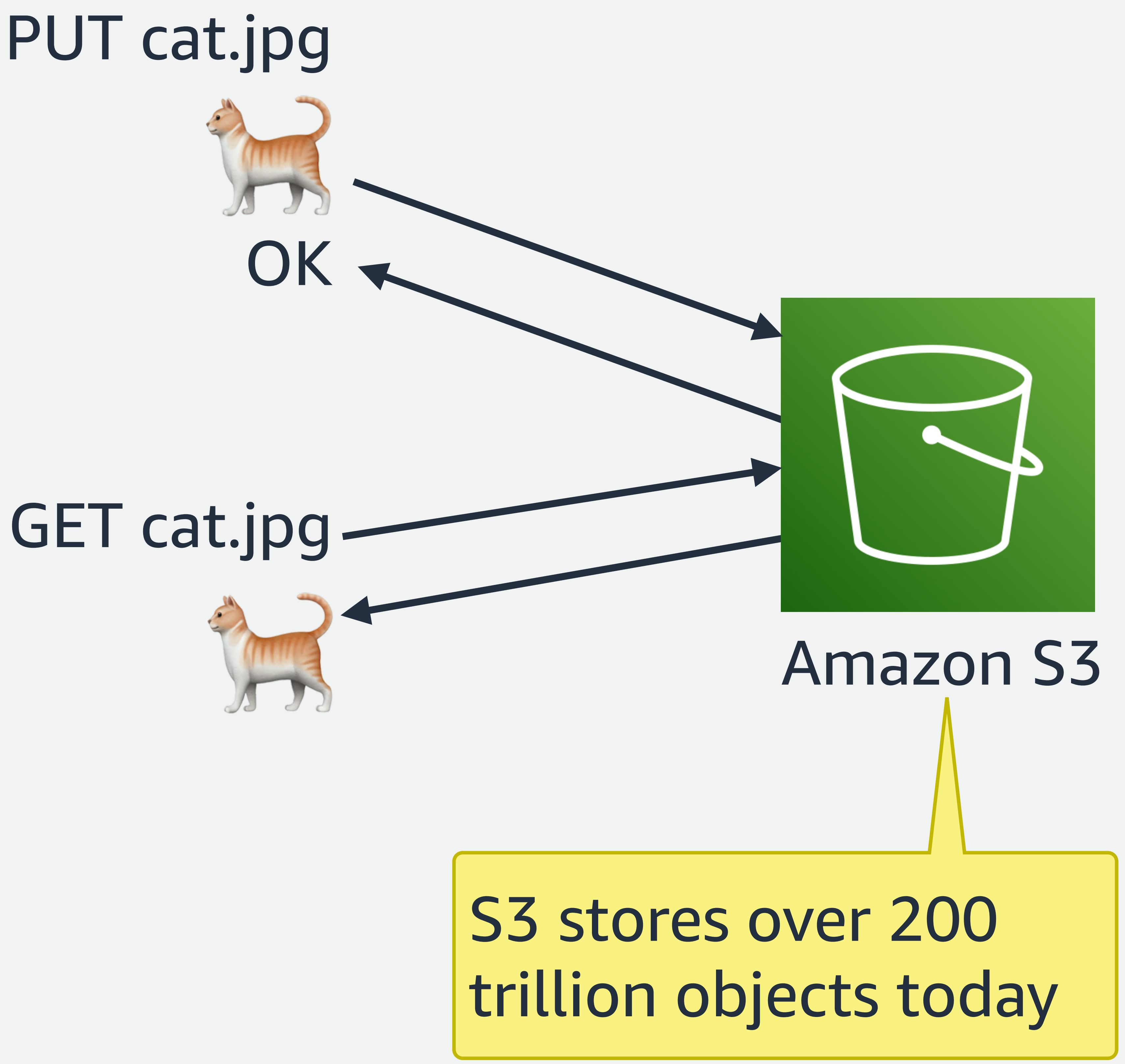
Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3

James Bornholt
Amazon Web Services & The University of Texas at Austin

joint work with Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andy Warfield

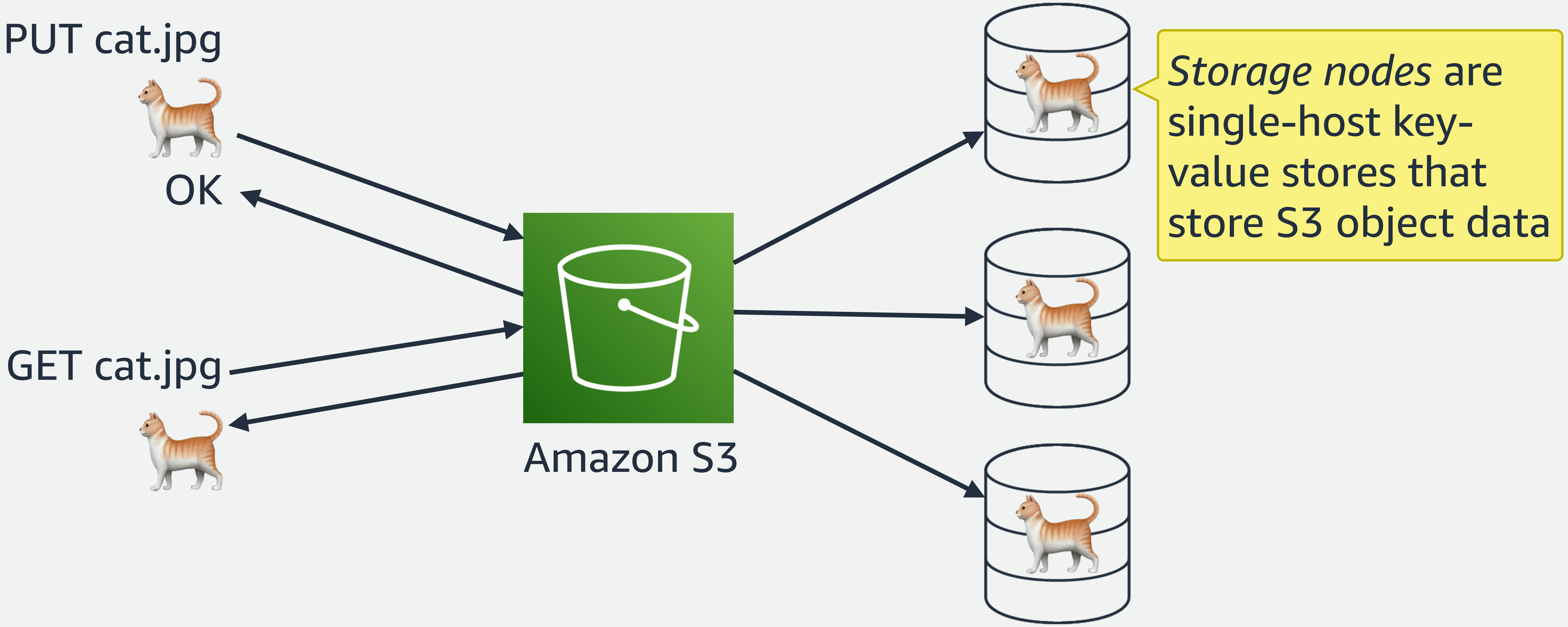
Object storage on Amazon S3

- Amazon S3 is an *object storage service* (PUT, GET), also known as a *key-value store*

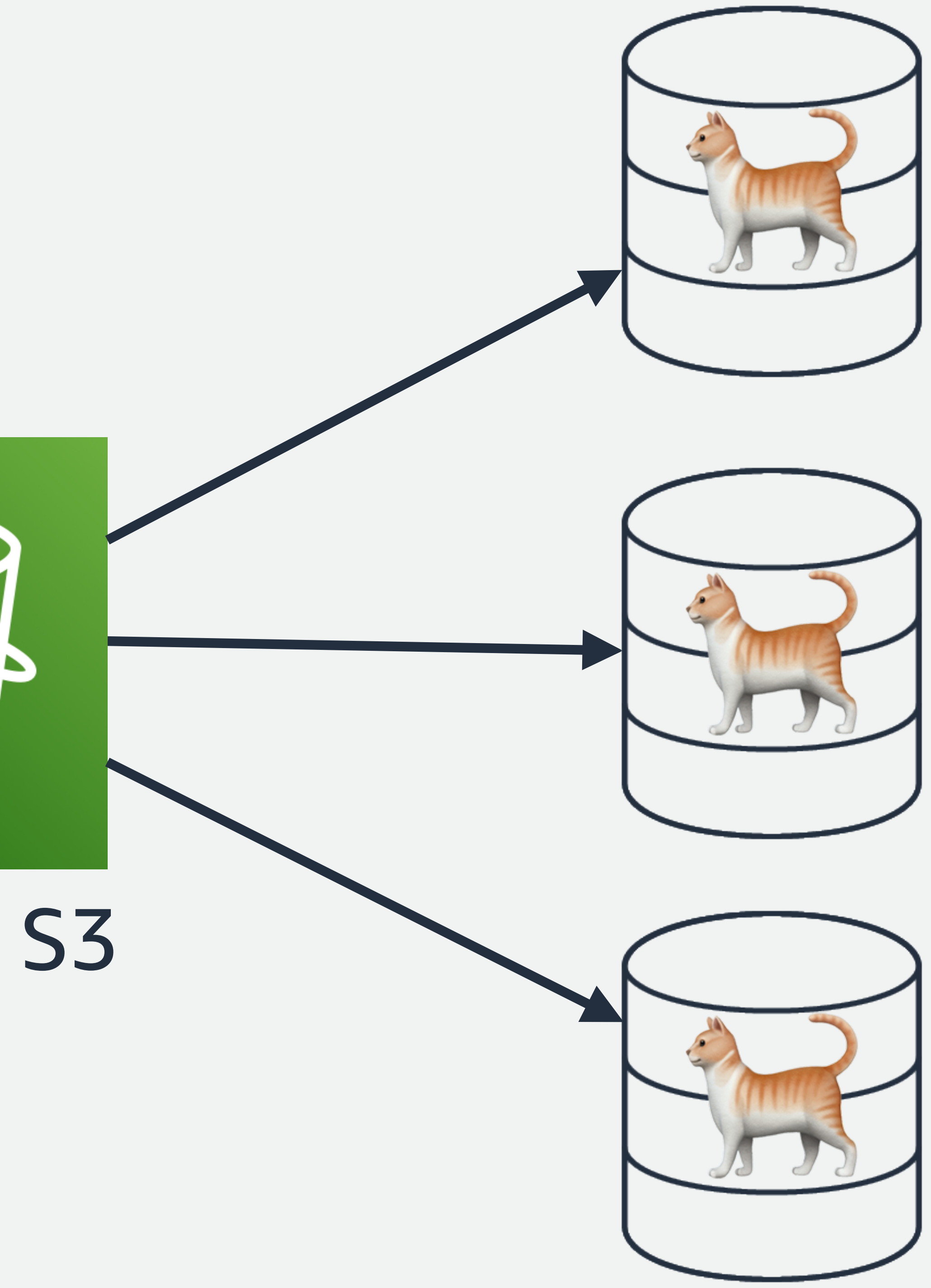


Object storage on Amazon S3

- Amazon S3 is an *object storage service* (PUT, GET), also known as a *key-value store*



S3's new ShardStore storage node



- Currently deploying **ShardStore**, a new storage node written in Rust
- 45k lines of code, ~100s of PBs in 2021
- Implementation is complex:
  - a log-structured merge tree…
  - …with support for zoned (append-only) storage
  - …soft updates for efficient crash consistency
  - …a bunch of fancy concurrency
  - …

What makes a storage system correct?

How can we validate correctness continuously?

What makes a storage system “correct”?

- Functional correctness — PUT, GET, DELETE, etc all do what we want them to do
  - “GET returns the right data”

What makes a storage system "correct"?

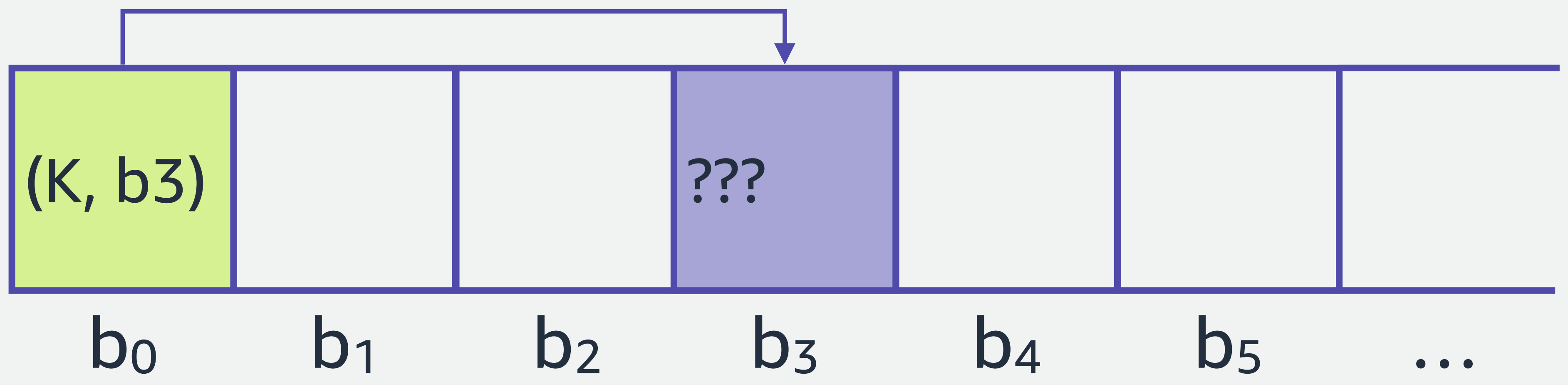
- Functional correctness — PUT, GET, DELETE, etc all do what we want them to do
  - "GET returns the right data"
- Crash consistency — disk is in a valid state after a crash

Put(K, V):

```
write(b0, (K -> b3));
```

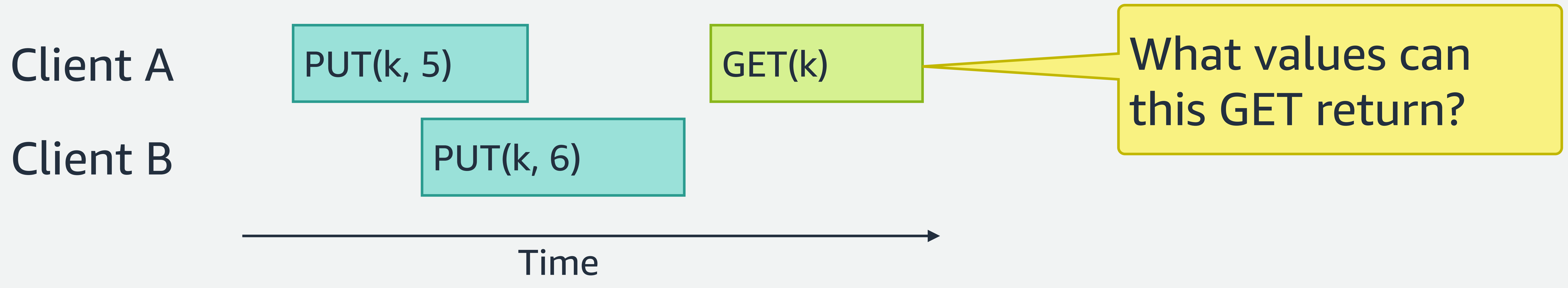
Crash

```
write(b3, V);
```



What makes a storage system "correct"?

- Functional correctness — PUT, GET, DELETE, etc all do what we want them to do
  - "GET returns the right data"
- Crash consistency — disk is in a valid state after a crash
- Correctness under concurrency (aka consistency, but not the same as crash consistency!)



What makes a storage system correct?

How can we validate correctness continuously?

We need *lightweight* formal methods

- Want to validate deep properties of the implementation
- Whatever we do needs to be maintainable in the long run
  - Our goal: future changes to ShardStore require no involvement from FM experts
- Integrate into a large project: 45k lines of Rust, weekly deployments, etc.

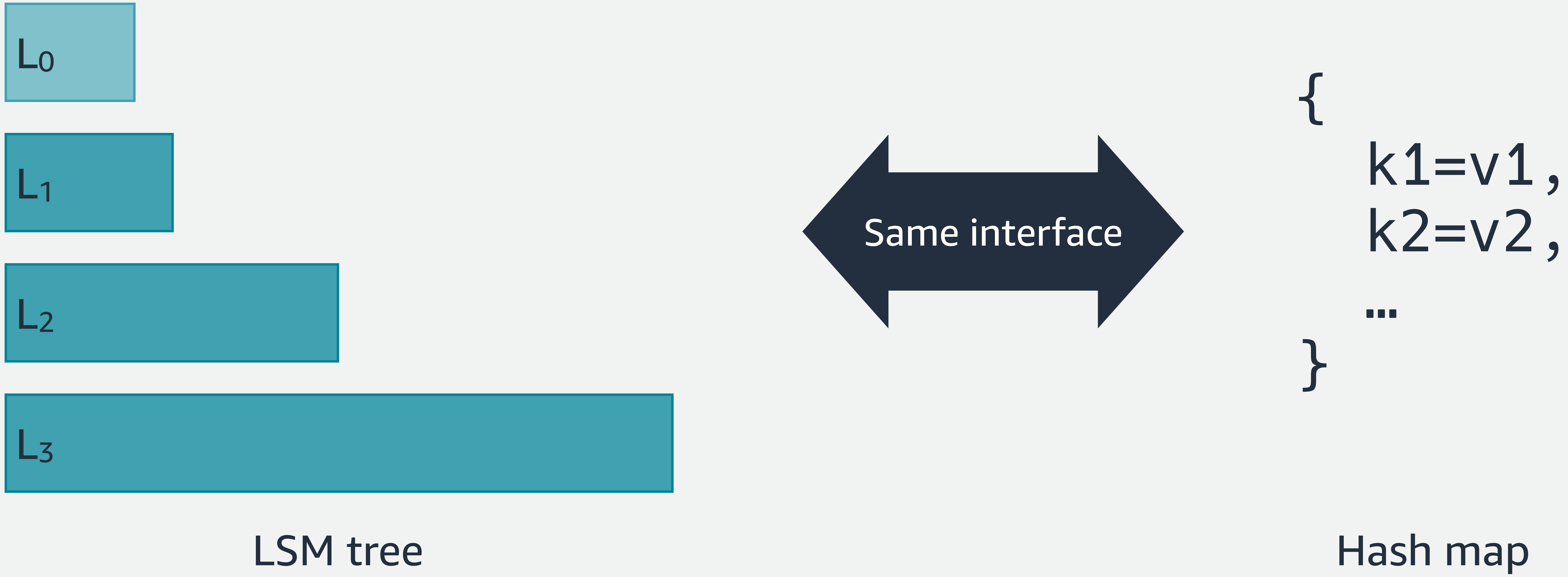
Lightweight formal methods

1. Executable *reference models* as specifications
2. Automated tools to check implementations against models
3. Coverage tools to track effectiveness over time

In return for being lightweight and automated, we accept weaker correctness guarantees than full formal verification

Writing *reference model* specifications

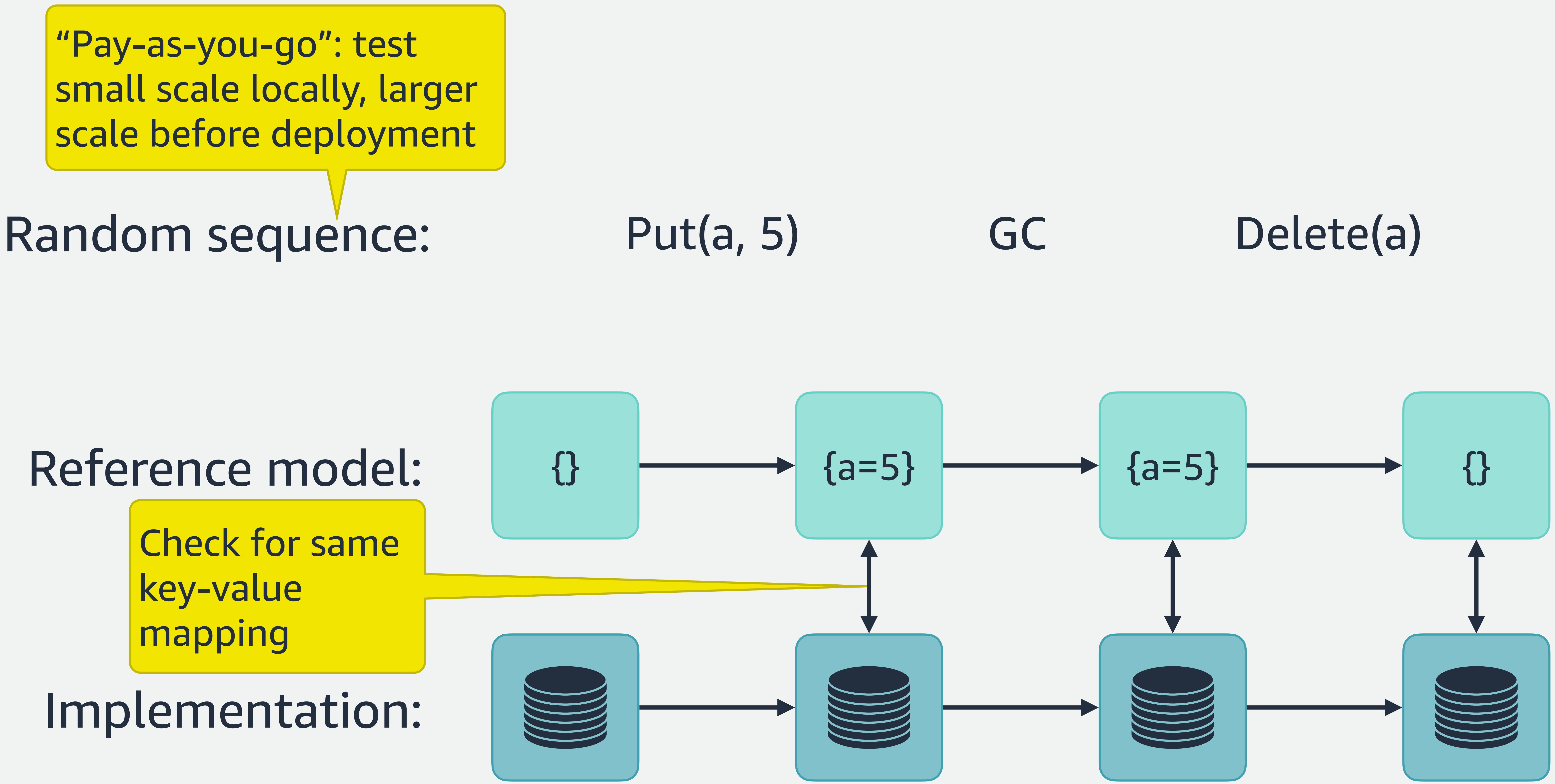
- Small, executable specifications, written in Rust
- Stored/reviewed/committed alongside the code



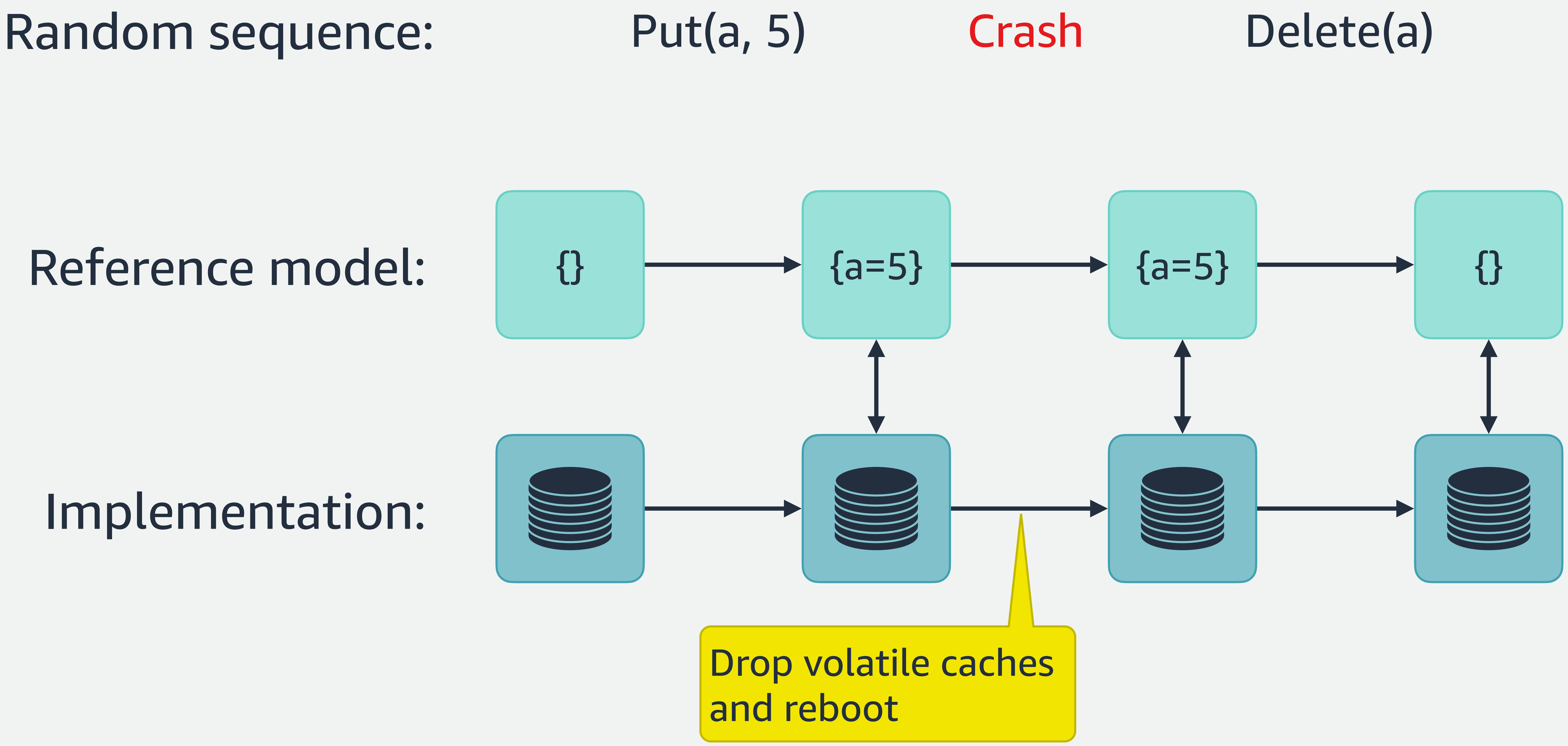
Correctness properties

- Decompose correctness into three parts and check each separately:
  - Functional correctness: refinement of the reference model
  - Crashes: refinement against a weaker reference model
  - Concurrency: linearizability against the reference model

Conformance with property-based testing



Conformance with property-based testing



Conformance with property-based testing

- Randomized testing can miss bugs
- Arrange biases to reduce this risk where we can

```
Put(key: u64, value: [u8])
Get(key: u64)
```

~0% chance we generate a key we already put

- Use coverage data to monitor code we're missing
- Apply heavyweight tools where it makes sense (serialization, undefined behavior, …)

Checking concurrent behavior

- We need a lightweight way to validate the behavior of our concurrent code
  - Multiple customer requests, background tasks, disk IO, etc.
- *Stateless model checking* is a way to test concurrent code by exploring potential interleavings
  - Automated — it’s just a push-button model checker
  - Lightweight — in Rust, it just looks like a unit test
  - Usable — “feels like cheating”

Checking concurrent behavior

```
shuttle::check(|| {
    // Set up some initial state
    let index = PersistentIndex::new();
    for (key, value) in &[...] {
        index.put(key, value);
    }

    // Spawn concurrent operations
    let t1 = thread::spawn(|| index.compact());
    let t2 = thread::spawn(|| index.reclaim());
    let t3 = thread::spawn(|| {
        for (key, value) in &[...] {
            assert_eq!(index.get(key), value);
        }
    });
})
```

Shuttle is a stateless model checker for Rust

Test interleavings of background tasks with GETs and check values are always correct

Experience with FM in production

- Automated lightweight tools prevent issues from even reaching code review
- Maintainable in practice:
  - 20% of model code by non-FM experts
  - 1/3rd of engineers have written their own new models/checks
  - In production for > a year
- “Pay-as-you-go” and continuous validation makes FM viable in a rapid production engineering process

What makes a storage system correct?

How can we validate correctness continuously?

SOSP 2021: “Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3”