READ MODE · ANSWER FIRST

Learn distributed systems correctness with TLA+, PlusCal and Jepsen

Model safety, liveness and stale-leader races in TLA+ and PlusCal then test a real cluster with Jepsen-style fault histories.

CHEAT SHEET · 8 ESSENTIAL IDEAS

The whole story in 8 lines

Specify the contract, explore every small interleaving and attack the real system until evidence matches the model.

1. Parallel rails make conflict and progress legible at the same time.
2. Interval geometry exposes real-time order and impossible atomic placements.
3. A sequence strip and paired durable rows expose four common commit bugs.
4. Program counters moving through lanes make enabled actions and crash points tangible.
5. State branching exposes nondeterministic choices and the finite path to a counterexample.
6. Identical timelines make the fence repair visible as a contrast rather than a narrated claim.
7. Concurrent clients, a fault glyph and an append-only history show testing as evidence collection.
8. The history physically shrinks into a minimal witness before the invariant verdict appears.

STAGE 1 OF 10

Setup

We are going to follow one segment commit for partition P7 at sequence 42. First, let us learn the ideas we need to decide whether that commit is correct.

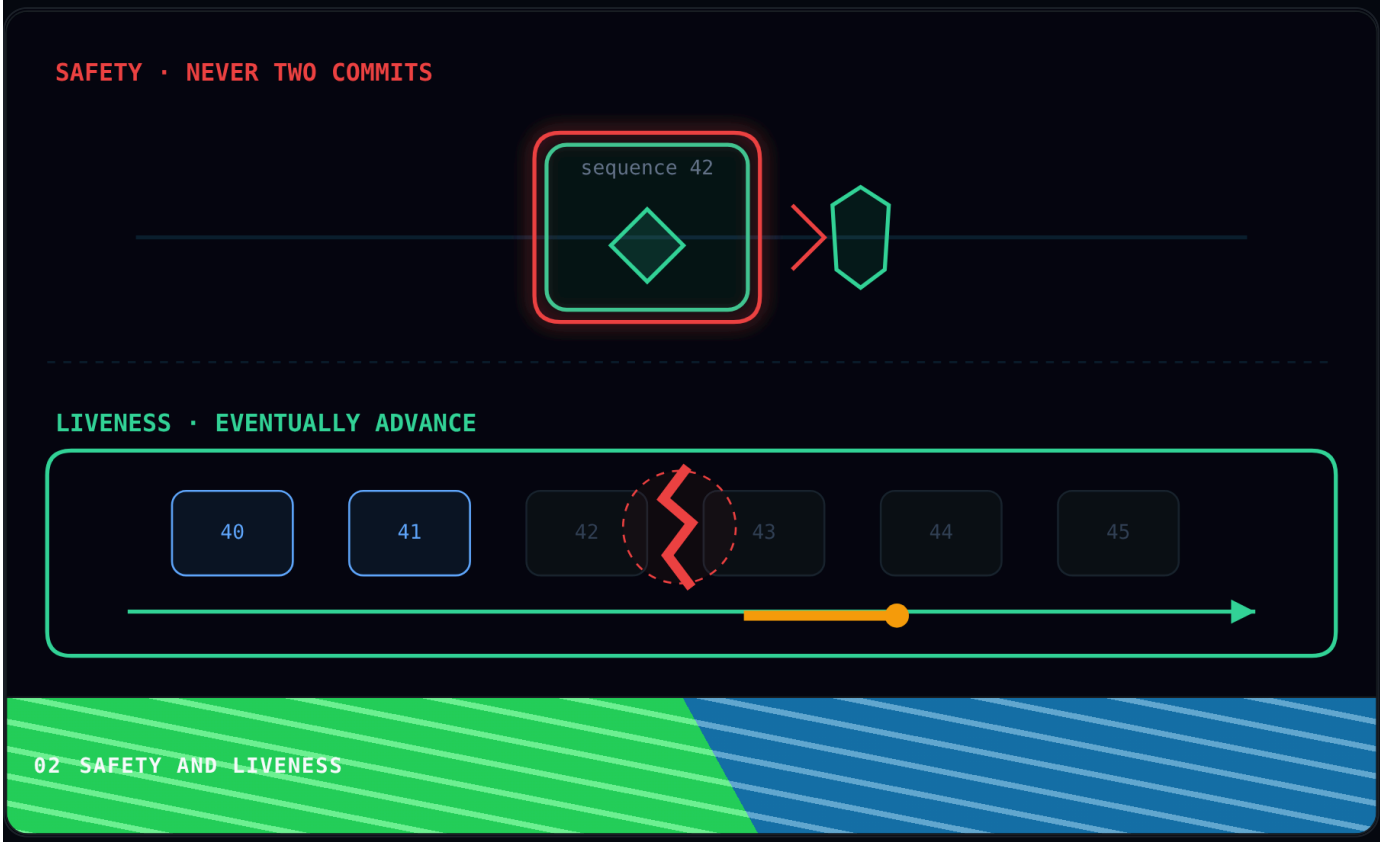
Safety means the system never reaches a forbidden state and liveness means useful work eventually continues when the required conditions return.

An invariant is a rule that must hold in every reachable state and linearizability asks where each operation could appear to happen all at once.

PlusCal describes the actors and their steps and tLA+ checks possible behaviors, while Jepsen records what a real cluster does during failures.

We will move from promises to exact rules, then to small exhaustive models and hostile real tests. Let us start with safety and liveness.

Safety and liveness



The top rail asks what must never happen, while the bottom rail asks whether healthy ingestion can keep moving because correctness needs both promises.

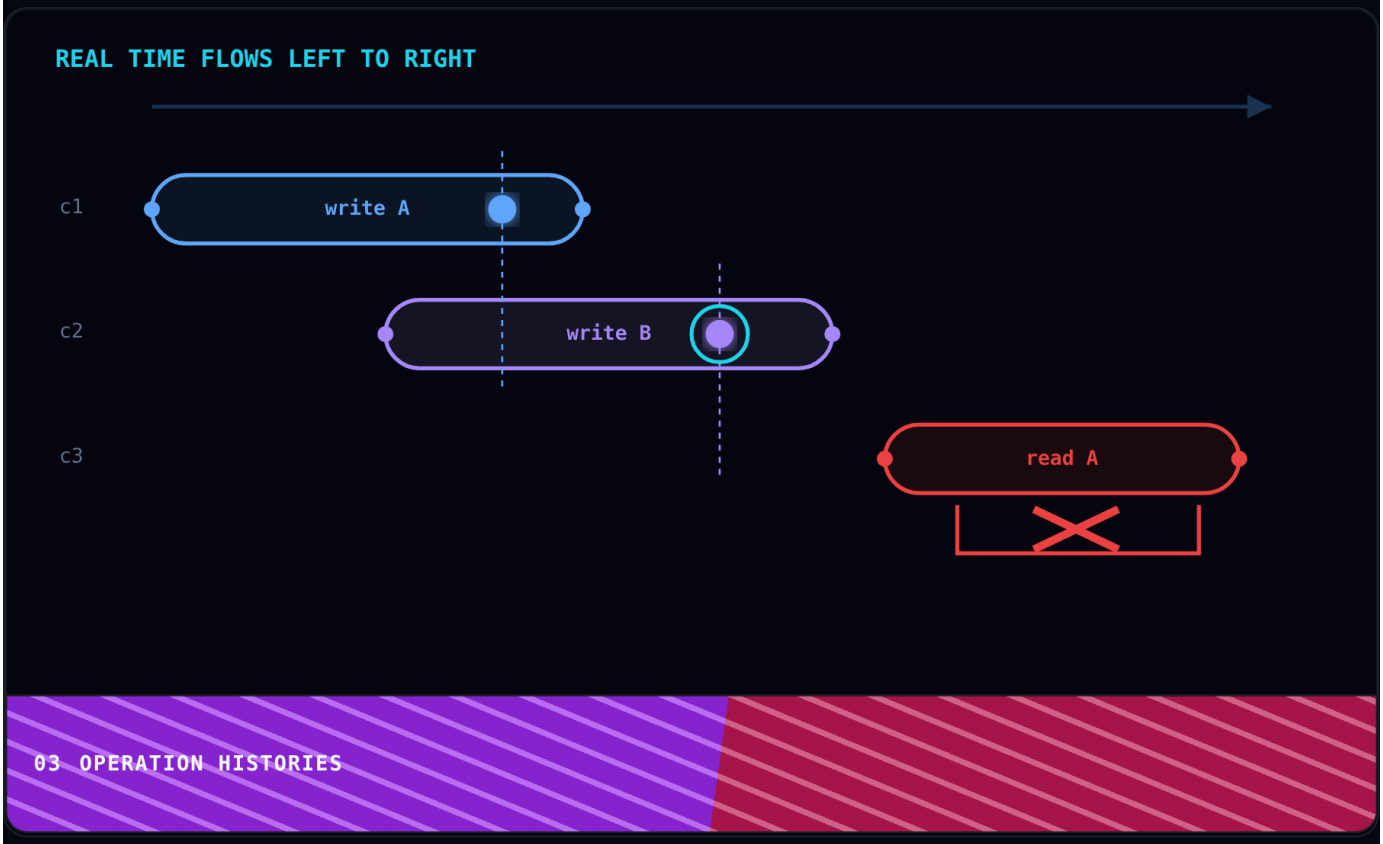
Segment A already fills the only commit slot for sequence 42 and safety says a conflicting segment must never enter that same slot.

Now a network partition stops acknowledgements, so is the system allowed to wait without breaking safety?

Yes, because progress can pause while the commit slot stays unique, and switching Network to Healthy lets the liveness rail move again.

Safety survived the wait, and liveness returned with the healthy network, so next, we will place operations on a real-time history.

Operation histories



03 OPERATION HISTORIES

Now that safety and liveness are separate, let us ask when each operation takes effect. Every bar shows one call from its start to its response.

Calls that overlap may be ordered either way. A linearization point chooses one instant inside each bar when that call appears to happen.

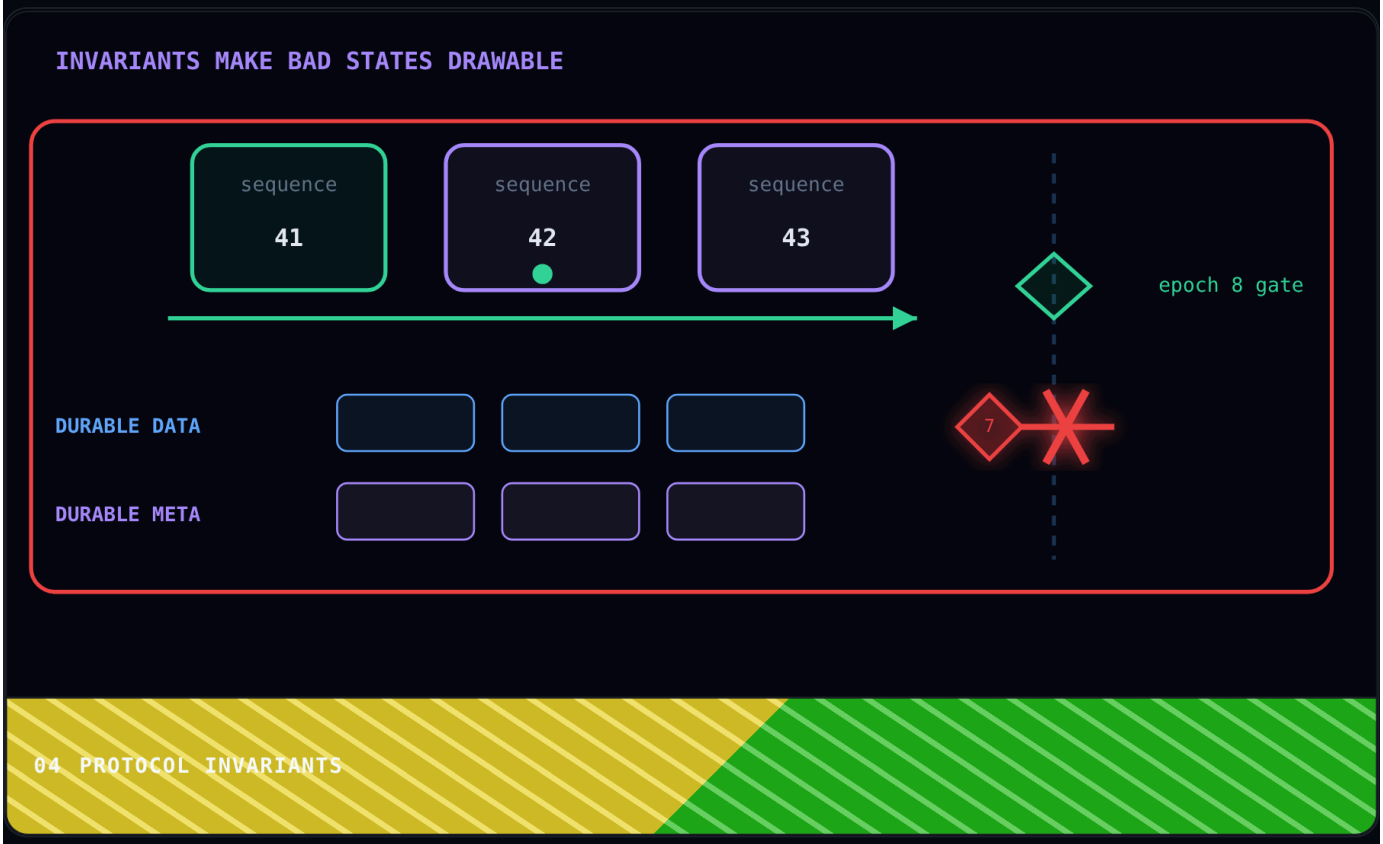
The read returns a value from the wrong candidate, so can you place its atomic point anywhere without breaking the history?

You cannot place that point in the broken History, but switching History to Valid lets every operation fit one legal real-time order.

Linearizability orders one object in real time, while serializability orders whole transactions, so next, we will turn protocol rules into invariants.

STAGE 4 OF 10

Protocol invariants



04 PROTOCOL INVARIANTS

A legal history needs exact rules and here, we can see the sequence order, the durable data row, and the durable metadata row.

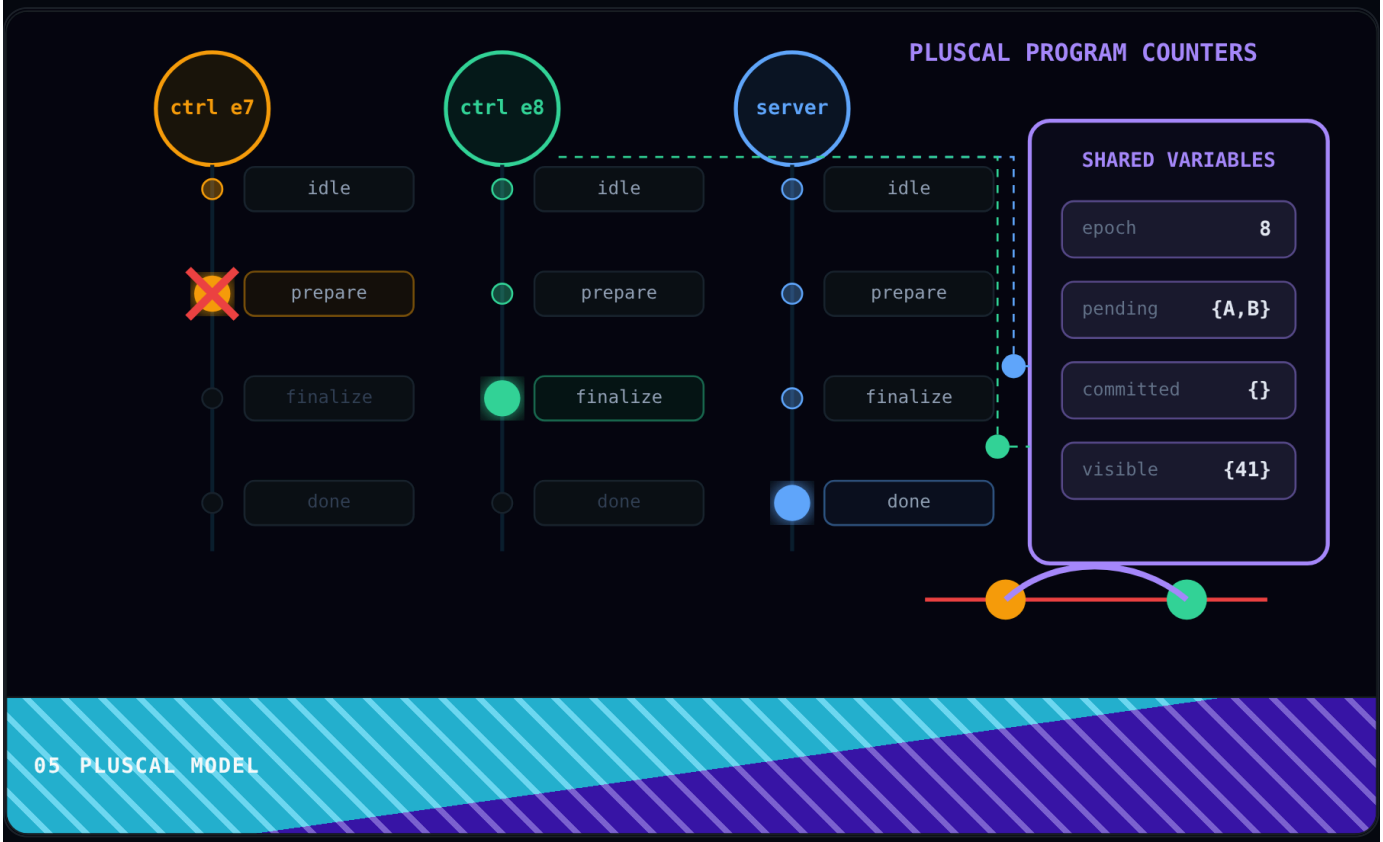
Sequence 42 has one commit slot and both its data and metadata must be safely stored before the commit seal can appear.

Sequence 43 waits behind 42, and the epoch gate checks leader authority, so which rule will the selected bug break?

Try each Bug and watch Conflict duplicate a slot, Torn write split data from metadata, Reorder skip a dependency and Stale leader fail the epoch check.

A good invariant names a forbidden state without telling developers how to code the fix. Next, we will model the actors in PlusCal.

PlusCal model



05 PLUSCAL MODEL

We now need a machine-checkable model. PlusCal gives the controller and server their own program counters while they share a small set of state variables.

The old controller prepares seg-A in epoch 7 and we keep only the shared facts that can change whether the protocol is correct.

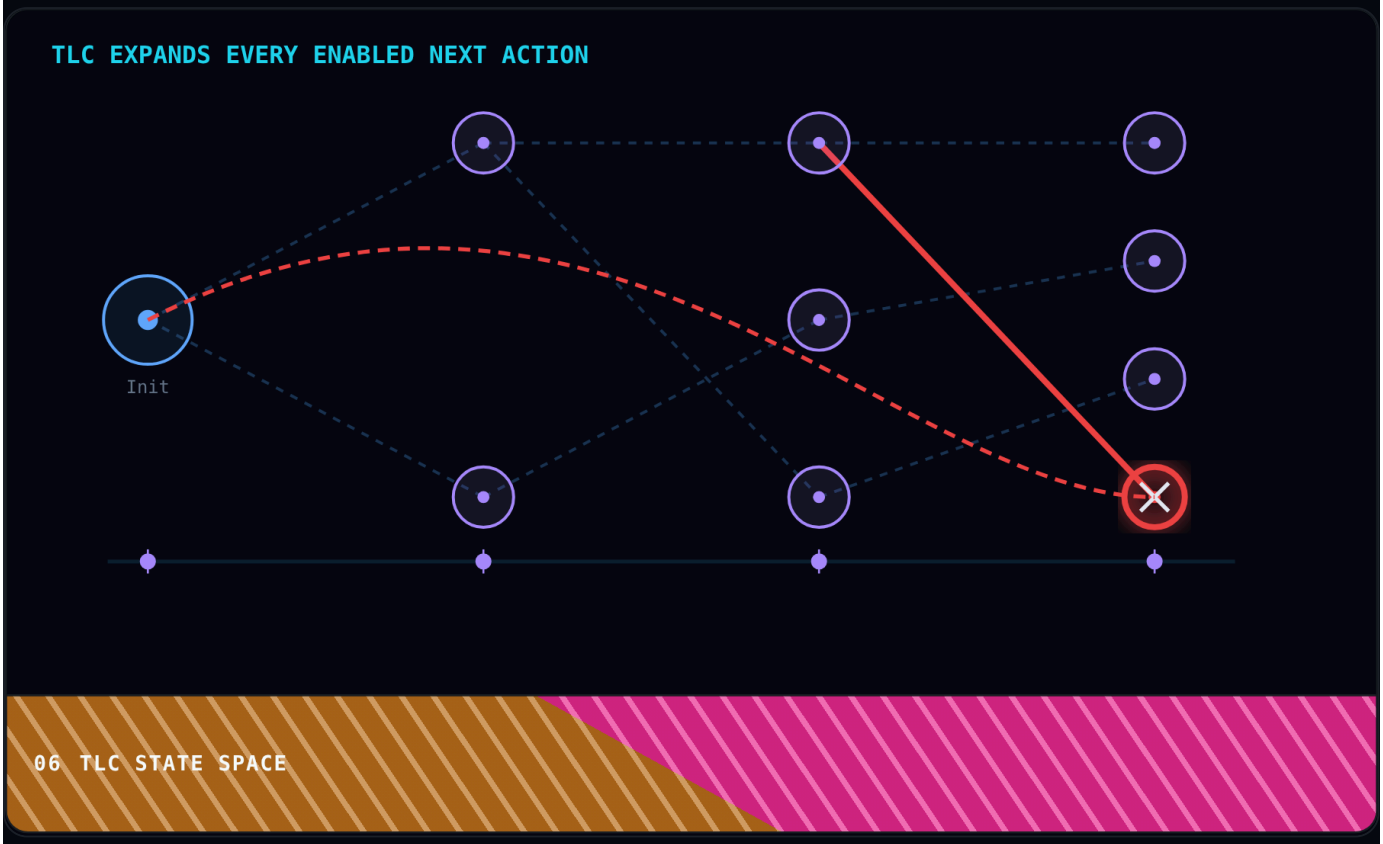
Set Failure to Crash and watch the old controller stop after prepare while the system remains able to elect a new controller.

Epoch 8 prepares seg-B while the delayed finalize from epoch 7 still exists, so which enabled action should run next?

Either finalize can run next and that uncertain choice is the exact interleaving we want TLC to explore.

PlusCal labels become TLA+ transitions that change state variables, so next, TLC will follow every enabled choice in this finite model.

TLC state space



PlusCal gave us the possible transitions and TLC starts at Init and tries every enabled Next action instead of choosing just one schedule.

The first branches are elect, deliver, and retry. Shared beginnings stay together, so the graph does not copy the same states again and again.

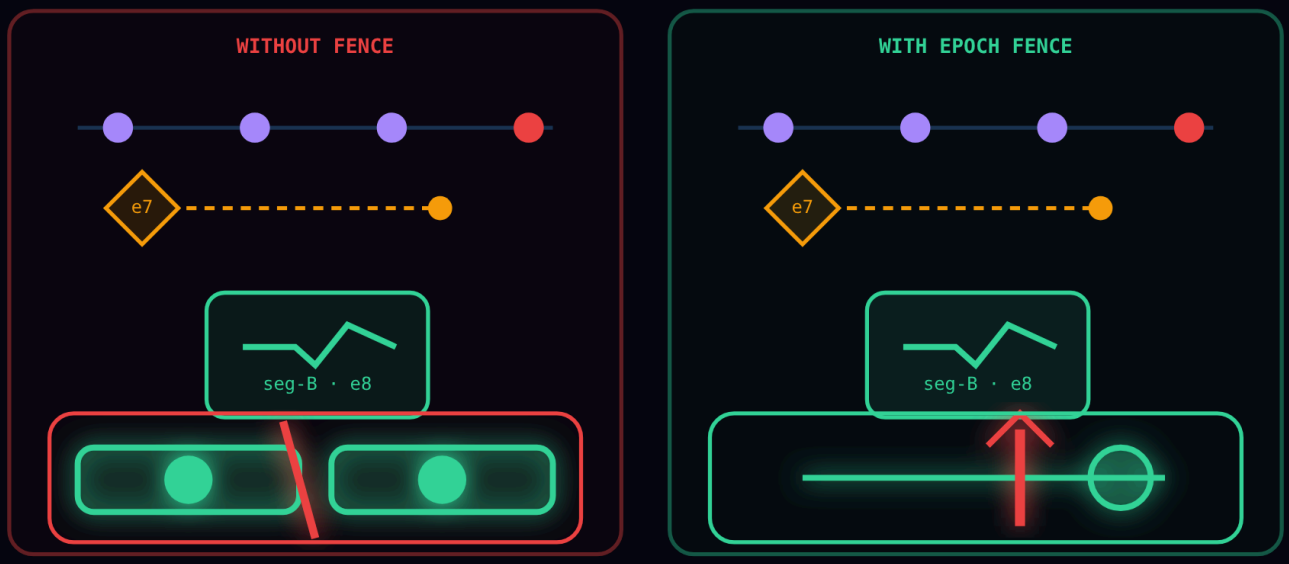
Increase Depth to reveal more schedules, but can any branch reach a state where two different candidates are committed?

Yes, because a red leaf appears when the stale finalize follows the epoch 8 commit, and TLC saves the exact path into that bad state.

Model checking gives us evidence from reachable states instead of guesses about timing, so next, we will read the counterexample and fix the authority boundary.

Counterexample

ONE TRACE · TWO COMMIT BOUNDARIES



07 COUNTEREXAMPLE

TLC found a bad state and let us replay the same events in two lanes so we can compare the broken protocol with the repaired one.

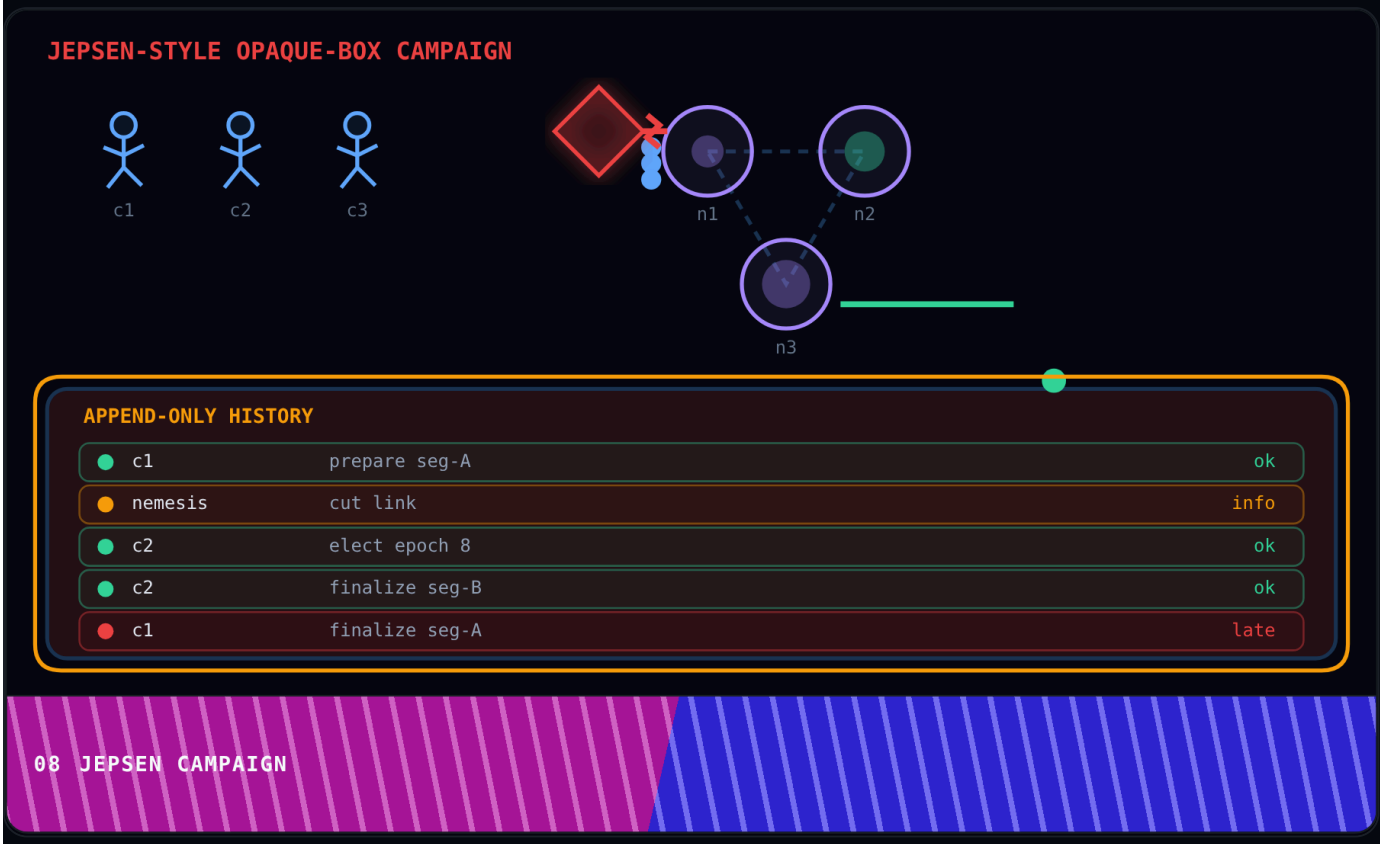
Epoch 7 prepares seg-A and then loses leadership and in both lanes, its delayed finalize message is still traveling.

Epoch 8 commits seg-B for sequence 42, so what happens when the old epoch 7 finalize finally arrives?

Without a fence, seg-A becomes a second commit. With a fence, the epoch gate rejects the same old message before it becomes visible.

The check belongs at the final, irreversible boundary. An old leader may compute, but it cannot finalize. Next, we will hunt for this race in a real cluster.

Jepsen campaign



The small model exposed a race, so Jepsen-style testing now runs the real controller, servers and storage while clients send operations.

The generator overlaps prepare, leader election, and finalize calls and change Clients and more operation lanes appear.

A separate nemesis injects trouble and use Fault to cut links, crash a leader, delay a message, or deliver one twice.

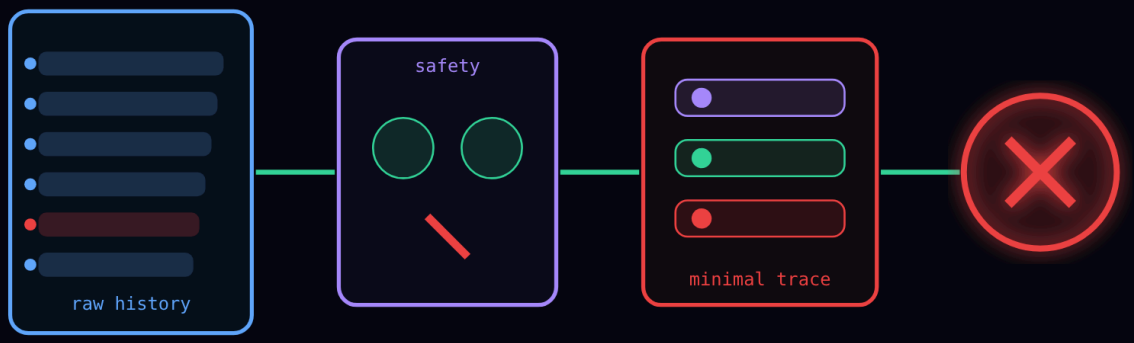
Every call and response goes onto the history tape, so will the system accept the delayed epoch 7 finalize?

The tape records epoch 8 committing seg-B, then the stale finalize arriving late and we now have real evidence for the checker to judge.

Jepsen cannot prove correctness, but it can expose real behavior during failures, so next, we will compare this history with the abstract contract.

History oracle

OBSERVED HISTORY → ABSTRACT VERDICT



09 HISTORY ORACLE

Jepsen gave us a raw history and the checker first translates real API events into the simpler operations and states used by our model.

The checker searches legal atomic orders for linearizability, evaluates invariants for safety and asks whether recovery eventually continued for liveness.

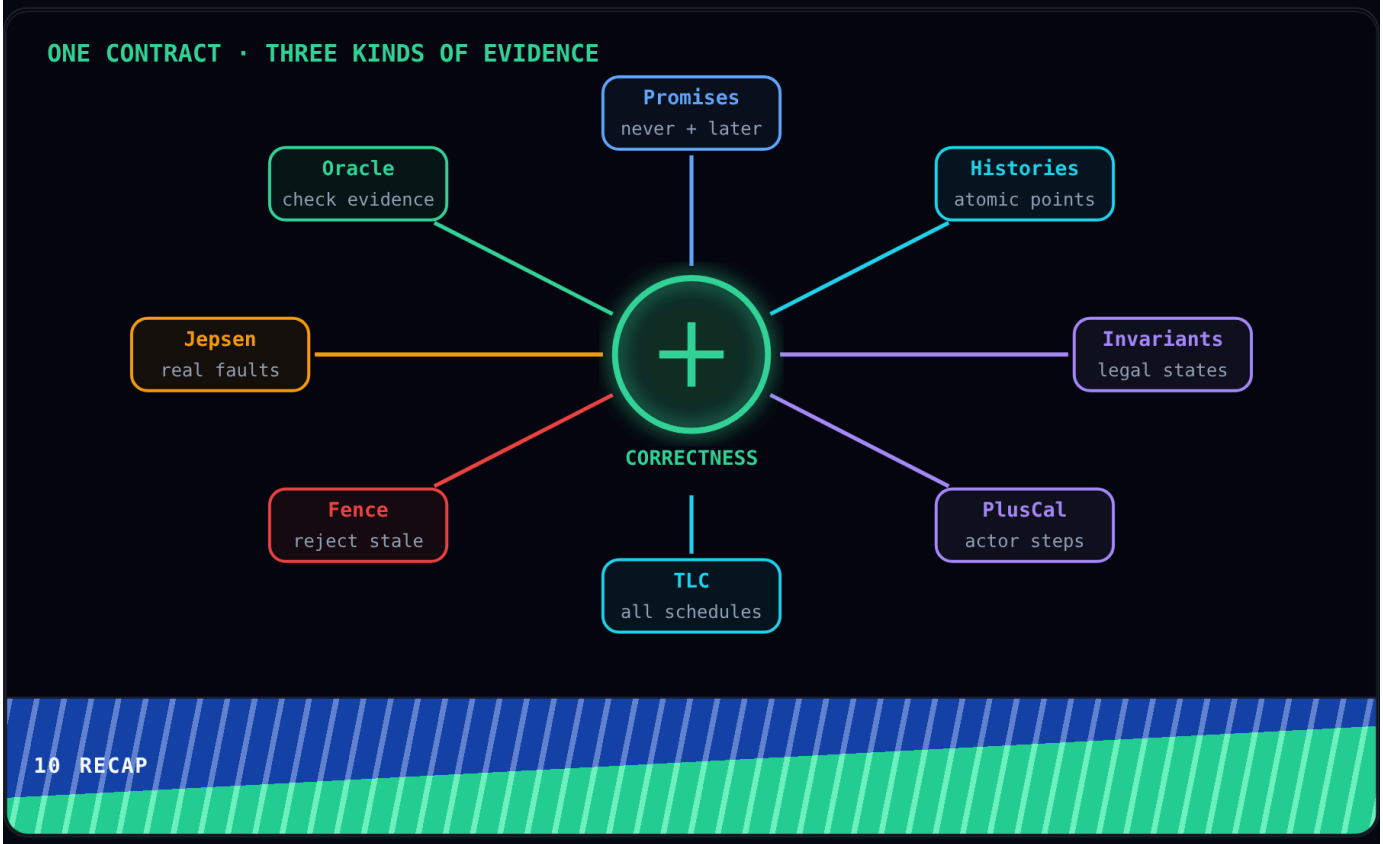
Choose a property in Check, but can the noisy history shrink to only the events that decide whether that property passed?

It shrinks to the election, the epoch 8 commit, and the stale epoch 7 finalize. The checker marks the exact relationship that breaks the contract.

The model defines what is legal, and the test supplies real evidence and now let us connect the entire correctness workflow.

STAGE 10 OF 10

Recap



10 RECAP

We began by separating safety from liveness and safety blocks conflicting commits, while liveness requires recovery when its assumptions hold.

Then we placed operations on a timeline and atomic points helped us ask whether concurrent results could fit one legal history.

We turned broad requirements into exact invariants about sequence slots, durable rows, and epoch authority.

PlusCal gave every actor a small labeled program, making each possible next step explicit.

TLC expanded those choices into a state graph and saved the path to any state that broke an invariant.

The counterexample showed us exactly why the epoch fence belongs at the irreversible commit boundary.

Jepsen-style clients and faults then collected a real history from the running implementation.

The oracle translated that evidence back to the same abstract contract and reduced the failure to a small witness.

The full method is a loop: define legality, exhaust the small model, attack the real system, and improve both whenever the evidence disagrees.