READ MODE · ANSWER FIRST

How Jepsen tests distributed systems with fault injection

Learn how Jepsen generates concurrent histories, injects network faults, checks linearizability, and reports distributed system failures.

CHEAT SHEET · 15 ESSENTIAL IDEAS

The whole story in 15 lines

Jepsen turns a distributed system promise into a fault filled concurrent history and checks whether any legal explanation survives.

1. A Jepsen test connects deployment, clients, generated work, faults, history, and a checker on one control node.
2. The control node prepares real database nodes so the experiment starts from a known cluster state.
3. A generator schedules operations and faults while logical processes perform one operation at a time.
4. Overlapping operation intervals create concurrency even though each individual process stays sequential.
5. Every request becomes an invocation event and every known response becomes a completion event.
6. Ok means success, fail means definite failure, and info preserves an outcome that may be unknown.
7. The nemesis is another scheduled process, so faults overlap ordinary client traffic instead of pausing it.
8. A partition removes communication paths and creates components that may observe different replica states.
9. Crashes stop a process while pauses freeze it, so recovery and pending operations can differ.
10. Healing a fault begins recovery, and Jepsen keeps observing how long convergence and availability take.
11. A workload creates operations while a model defines the legal state transitions those operations should obey.
12. Linearizability places each operation at one instant inside its interval while preserving real-time order.
13. The checker searches candidate sequential orders and rejects every branch that breaks time or model rules.
14. A stale read after a completed write leaves no legal linearization point and therefore proves a safety violation.
15. A Jepsen result is evidence from an experiment, combining safety checks with latency, availability, and artifacts.

Setup

01 SETUP

Start with one shared register named x, copied across three database replicas. Every copy begins at zero. The database promises that a read beginning after a completed write will return that new value.

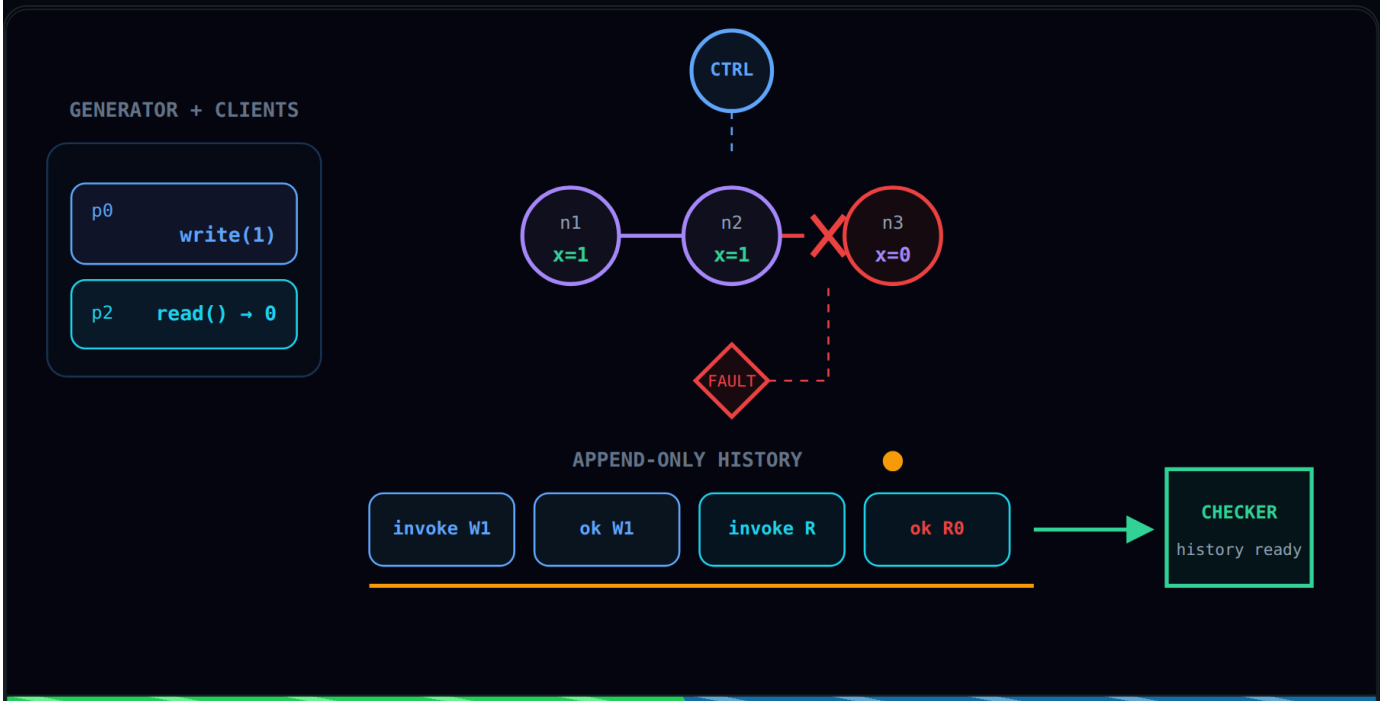
Process p0 sends write one through the database and receives an ok response. That completed response matters because the system has now told the client that x equals one.

Now a network partition isolates n3 while it still holds the older value zero. A partition alone is not a correctness bug, but it creates the stressful condition Jepsen wants to investigate.

After the write has completed, process p2 starts a new read against isolated n3. The replica returns zero, placing an older answer after the database already acknowledged one.

Jepsen records the write invocation, its ok response, the later read invocation, and the returned zero in order. The rest of this lesson asks how Jepsen created that evidence and whether any legal history can explain it.

Test Anatomy



02 TEST ANATOMY

Setup gave us one suspicious result: write one completed, yet a later read returned zero. Now we will keep those exact operations visible while Jepsen assembles the experiment around them.

The generator chooses those operations, and client processes send them through the database's normal API. The control node coordinates the run without becoming a database replica, so Jepsen observes what a real application could observe.

The nemesis cuts selected network links while the same clients keep working. History records every invocation and completion, but which component coordinates work, faults, evidence, and analysis as one repeatable run?

PAUSE AND PREDICT

Which component coordinates the full Jepsen experiment?

- The control node
- One database replica
- The checker by itself

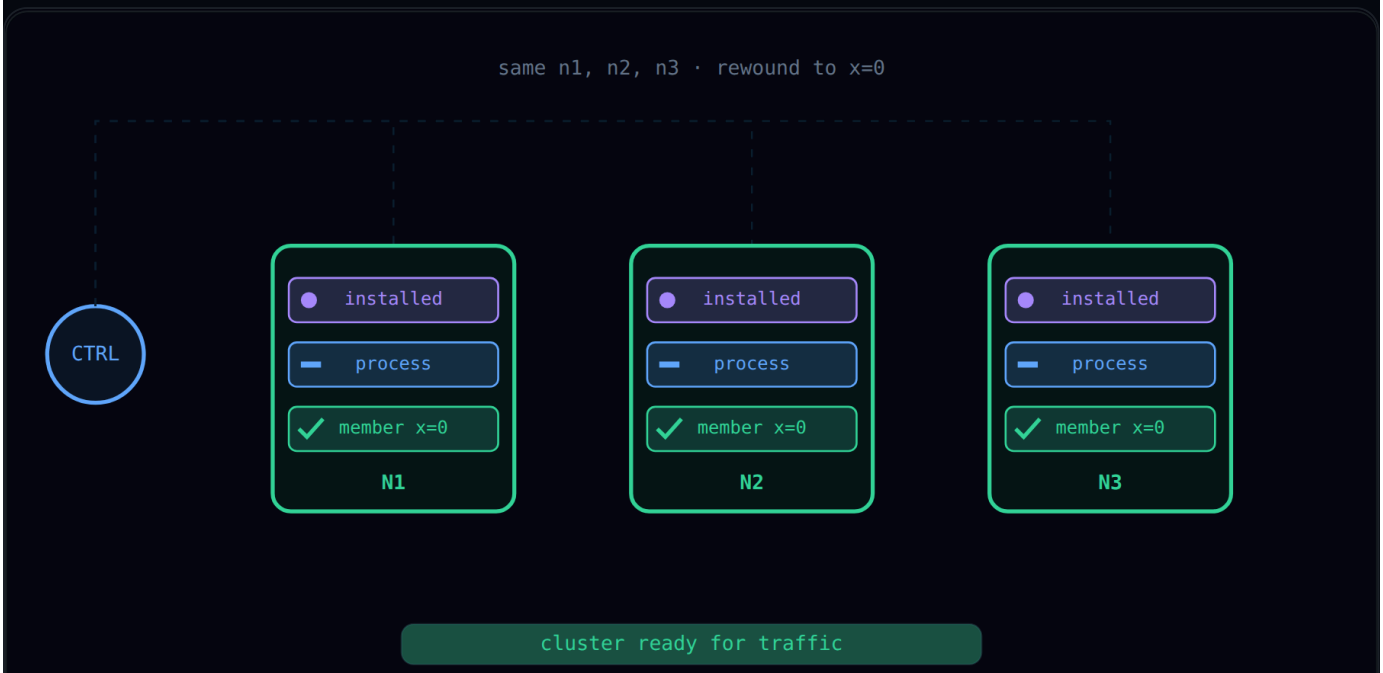
Skip this check

The control node closes the experiment by preparing nodes, scheduling work and faults, preserving history, and sending that history to the checker. Each role now has meaning because it acted on the incident we already understand.

Jepsen therefore lives around the system under test rather than inside its implementation. Next we will rewind these same n1, n2, and n3 replicas to see how the control node creates a trustworthy starting state.

STAGE 3 OF 17

Build the Cluster



03 BUILD THE CLUSTER

We just saw Jepsen wrap an experiment around n1, n2, and n3. Now we rewind those same replicas to x equals zero and inspect how the control node prepares them before any request is allowed.

The operating system adapter prepares each host. The database adapter installs software, writes configuration, starts processes, and joins replicas into one cluster. A real Jepsen test can also tear this state down, allowing repeated runs to begin from the same declared recipe.

A node can start without joining the group correctly, which creates a readiness problem before the history can be trusted. Which signal confirms membership, leader discovery, and replication readiness?

PAUSE AND PREDICT

When is the cluster ready for trustworthy test traffic?

- Every expected replica reports ready
- Every process has started
- One replica answers a ping

Skip this check

The test waits until every expected replica reports ready, which turns a pile of machines into a known starting point. A larger replica set makes the readiness gate cover more nodes. Only after every expected node crosses that gate does the workload have a trustworthy baseline.

Once setup is complete, our familiar cluster holds x equals zero and is ready for work. Next the generator will place the exact write and reads from Setup onto logical client processes.

STAGE 4 OF 17

Generate Operations

04 GENERATE OPERATIONS

The cluster is ready with x equal to zero, so the same write and reads from Setup now enter the generator queue. The generator chooses eligible work without needing to know the database driver details.

The generator assigns write one to p0, the safe read to p1, and the later read to p2. Each process remains sequential even when several processes are active together.

A slow request can occupy one process while other processes remain free. Waiting for every process would accidentally synchronize the workload, so how should the available lanes keep those useful races alive?

PAUSE AND PREDICT

One logical process is blocked. What should the generator do next?

- Schedule work on another free process
- Wait until every process is free
- Put another operation on the busy process

Skip this check

The generator schedules each eligible operation onto an available process while busy processes keep their place. At three lanes, the exact p0, p1, and p2 operations from our incident can all remain active independently.

Adjust Concurrency from one lane to several. Notice how one sequential lane has no race, while overlapping lanes multiply the candidate orders the checker must consider.

The schedule now contains our three familiar operation cards on named process lanes. Next we will keep their identities fixed and stretch each card into the interval when that operation was active.

Concurrent Clients

05 CONCURRENT CLIENTS

We just left the write and reads as cards on process lanes. This time view stretches those same operations from invocation to completion, so their identities stay fixed while the representation changes.

Process p0 performs write one while p1 performs its safe read. Their intervals overlap, which means neither operation has a completely fixed order relative to the other.

Printed events still appear one line at a time, but line order cannot reveal which operations had freedom to exchange order. Which start and finish boundaries expose actual concurrency?

PAUSE AND PREDICT

Which evidence proves that two operations were concurrent?

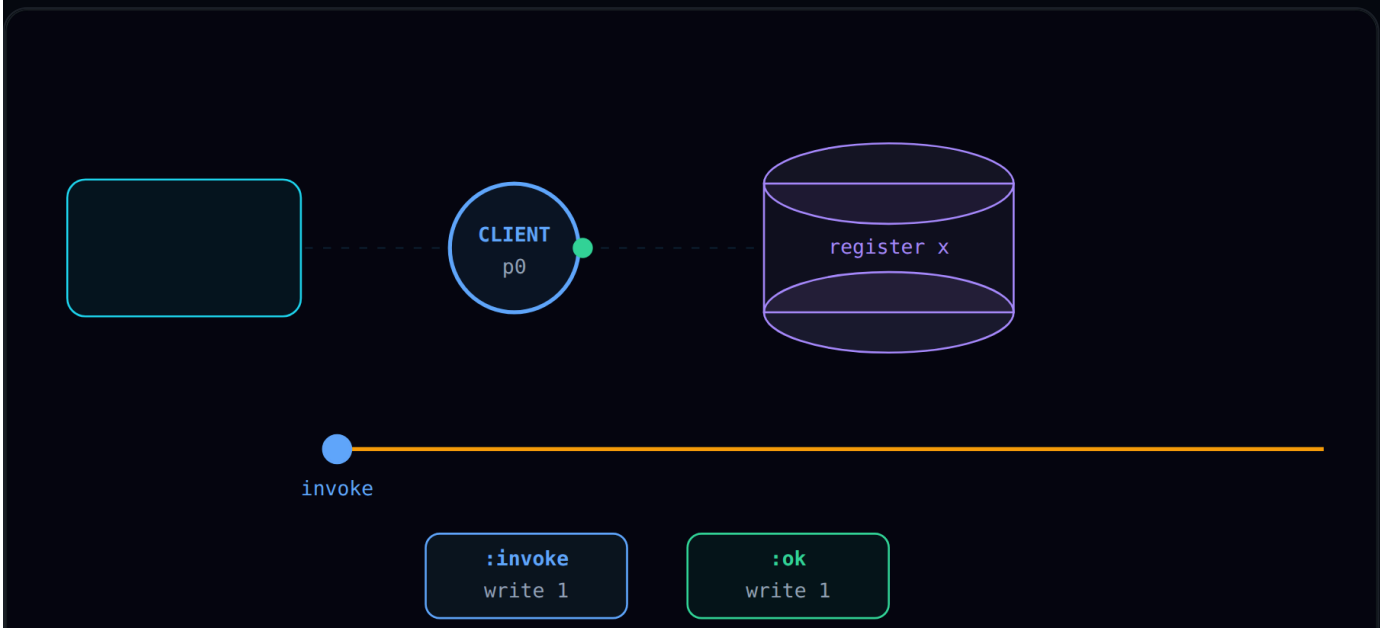
- Their invocation-to-completion intervals overlap
- Their log lines are adjacent
- They ran on the same logical process

Skip this check

The overlap allows more than one candidate order for p0 and p1. The p2 read is different because it begins after write one completed, so real time forces that read to stay later.

Each interval has a start and finish boundary. Next we will keep these same operations and turn those boundaries into the invocation and completion events stored in Jepsen history.

Client to History



06 CLIENT TO HISTORY

We saw operation intervals, and now we will build one interval by following write one from an abstract generator card through a real client call. This is the boundary where a planned operation becomes an observation about the system under test.

Before sending anything, Jepsen appends an invocation event that names the process, function, value, and start time for this operation. Recording the beginning first means the history still knows the request existed even if every later network message disappears.

The client sends the request to a database node, but a lost connection can leave the server and client with different knowledge. Which outcome records that uncertainty about whether the write took effect?

PAUSE AND PREDICT

The connection disappears before the reply. Which completion is honest?

- **Info: the result is uncertain**
- **Fail: it definitely did not happen**
- **Ok: it definitely happened**

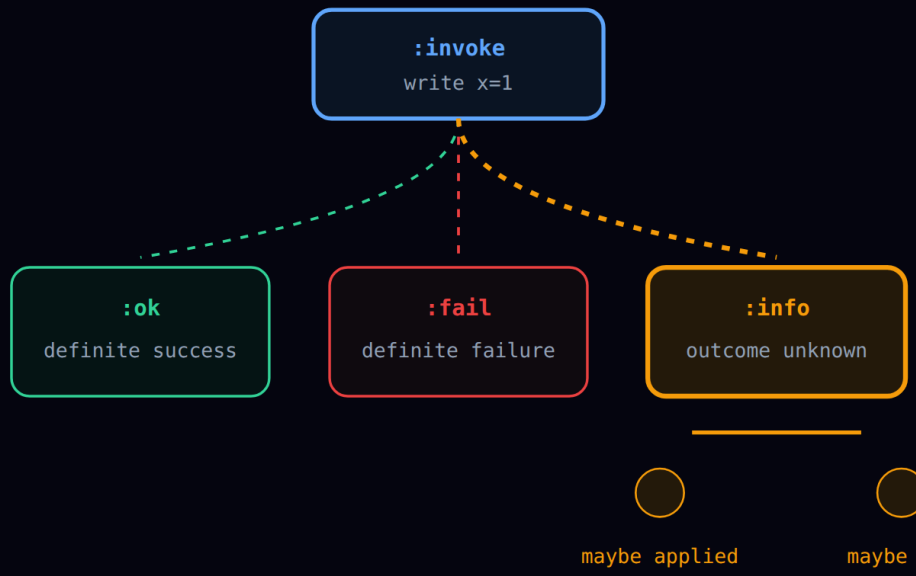
Skip this check

A clear success becomes ok, while a timeout becomes info. The server may have applied the write even though the client never learned the answer. Comparing those endings shows how honest uncertainty preserves more information than pretending every timeout failed.

Switch Response between Success and Timeout. The request path stays fixed, but the history changes from a counted write to one that may still have happened.

Every operation now has a beginning and the most honest ending available, which sets up the four event types we must interpret carefully. The checker will later reason from these records, so their wording must match what the client actually knows.

History Outcomes



07 HISTORY OUTCOMES

The client produced an invocation and a completion, so let us separate the four event types by the knowledge each one contributes to the history. Invocation begins the interval, while ok, fail, or info describes the strongest conclusion available at its end.

An ok event says the operation succeeded, while fail says it definitely did not take effect. The client adapter should use fail only when the operation truly could not have happened.

A timeout may happen after the server applied a write but before the reply arrived. Calling it fail could erase a write that other replicas observed, so which event type preserves both possibilities?

PAUSE AND PREDICT

The write times out after reaching the server. Which history outcome is honest?

- **Fail: the write definitely did not happen**
- **Info: the outcome remains uncertain**
- **Ok: the write definitely happened**

Skip this check

Info preserves both possibilities and forces later analysis to consider that the operation may have taken effect. Each possible outcome attaches different knowledge to the same invocation. This wider search is more difficult, but it is faithful to the evidence.

Switch Outcome through Ok, Fail, and Info. Compare how the same invocation becomes counted, erased, or deliberately uncertain in the checker’s history.

The history can now represent uncertainty without lying, and next the nemesis will create the failures that make those uncertain outcomes common. We are ready to overlap ordinary requests with deliberate disruption rather than testing only calm, healthy behavior.

The Nemesis

08 THE NEMESIS

We can record uncertain outcomes, and now a special nemesis process runs beside the client processes so the test can create trouble at controlled times. The name sounds dramatic, but technically it is another participant whose operations start and finish in the same schedule.

The generator schedules a nemesis start operation while reads and writes continue, which means the fault and ordinary traffic truly overlap. Because the fault is recorded, we can later align a suspicious response with the exact disruption that surrounded it.

Stopping every client before breaking the network would hide bugs that require requests to race with failure. Would that quiet maintenance exercise really test the stressful concurrent execution we care about?

PAUSE AND PREDICT

How should client traffic and nemesis faults be scheduled?

- Let their schedules overlap
- Stop clients before every fault
- Finish every fault before sending requests

Skip this check

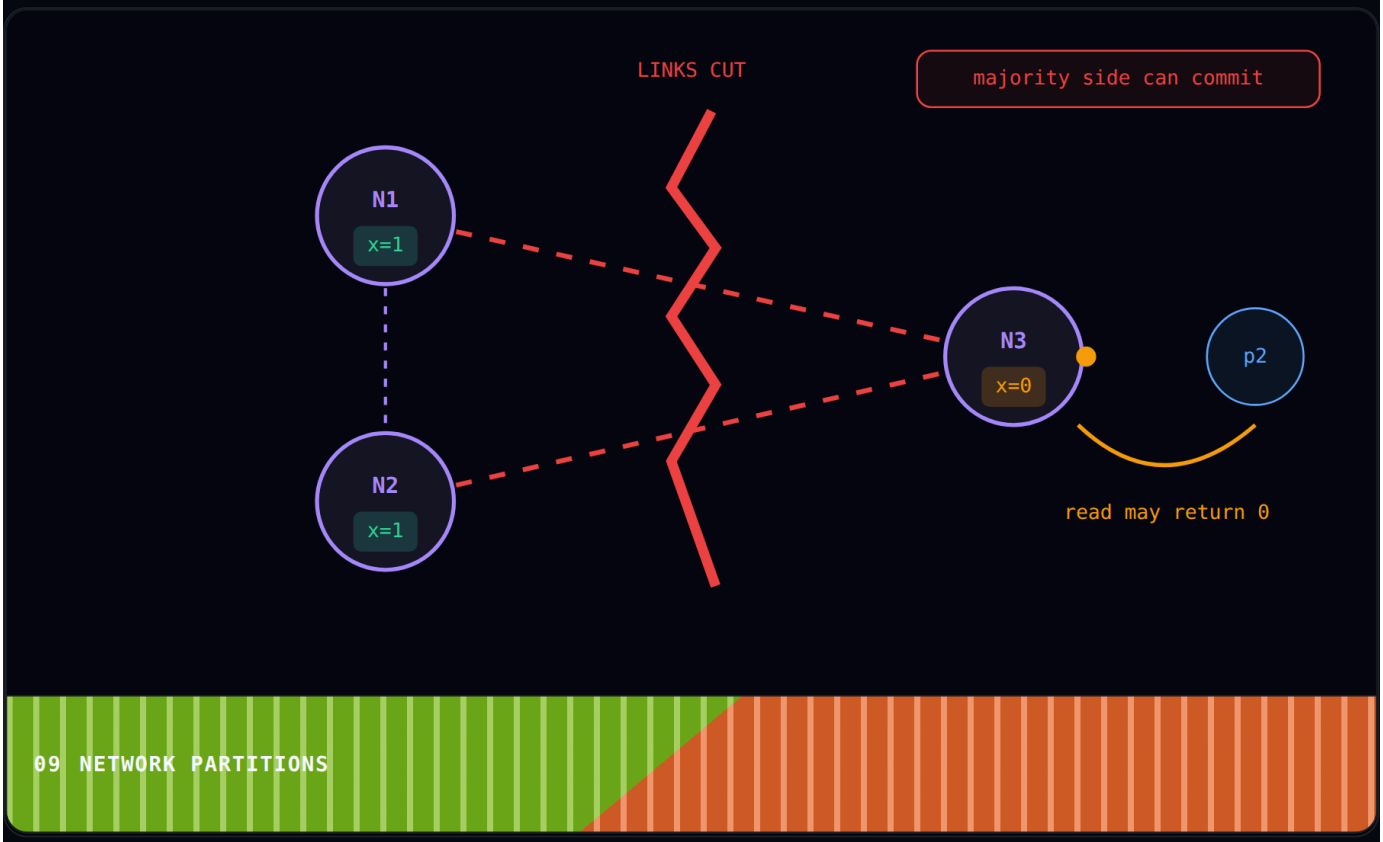
No, so both schedules advance together and the history records their overlap. Different disturbances preserve the same experiment shape. Controlled means we know what fault was attempted and when,

not that the test is gentle.

Switch Fault among Partition, Crash, and Clock. Follow the same in-flight request as it arrives late, loses its sender, or becomes vulnerable to duplicate work.

The nemesis gives us controlled failure timing, so next we will examine the most famous fault it creates: a network partition. This fault is especially useful because machines can remain alive while losing the ability to agree.

Network Partitions



09 NETWORK PARTITIONS

The nemesis is ready, and a network partition changes communication paths while all three database processes may remain alive and responsive. That distinction is crucial: a partition removes conversations between machines, not necessarily the machines themselves.

Before the fault, n1, n2, and n3 exchange replication messages and agree that register x contains the value one. This common state gives us a clean reference point for noticing which replica falls behind after links are cut.

The nemesis isolates n3 while a client can still reach it, but that path cannot prove the replica has current knowledge. Which response protects consistency when the replica cannot learn the latest value?

PAUSE AND PREDICT

An isolated replica still receives a read. Which response keeps the history linearizable?

- Return its local stale value
- Reject until it can coordinate
- Guess the majority value

Skip this check

The isolated replica may reject the read or answer from stale state. Those outcomes make the consistency tradeoff visible. Different communication splits create different components. A two-plus-one

split leaves a coordinating majority, while three isolated nodes remove every replica-to-replica path.

Switch Split between 2 + 1 and 1 + 1 + 1. Watch the commit verdict cross the majority threshold even though every replica remains alive.

A partition leaves processes running but disconnected, while other faults stop or freeze a process itself, which is our next distinction. Similar client symptoms can therefore come from very different internal conditions.

STAGE 10 OF 17

Crash and Pause

A partition removed links, and now we focus on n3 itself because a crash and a pause can look similar to clients while behaving differently inside the machine. In both cases requests may time out, yet the process lifecycle and retained memory are not the same.

A crash ends the database process and closes its heartbeat trace, while a pause freezes execution and leaves the process memory waiting in place. Resuming a paused process may continue old work, whereas restarting a crashed process follows startup and recovery logic.

Both faults can trigger client timeouts even though their recovery paths differ. Which heartbeat and in-memory state clues reveal whether recovery should restart or simply resume?

PAUSE AND PREDICT

Which clue distinguishes a crash from a paused process?

- A crash loses process state; a pause freezes it
- Only a crash causes client timeouts
- A paused process keeps sending heartbeats

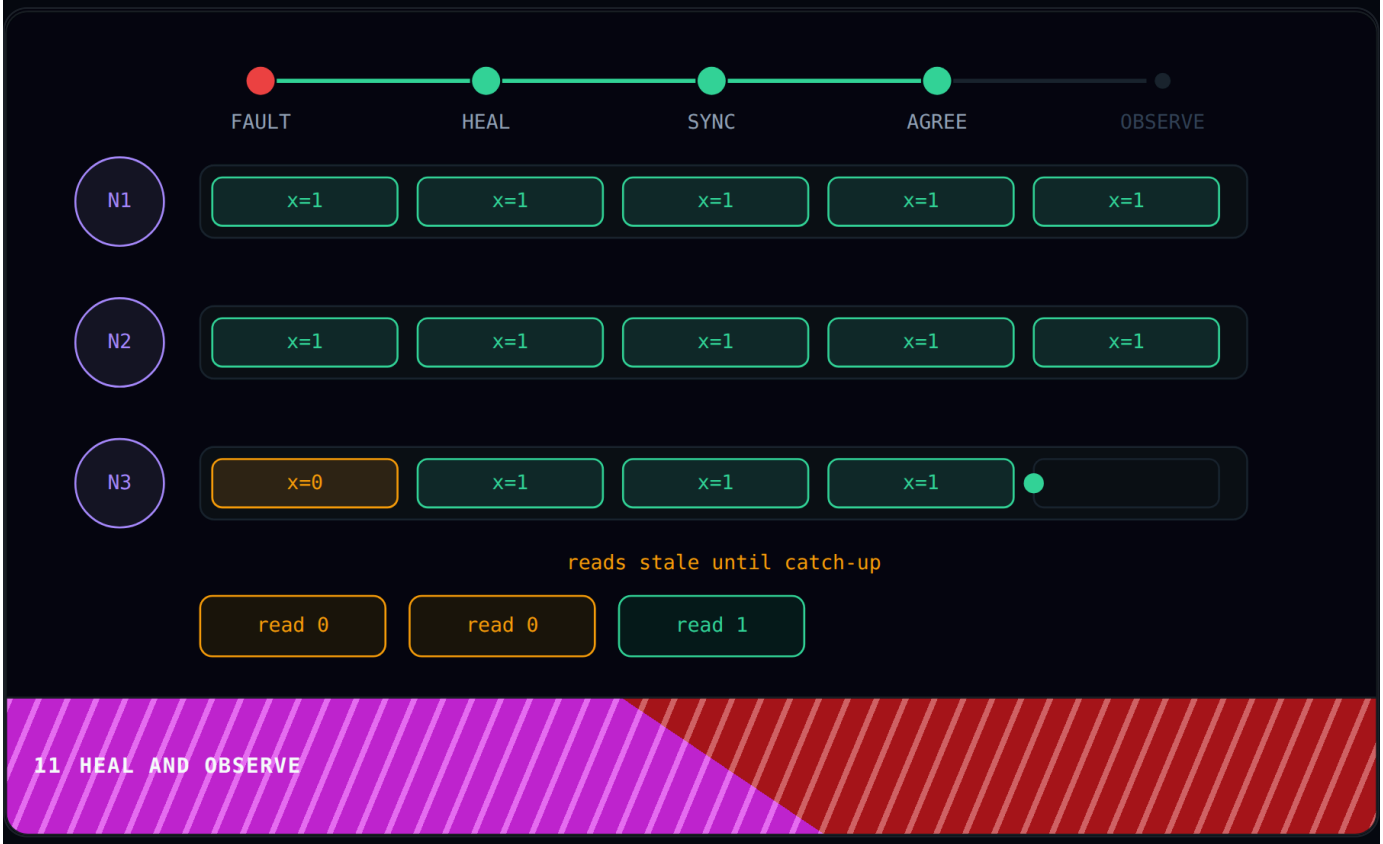
Skip this check

The state strip disappears for a crash but remains frozen for a pause, and the recovery arrows therefore take different paths. Both failures use the same encoding. The shared picture makes the one meaningful difference easier to see.

Switch Failure between Crash and Pause. Compare a restart from durable state with a resumed process whose memory and pending work never disappeared.

Jepsen can heal either fault, but healing only starts recovery. Next we will keep watching until the cluster converges or the recovery window expires because removing the disturbance does not instantly restore health.

Heal and Observe



11 HEAL AND OBSERVE

The fault stopped or isolated part of the cluster, and now the nemesis restores communication while the test continues sending reads and writes. Continuing the workload prevents recovery from becoming a special quiet period that real users would never receive.

Nodes n1 and n2 already hold value one while n3 begins with value zero, so the repaired links must carry missing state across the cluster. The visual gap between those values is the recovery work that remains after the network itself is healthy.

The network is healthy again but n3 remains stale, so the first successful ping proves only restored transport. What observation would establish restored service correctness?

PAUSE AND PREDICT

What proves that recovery restored correct service?

- **Replicas converge and later reads agree**
- **The isolated node answers one ping**
- **The network link turns healthy**

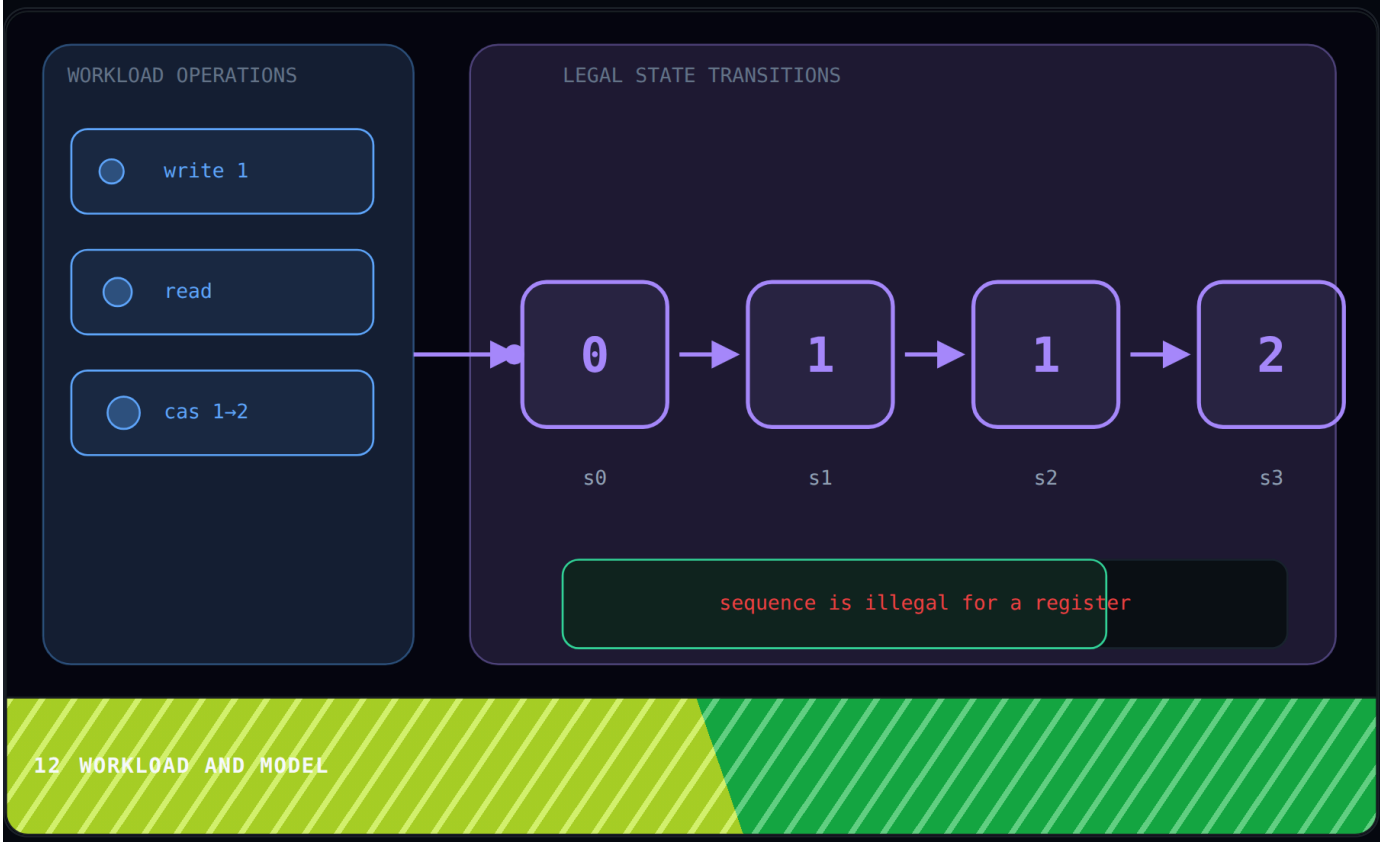
Skip this check

Recovery lands only when n3 receives value one and later reads agree again. Fast and slow catch-up paths produce different observation windows. This lets the report distinguish a brief interruption from a long tail of stale or unavailable behavior.

Switch Recovery between Quick and Slow. The network is already healed, but the readout shows whether clients are current now or remain stale until catch-up finishes.

We now have a complete fault filled history, but a checker still needs a precise workload and model before it can call any behavior correct or incorrect. Raw events become evidence only when they are compared with an explicit promise.

Workload and Model



12 WORKLOAD AND MODEL

We finished collecting a history, and now the checker needs to know what kind of object those operations were supposed to implement. A list of reads and writes has no universal meaning until we define the state and rules behind those names.

For our register, write one replaces the current value, and read returns it. Compare and set changes state only when the expected value matches. These rules are enough to replay a proposed sequential story one operation at a time.

The same sequence can be legal for one datatype and impossible for another because sets, queues, and registers allow different transitions. Which explicit rules bind the checker to the intended promise?

PAUSE AND PREDICT

What tells the checker which histories are legal?

- The declared datatype state transitions
- The database node logs
- Wall-clock event order alone

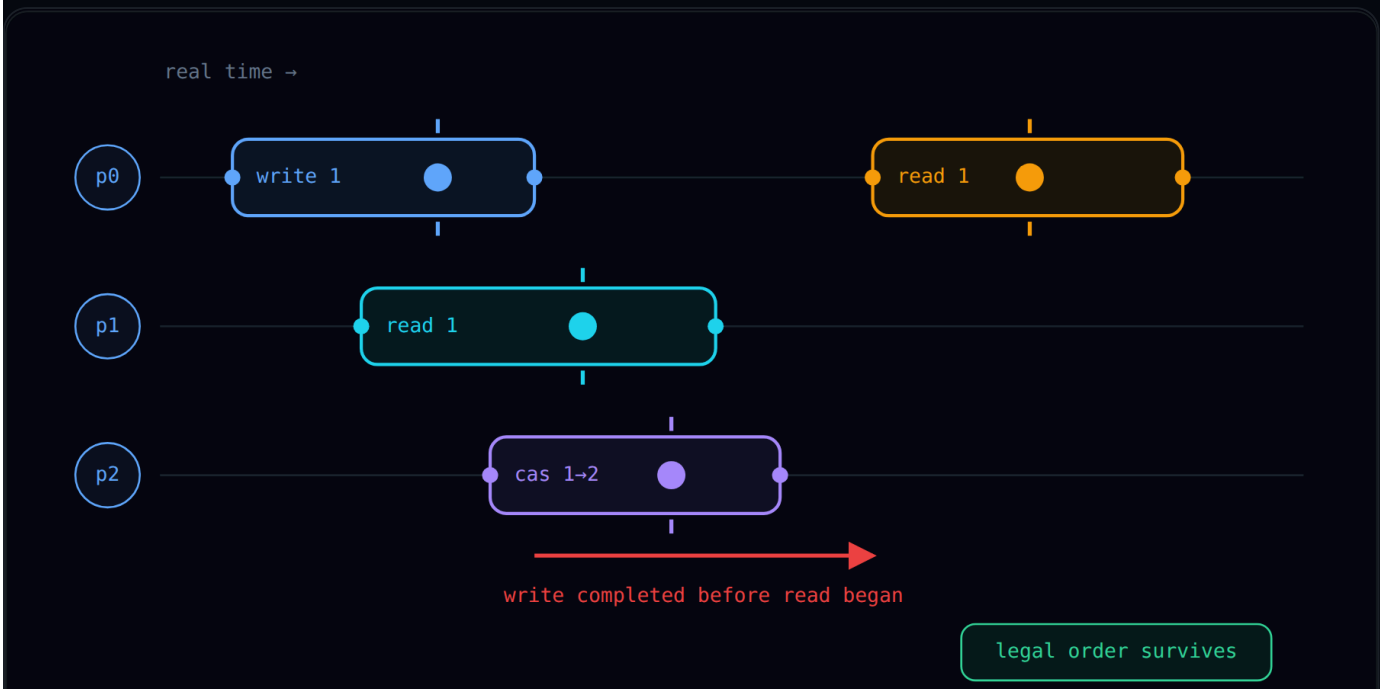
Skip this check

The workload and model use the same operation vocabulary, allowing every observed response to be replayed against explicit state transition rules. Register and set workloads change those laws together. The checker judges the declared object rather than relying on vague intuition about what looks sensible.

Switch Workload between Register and Set. The observed sequence stays fixed, while the model’s legal transitions decide whether that sequence is accepted or rejected.

Our register model now knows which sequential stories are legal, and the next question is how linearizability connects one such story to real concurrent time. We need a rule that respects both datatype behavior and the timing visible to clients.

Linearizability



13 LINEARIZABILITY

The register model gives us sequential rules, and linearizability asks whether the concurrent history can look like one legal sequence of atomic moments. Operations need not execute in one instant, but their effects must be explainable as if they did.

Each operation may take effect anywhere inside its own interval. Overlapping operations can therefore swap order without contradicting real time. That flexibility makes concurrent histories explainable even when no observer saw one global sequence directly.

Write one finishes before the final read begins, so real time removes some ordering freedom. Where can that read be placed without respecting the completed write?

PAUSE AND PREDICT

Can the final read be ordered before the completed write?

- No, real time places it afterward
- Yes, every operation may move freely
- Only if both use the same timestamp

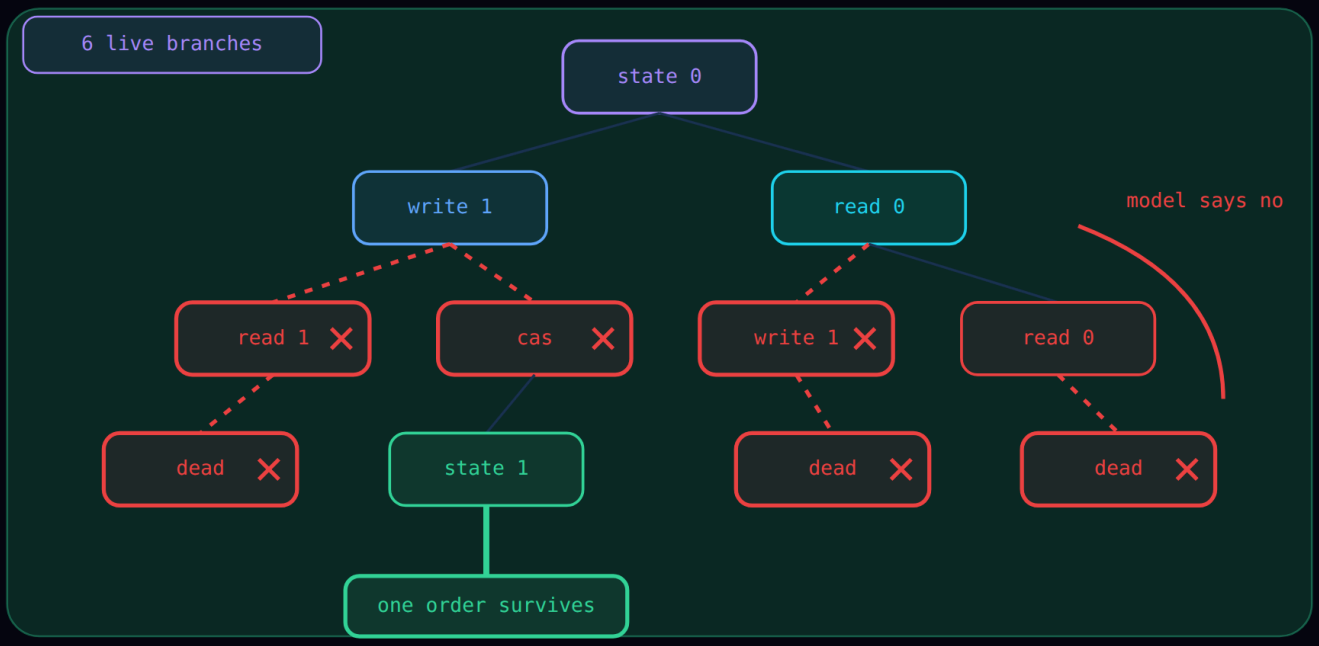
Skip this check

It cannot move earlier than the completed write, which means returning one fits the model while returning zero does not. The safe and stale histories keep the same intervals. Only the response changes, yet that small change removes every legal atomic placement.

Switch History between Legal and Stale. Keep every interval fixed and compare whether one legal atomic order survives the final response.

Linearizability turns timing into ordering constraints, and next the checker will search every candidate order that still respects those constraints. The search is where an intuitive timeline becomes a repeatable algorithmic verdict.

Checker Search



14 CHECKER SEARCH

Linearizability gave us constraints, and now the checker starts with the initial register value zero and a frontier of operations that may legally happen next. The frontier contains only operations whose required real-time predecessors have already been placed.

Concurrent operations create branches because either one may be tried first, while real time edges prevent already completed work from moving after later invocations. Every branch is a proposed sequential explanation, not another execution of the database.

Each branch replays register state, but a read can return a value that branch never produced. If the response contradicts the model state at this point, should the proposed order remain alive?

PAUSE AND PREDICT

What should the checker do with a model-inconsistent branch?

- **Prune it immediately**
- **Keep it until every operation is placed**
- **Rewrite the observed response**

Skip this check

That branch ends immediately while consistent branches keep growing until one complete legal order survives. Greater overlap gives the search tree more candidate branches. More concurrency increases freedom, while real-time order prunes freedom before model checking even begins.

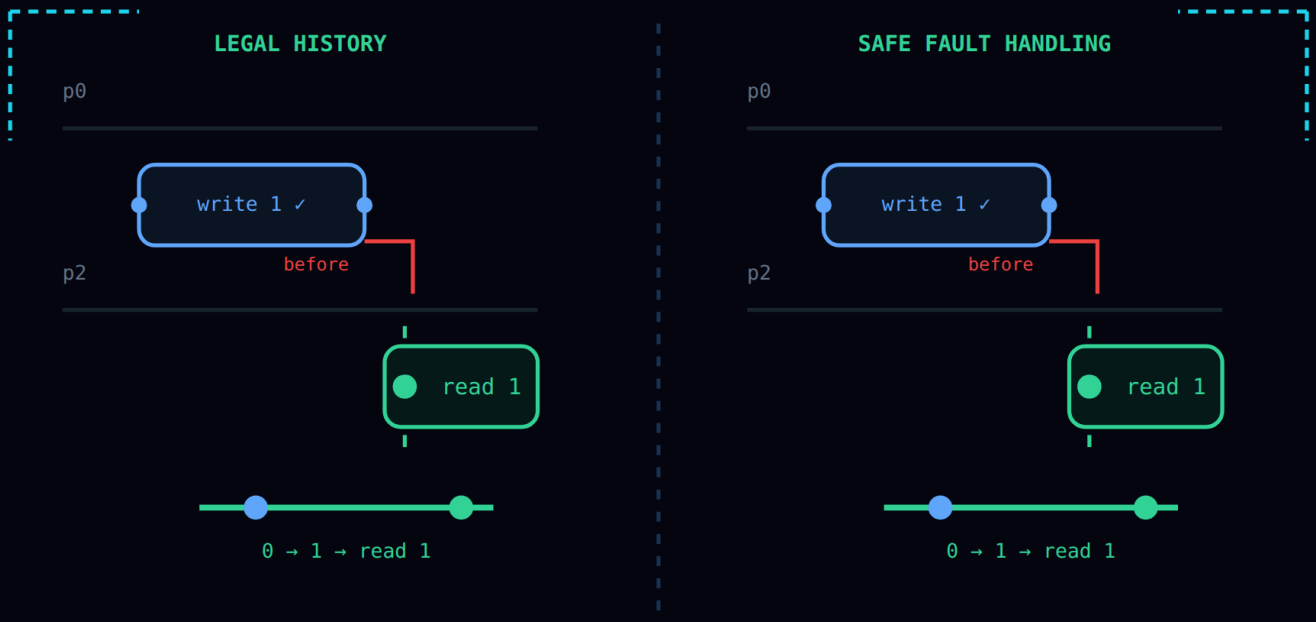
Switch Overlap between More overlap and Less overlap. Compare a branching search with several live candidates against the single order forced by real time.

A valid history needs only one surviving explanation, but our stale read kills every explanation, which is the visual proof in the next stage. We will compare a legal and illegal ending with all other facts held constant.

Reveal the Violation

★ If you remember one thing · A read that starts after write(1) completed cannot legally return the older value 0.

Try it in Watch mode. Produce a history the checker must reject.



15 REVEAL THE VIOLATION

The checker search prepared us to compare two histories with the same timing, the same partition, and the same completed write of one. Holding those facts constant is important because it isolates the final response as the reason the verdict changes.

On the safe side, isolated n3 rejects the request or returns one, so the final read can sit after the completed write without breaking register rules. Rejecting may reduce availability, but it does not invent an older value after a newer write is complete.

On the stale side, the read begins after write one completed but returns zero. Moving within the read interval never crosses that completed write, so where could a rescuing atomic point possibly go?

PAUSE AND PREDICT

The stale read returns 0 after write 1 completed. What will the checker conclude?

- **Legal: concurrent operations can reorder**
- **Illegal: no valid order exists**

Unknown: the partition hides the answer

Skip this check

No legal point remains because real time places the read after the write while the register model requires value one. The two fault outcomes visibly diverge at the same final interval. This is the central payoff: one observed zero turns a stressful run into a proof of incorrect behavior.

Before we move on, try the Fault outcome control yourself. Compare Reject or fresh with Serve stale and notice which choice leaves the checker without a legal order.

That single impossible response proves a safety violation, and now we will finish by learning what the full report establishes and what it cannot claim. Strong evidence also requires careful language about the limits of one experiment.

Read the Result

16 READ THE RESULT

We found an impossible stale read, and Jepsen now assembles the checker verdict with the exact history, timeline, logs, latency measurements, and availability behavior. These artifacts let another engineer inspect not just the conclusion but the path from execution to conclusion.

A safety verdict answers whether the observed history obeyed the selected model, while throughput and latency graphs show how much service remained during the fault and recovery. Correctness and availability answer different questions, so a useful report keeps both instead of compressing everything into one score.

One invalid history proves the implementation violated its promise, while valid checked histories cover only the executions we explored. How should the report describe that bounded success without claiming every possible timing, failure, workload, and machine state?

PAUSE AND PREDICT

Every checked history is valid. What does that result establish?

- **The implementation is universally correct**
- **This experiment found no violation**
- **The service stayed fully available**

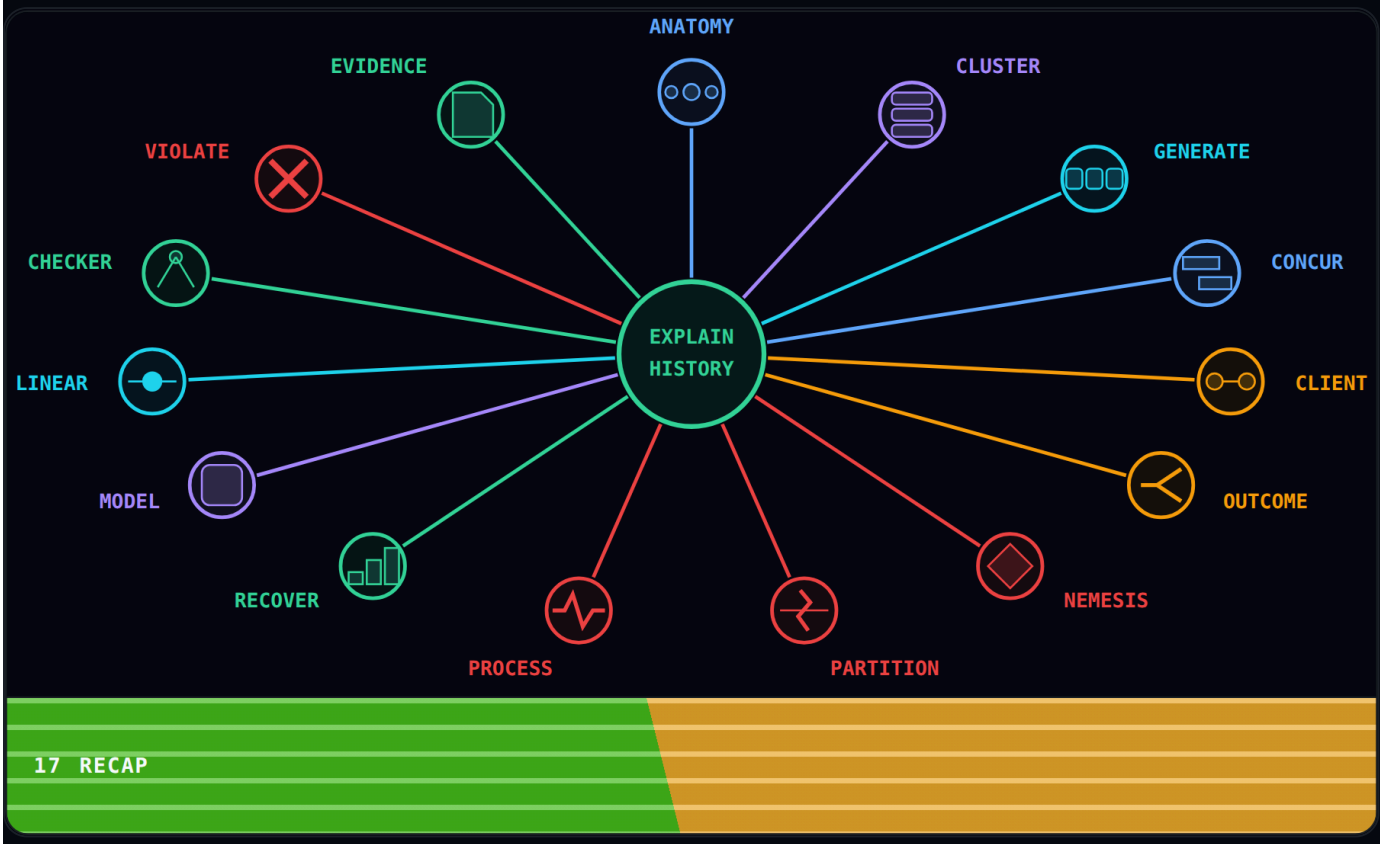
Skip this check

A valid run means this experiment found no violation, not that every possible execution is correct. Valid and invalid verdicts keep the surrounding evidence but support different claims. An invalid run supports a concrete counterexample, while a valid run supports only a bounded statement about what was tested.

Switch Verdict between Valid run and Violation. Compare a bounded claim that this run found nothing with a counterexample that proves one concrete bug.

Jepsen is powerful because it creates precise, reviewable evidence under real faults. Now let us step back and connect the entire experiment from setup through verdict. The value comes from the chain of evidence, not from a mysterious red or green badge.

Recap



We started with test anatomy and learned that one control node connects deployment, generated work, clients, faults, history, and checking into a repeatable experiment.

Then we prepared a known cluster state because a correctness experiment cannot explain later behavior if its starting database membership was already uncertain.

We asked the generator for operations and assigned them to logical processes. Each process stayed sequential while the whole workload remained concurrent.

We drew operation intervals on process lanes and saw that overlap, rather than log line adjacency, determines when operations may exchange order.

We followed one client request into the history and preserved both its invocation and completion so its full interval remained available to the checker.

We separated ok, fail, and info outcomes. An unknown response must remain possible instead of being simplified into a definite failure.

We scheduled the nemesis beside ordinary clients so partitions, crashes, pauses, and clock trouble could race with real operations.

We cut network links into components and learned that a live isolated replica can still be dangerous when it serves data without current state.

We distinguished crashes from pauses by their heartbeat and stored state, which explained why similar client timeouts can require different recovery paths.

We healed the fault and kept observing until replicas converged. Restored communication begins recovery rather than completing it instantly.

We paired a workload with a model so the checker knew both which operations appeared and which state transitions those operations were allowed to produce.

We defined linearizability by placing one atomic point inside every operation interval while preserving the order of completed work and later invocations.

We watched the checker branch through candidate sequential orders. It pruned each path that violated real time or the register model.

We revealed the stale read after write one completed and saw every legal position disappear, which turned one tiny response into a proof of violation.

Finally, we read the evidence bundle with care because an invalid history proves a bug while a valid run only reports that this experiment found none.

All fifteen lessons now converge on one idea. Jepsen makes faults, concurrency, observations, and promises precise enough that a distributed system must explain the history it actually produced.