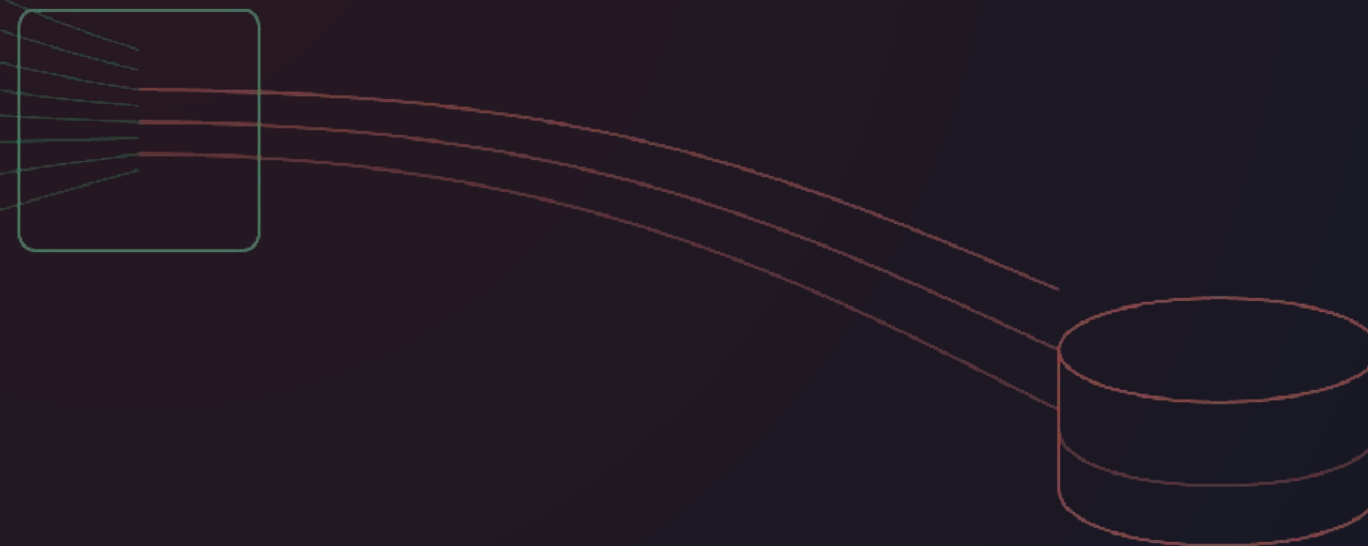


IBM CONTEXTFORGE · LOAD TEST, 3 JANUARY 2026

3,000 users exposed it.

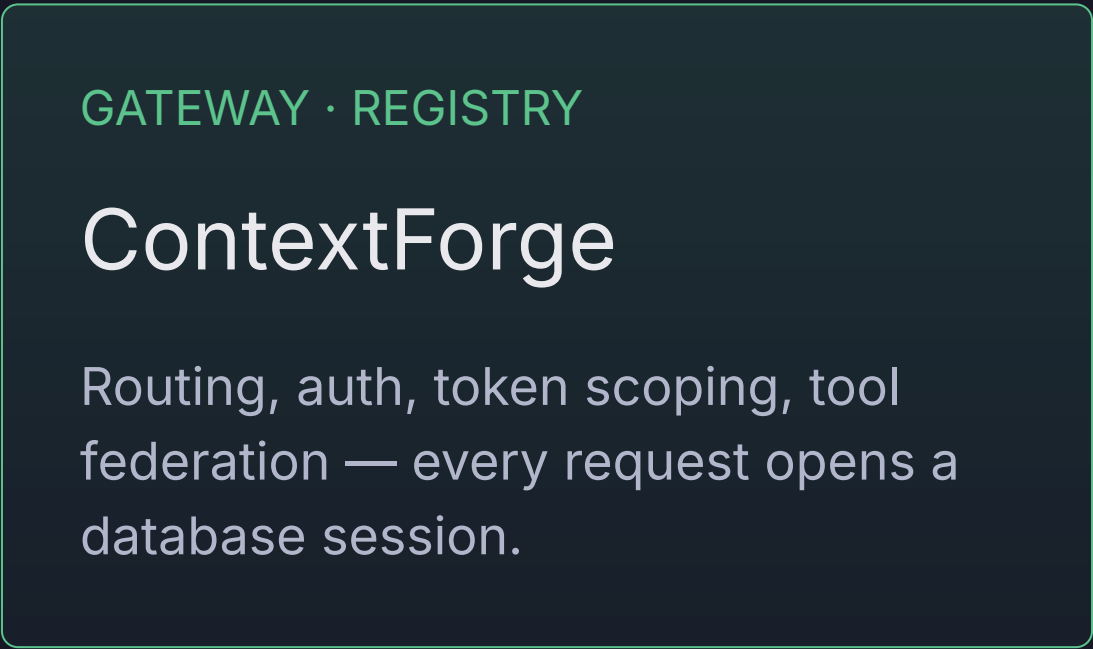
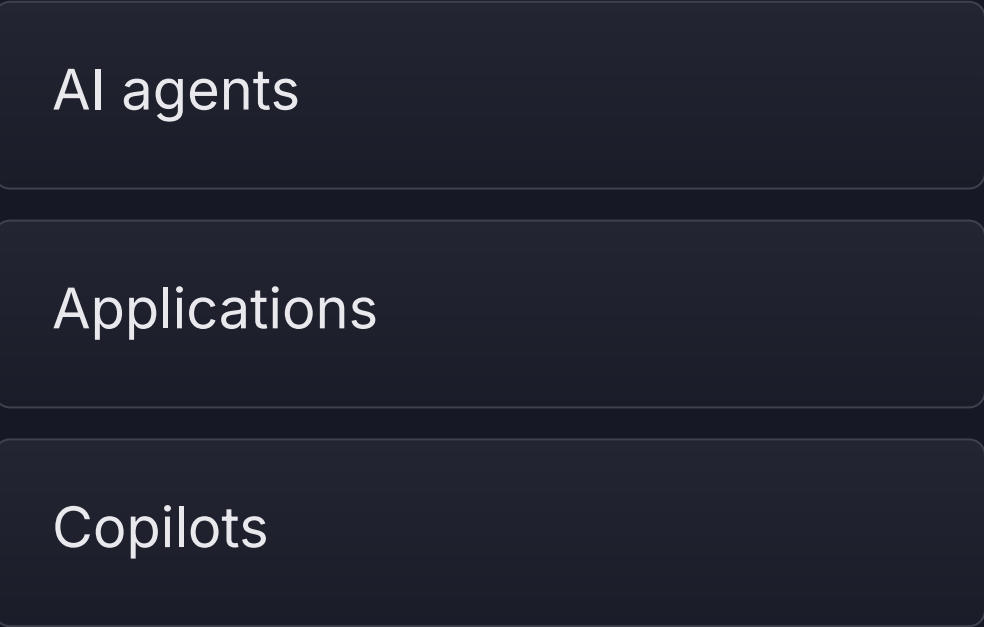
Could the code have told us first?



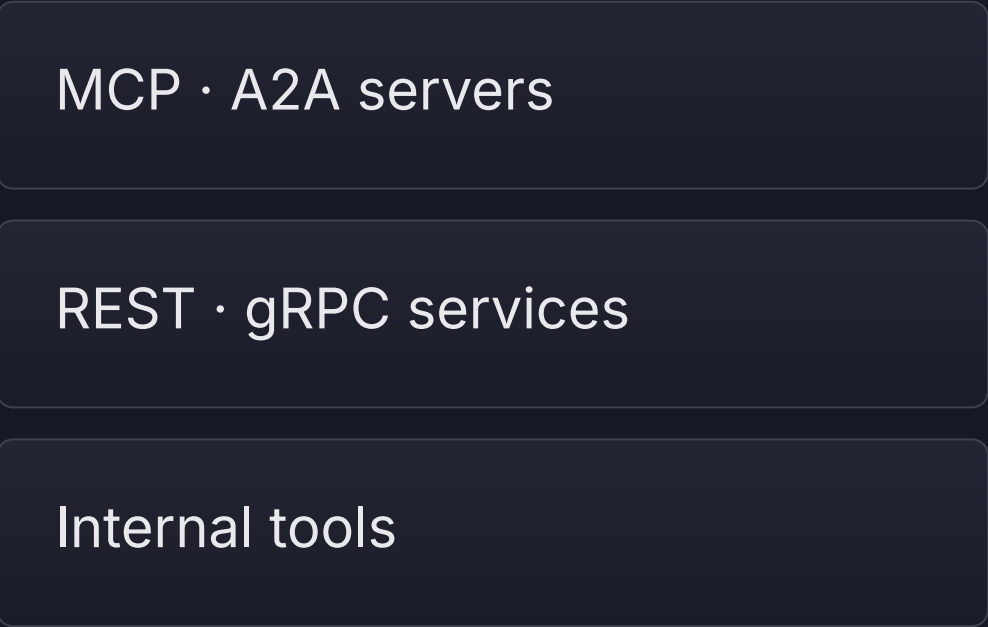
IBM ContextForge

An open-source AI gateway and registry for MCP, A2A, REST and gRPC — sitting in a high-concurrency path.

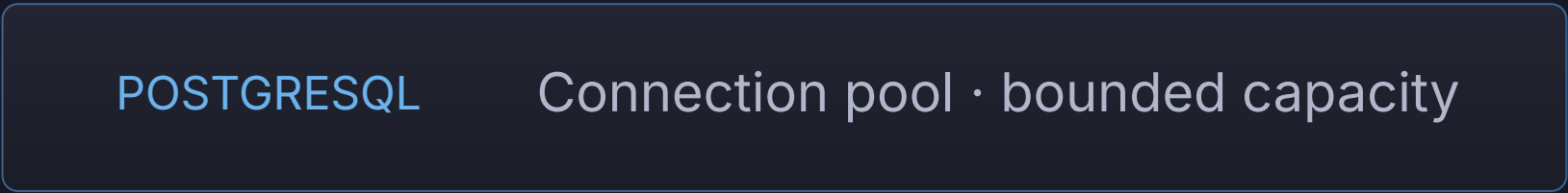
CALLERS



DOWNSTREAM



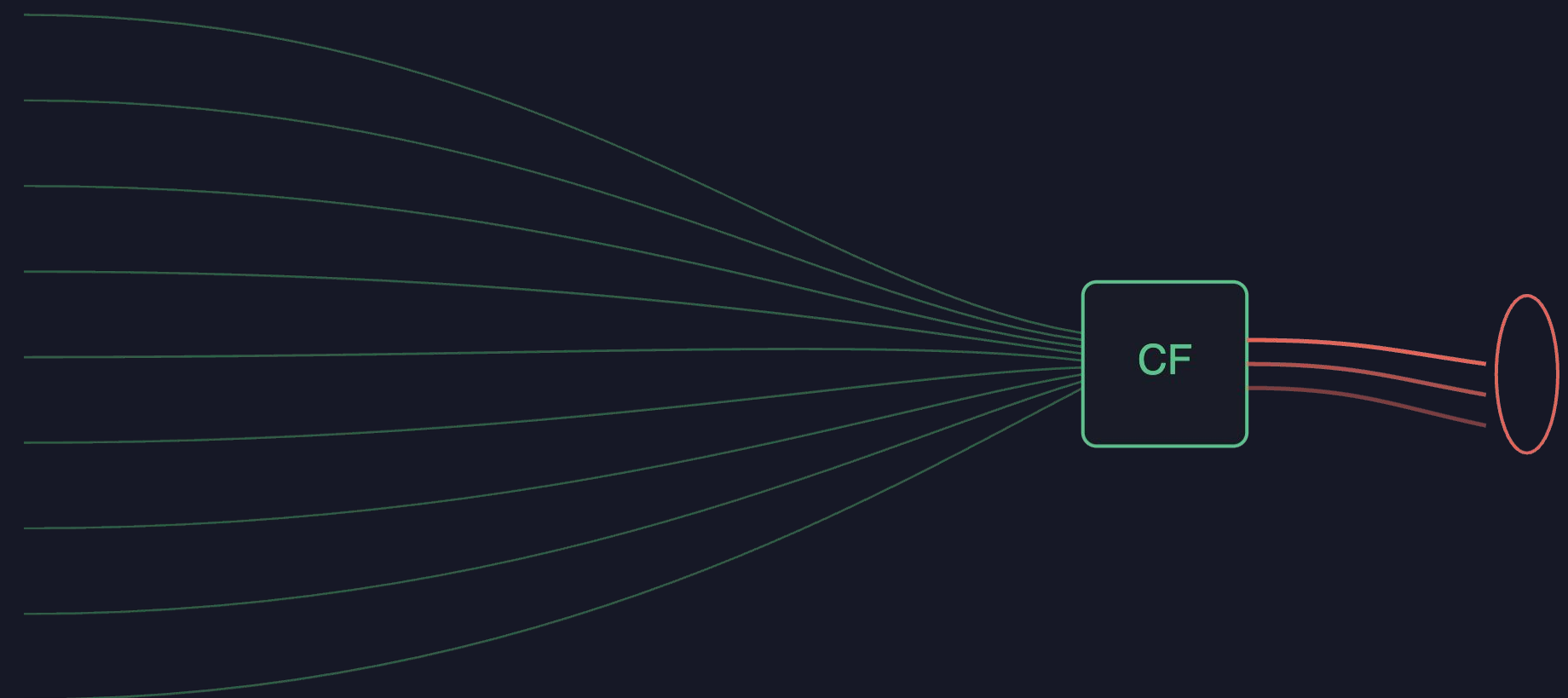
ONE SESSION PER REQUEST — A FINITE RESOURCE



3,000

CONCURRENT USERS

A test designed to expose behaviour that never appears during ordinary development.
Something unusual appeared in PostgreSQL:
connections stopped cycling normally.



REQUESTS HEALTHY

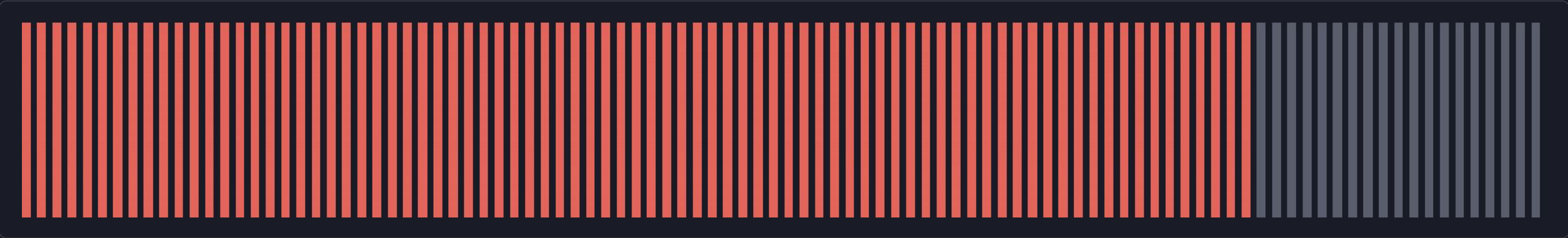
CONNECTIONS BACKING UP

42–81 connections stuck.

Occupied, but doing no useful work — and unavailable for reuse.

IDLE IN TRANSACTION

AVAILABLE CAPACITY



42–81

Connections observed in `idle in transaction`

Finite

Pool capacity — every held connection subtracts from it

0

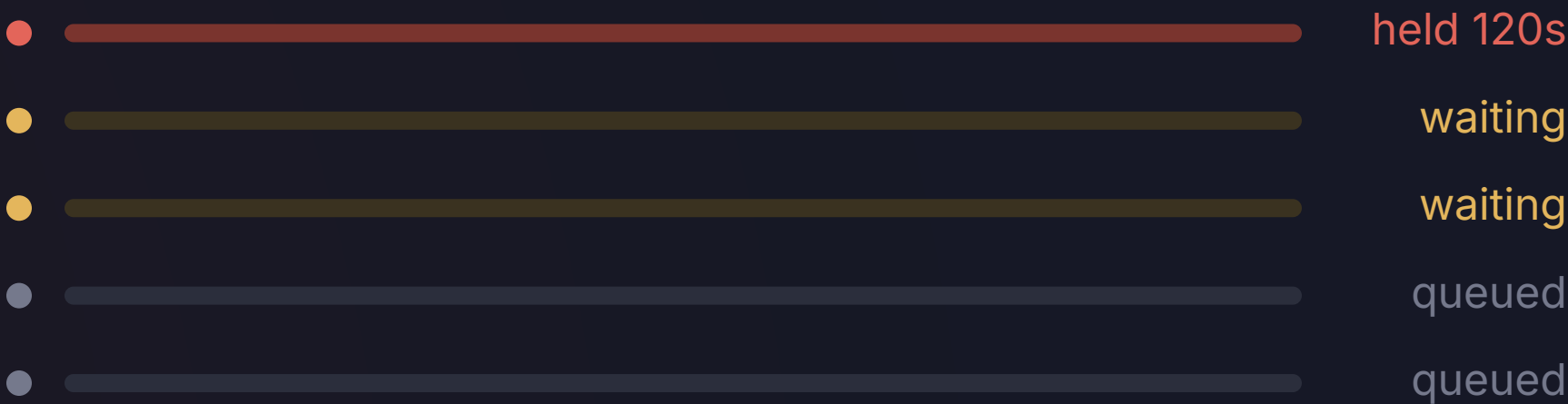
Useful work being performed on those sessions



A connection that should have been released... could stay stuck.

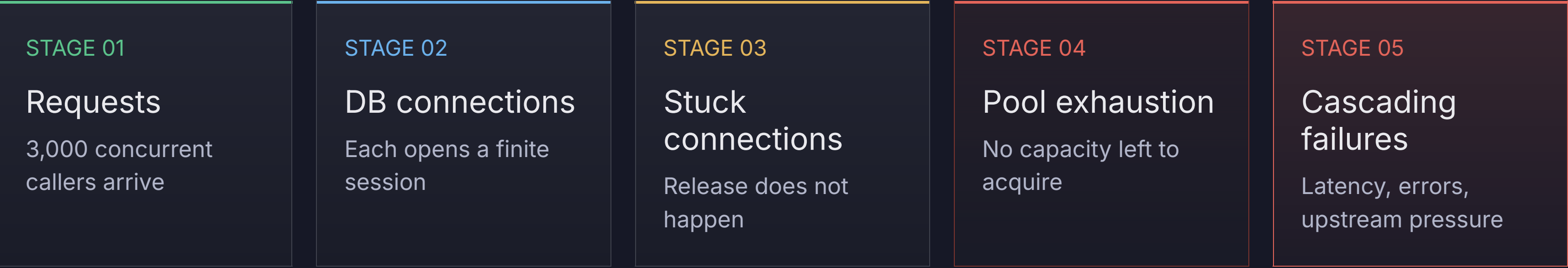
Two minutes is an eternity when thousands of requests are competing for a limited pool.

REQUESTS QUEUED BEHIND IT



How a cleanup defect becomes an outage.

Each stage is ordinary. The amplification is not.



NORMAL OPERATION

SYSTEM-LEVEL RELIABILITY FAILURE

mcpgateway/middleware/token_scoping.py

historical

```
finally:  
    db.commit()  
    db.close()
```

↓ Execution order — commit, then close

It looks reasonable.

A `finally` block, a commit, a close. The intent is clearly to clean up the database session regardless of what happened earlier.

But `finally` does not mean every statement inside it is guaranteed to execute.

There is a failure path hiding here.

token_scoping.py

commit raises

finally:**db.commit()**

raises

~~db.close()~~

Execution never reaches close().

What if commit throws?

`db.commit()` is itself an operation that can fail. If it raises, Python exits the sequence immediately.

Close appears inside `finally` , but its execution depends on the preceding commit succeeding.

The cleanup operation is not truly guaranteed.

Three lines → pool exhaustion.

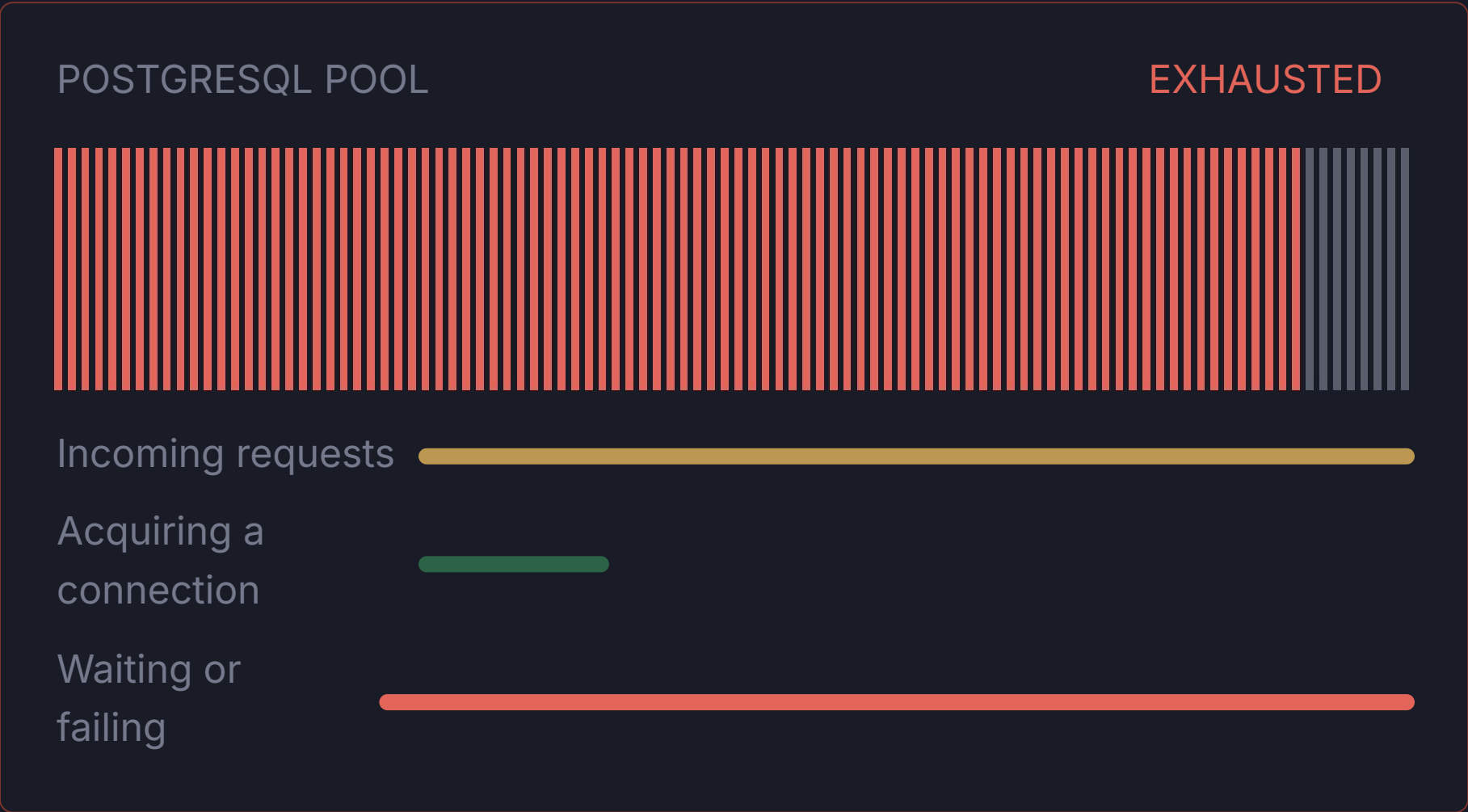
THE DEFECT

```
finally:  
    db.commit() ✖  
    db.close()
```

One operation in the cleanup sequence can fail, preventing the actual resource release from running.

↑
x 3,000 USERS

THE PRODUCTION SYMPTOM



Make release unconditional.

The implementation change is tiny. The conceptual change is not.

BEFORE

```
finally:  
    db.commit()  
    db.close()
```

Release depends on commit succeeding.



IBM FIX

```
finally:  
    try:  
        db.commit()  
    finally:  
        db.close()
```

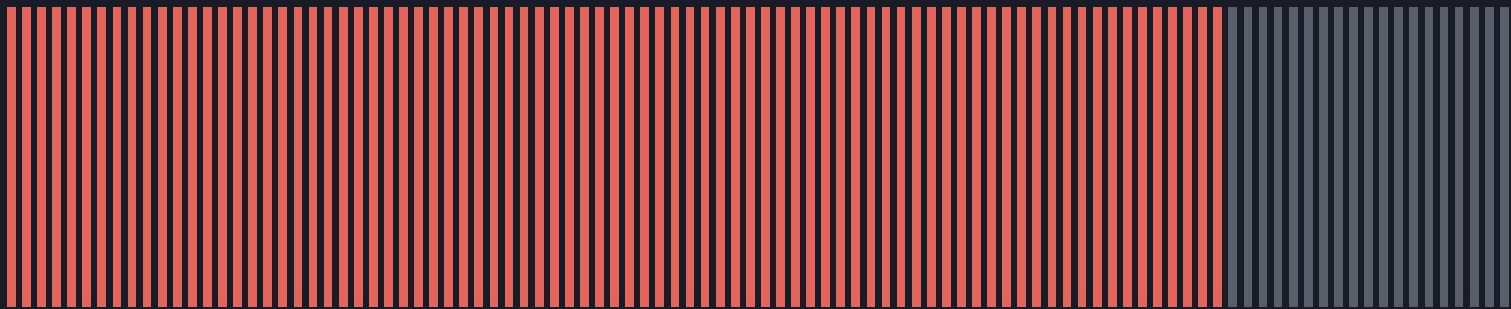
Release survives a failing commit.

If you acquire a finite production resource, its release must survive every relevant failure path.

Same load. Different result.

BEFORE

POOL EXHAUSTED



Stuck connections

42–81

Time to release

120 s

AFTER FIX

HEALTHY CYCLING



Stuck connections after test

0

Connection cycling

1–2 s

SLIDES 1–11 · CLOSED

The incident analysis

Failure observed under 3,000 users

Defect located in the cleanup path

Fix written and validated by IBM



FROM HERE · THE TOMOSU TEST

Could Tomosu have caught it first?

Rewind the repository to the vulnerable code

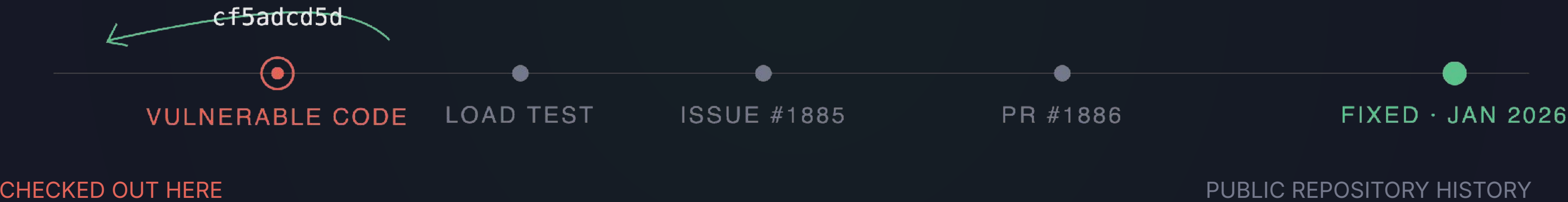
Give Tomosu the source and nothing else

Score the result against a locked criterion

Everything IBM learned came from production-scale conditions. The experiment from here asks whether Tomosu could surface the same reliability weakness proactively — from the code, before the load test.

Now we rewind.

We are not asking Tomosu to review IBM's patch.
We want the code exactly as it stood before the remediation existed.



No hints.

Otherwise we would only be testing whether Tomosu can repeat information we already supplied.

THE ENTIRE WORKSPACE

mcpgateway/middleware/
token_scoping.py

755	32 KB	1
LINES	SIZE	FILE

WITHHELD

Issue #1885	PR #1886
The fix	Load test results
Idle in transaction diagnosis	Release notes

And no instruction from us about what to look for.

Success criterion — locked before scanning.

SEALED

Identify unsafe database-session cleanup where an operation before `close()` can fail and prevent reliable connection release.

DIRECT HIT

Names the unsafe session lifecycle on this path

ADJACENT

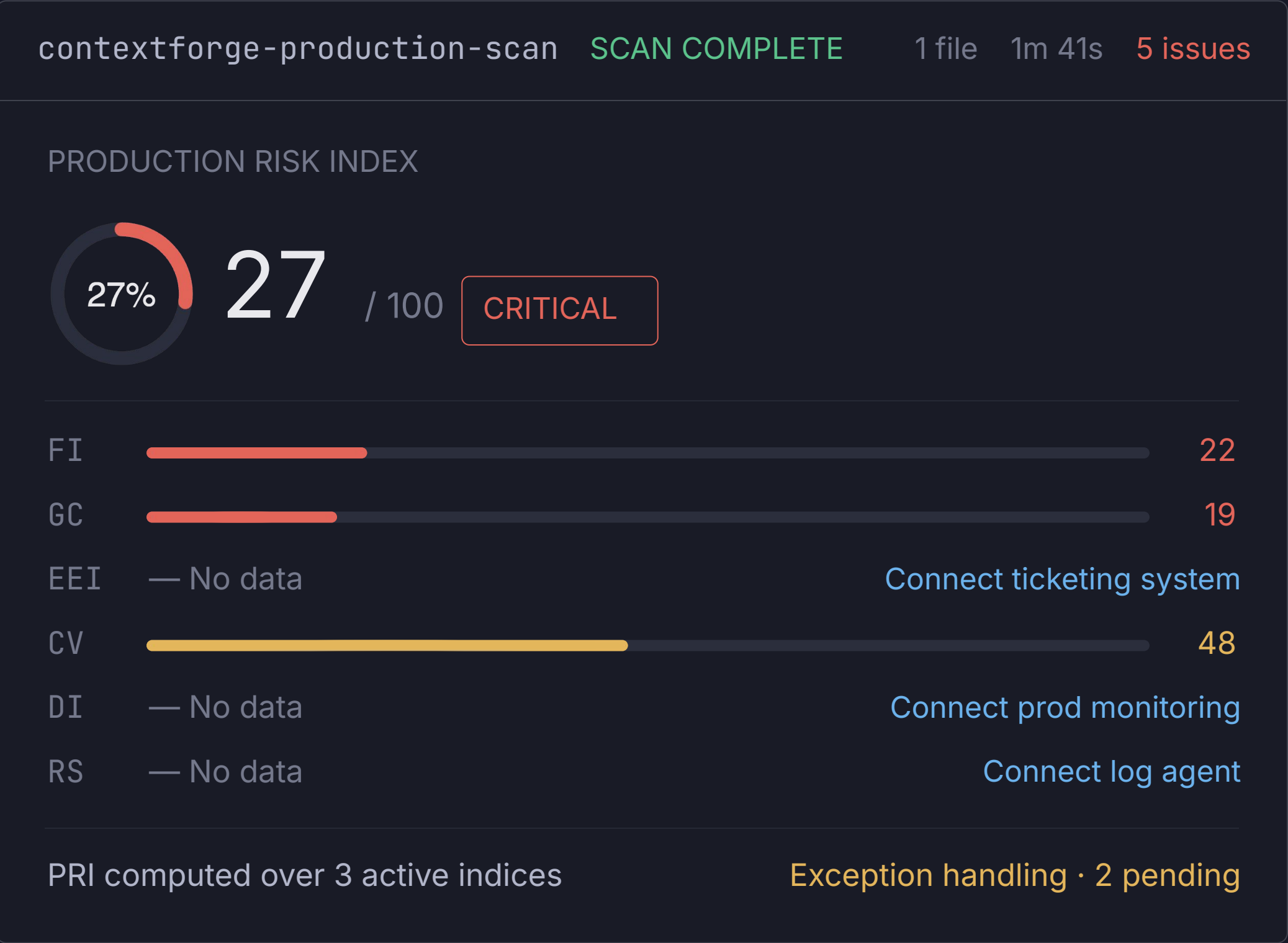
Generic exception-handling or database advice

MISS

Anything else — and no reinterpretation afterward

Blind code scan.

Historical vulnerable source only. No incident, no load test, no diagnosis, no instruction.



WHAT TOMOSU WAS NOT TOLD

- That this was an IBM bug
- That a 3,000-user load test existed
- That PostgreSQL connections were stuck
- What kind of issue to look for

From Tomosu's perspective: application code, evaluated for production reliability risk.

Tomosu named it as a reliability weakness.

DIRECT HIT

finding · token_scoping.py

high severity

GRACEFUL CLEANUP

RESOURCE LIFECYCLE

DATABASE SESSION

Unsafe database-session lifecycle: an operation performed before the session is closed can fail, leaving the connection unreleased.

Identified from the historical source alone

NO INCIDENT CONTEXT

AGAINST THE LOCKED CRITERION

- ✓ Names the database-session cleanup path
- ✓ States the preceding operation can fail
- ✓ Ties it to unreliable connection release

Adjacent Miss

Direct hit.

Connection release becomes unconditional.

Tomosu was not reproducing IBM's patch — it had not seen it. It corrected the same reliability invariant the load-test investigation eventually exposed.

recommendation · graceful cleanup

proposed change

finally:

~~-db.commit()~~

~~-db.close()~~

+db.close()

Read-oriented path — no commit required

RELEASE GUARANTEED

Different code. Same reliability property.

VULNERABLE

```
finally:  
    db.commit()  
    db.close()
```

Release depends on commit

IBM FIX

```
finally:  
    try:  
        db.commit()  
    finally:  
        db.close()
```

Commit preserved, close protected

TOMOSU · BLIND

```
finally:  
    db.close()
```

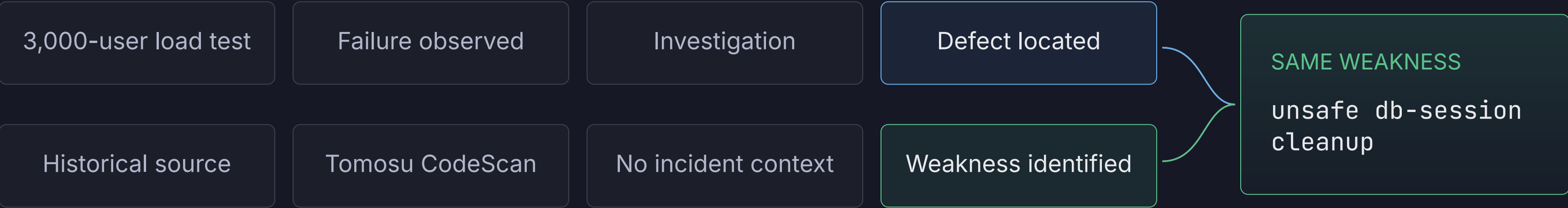
Commit dependency removed

Same reliability invariant: release must not depend on commit succeeding.

Two paths to the same weakness.

Static analysis does not replace load testing. It moves some of the learning earlier.

PATH ONE · DISCOVERED IN PRODUCTION CONDITIONS



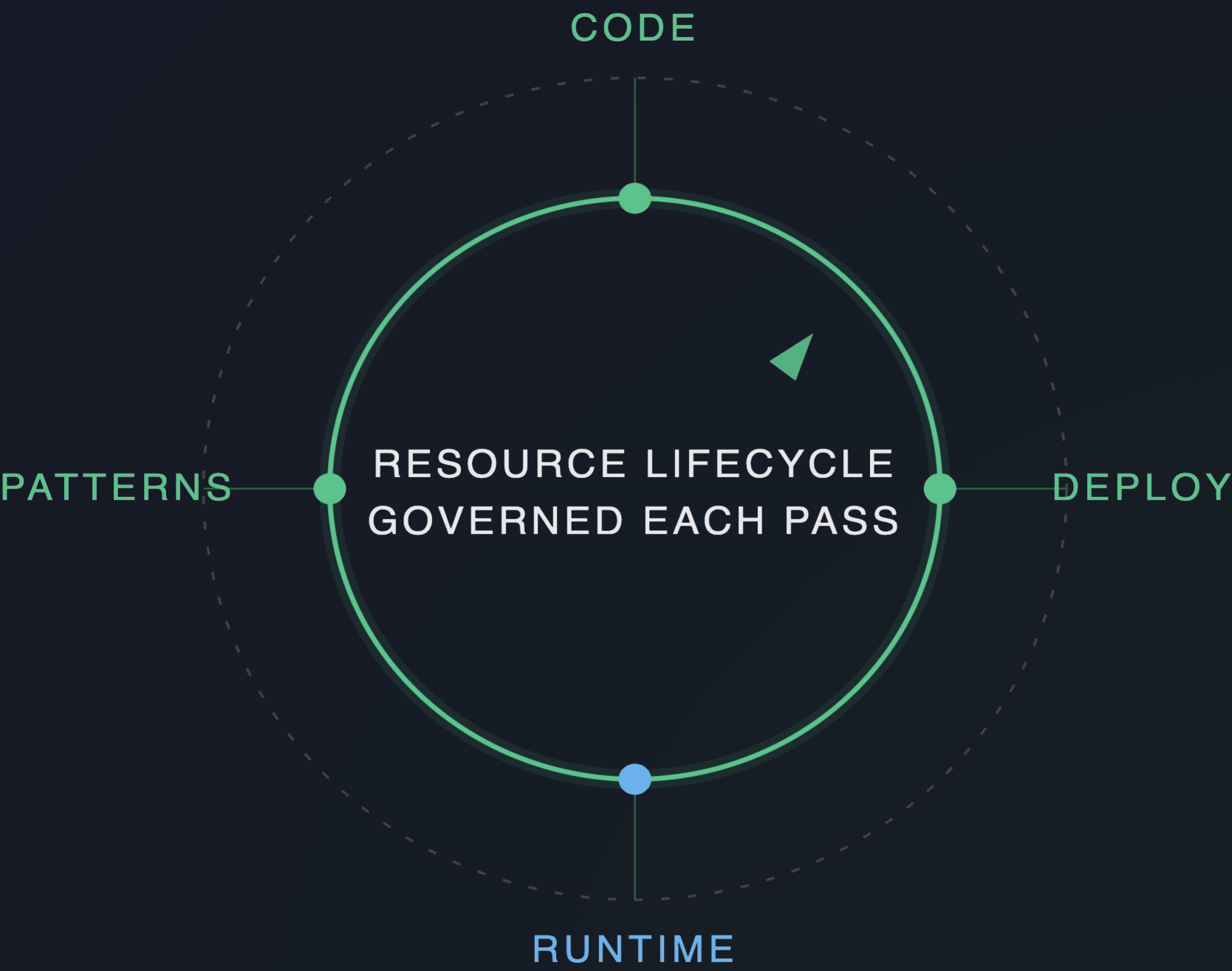
PATH TWO · IDENTIFIED FROM CODE

A weakness exposed by high-concurrency testing was also identifiable directly from the application source. That is the gap Production Reliability Governance addresses.

Reliability is a lifecycle, not a one-time code review.

Code carries assumptions about resources, retries, timeouts, dependencies and failure paths. Deployment changes the environment those assumptions run in. Runtime exposes what happens under real traffic and real failures.

CODE	Resource-lifecycle analysis
DEPLOY	Readiness and risk context
RUNTIME	Production signals under real load
BACK	Runtime insight returns to code



What will your load test find?

And could your code tell you first?