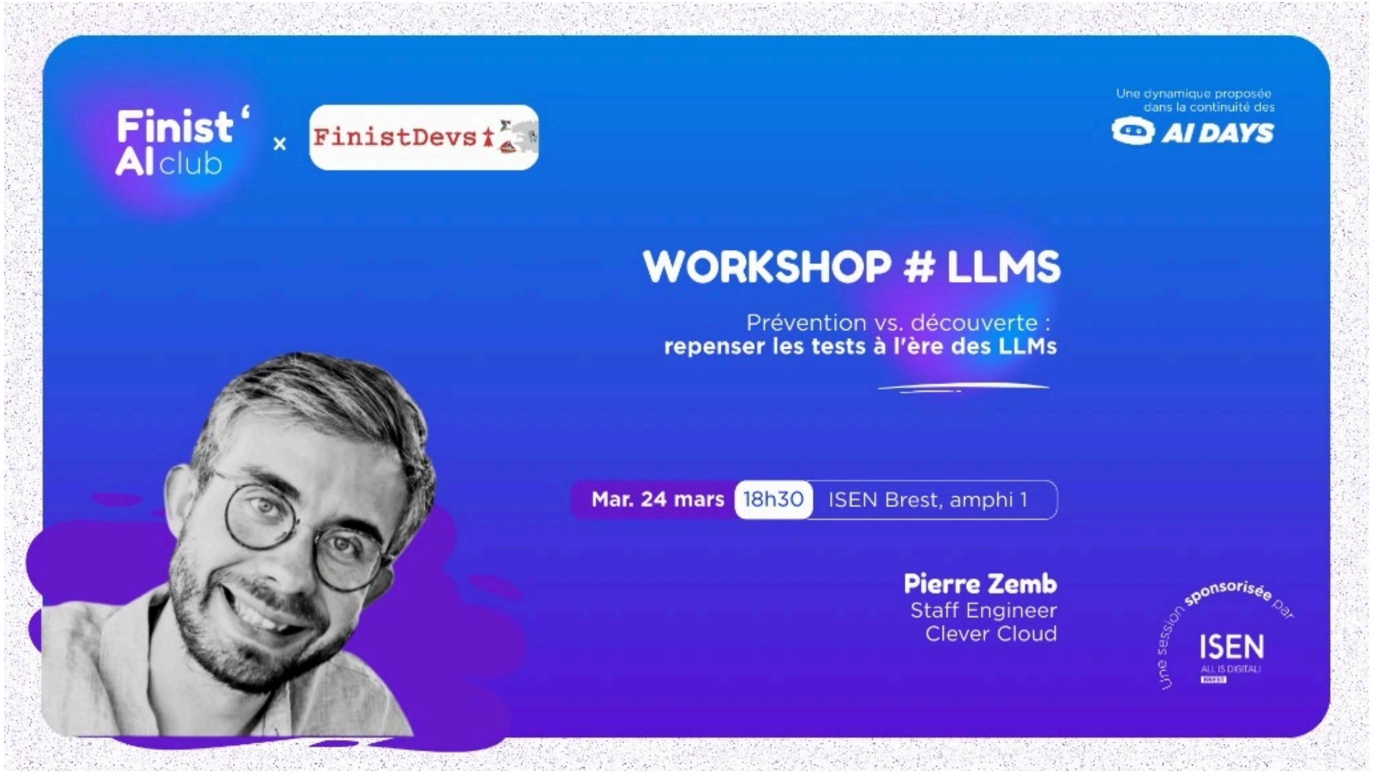


Finist' AI club × FinistDevs

Une dynamique proposée dans la continuité des AI DAYS

WORKSHOP # LLMS

Prévention vs. découverte :
repenser les tests à l'ère des LLMs

Mar. 24 mars 18h30 ISEN Brest, amphi 1

Pierre Zemb
Staff Engineer
Clever Cloud

Une session sponsorisée par ISEN ALL IS DIGITAL! BREST

#finistDevs #finistAI #yncrea

FinistDevs

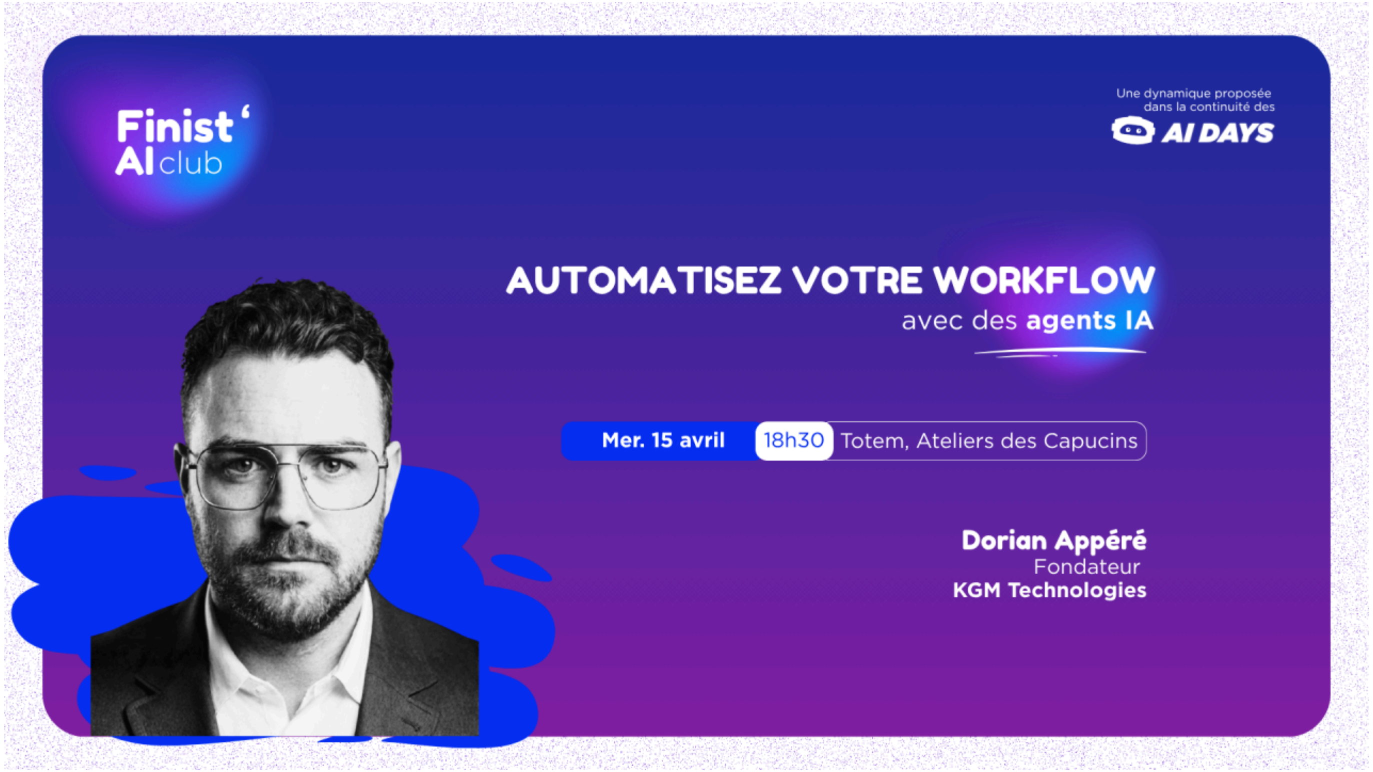
Soirée FinistDevs

Mardi 14 avril 2026

18h30

Chez zenika brest

GDG Finistère



Finist' AI club

Une dynamique proposée dans la continuité des AI DAYS

AUTOMATISEZ VOTRE WORKFLOW
avec des **agents IA**

Mer. 15 avril 18h30 Totem, Ateliers des Capucins

Dorian Appéré
Fondateur
KGM Technologies

Testing: Prevention vs Discovery 🔬

rethinking testing in the age of LLMs 🤔

Pierre Zemb — Clever Cloud

$ whoami 👋

- Pierre Zemb — Staff Engineer @ Clever Cloud 🇫🇷
- Former ISEN student
- Specialized in distributed systems
  - Building, contributing, operating…
- OSS maintainer
- Co-leading the FinistDevs
- Squash player

$ (also) whoami 👋

One of my most "wait what" bugs 🤯

Story: Network partition then NullPointerException on restart

1. Normal day, then: network partition + disk full on journal nodes (multiple cascading failures)
2. 70-machine Hadoop cluster goes belly-up
3. Can’t self-heal, reboot, **NullPointerException at startup**
4. Bug was known. Patched in newer HDFS version.
5. Emergency: backport patch, recompile, redeploy on critical 70 machines cluster

The question: Why does a NullPointerException happen during *recovery*?

Dev vs Prod 🚗

Learning to drive ≠ Driving in real-life

- 📝 Dev = passing the theory exam
- 🏎️ Prod = driving in Paris rush hour
- Recovery paths, split brain, cascading failures — none of this exists on localhost
- And more importantly:
  - **how can we test any fix?**
  - How can we **cultivate a production-oriented culture** for developers?

🤖 LLMs generate code faster than ever

More code, faster. Same blind spots. More potential bugs.

Amazon holds engineering meeting following AI-related outages, March 2026

> "Trend of incidents" with "high blast radius" and "Gen-AI assisted changes" — Amazon internal briefing, reported by the Financial Times

Junior and mid-level engineers now require senior sign-off for any AI-assisted changes.

SWE-CI: Evaluating Agent Capabilities in Maintaining Codebases via Continuous Integration — Chen et al., 2026

> "Most models achieve a zero-regression rate below 0.25"

> "Current LLMs still struggle to sustain code quality over extended evolution, particularly in controlling regressions."

Cheap code, expensive what? 💡

> "Code is becoming like fast food. Cheap, fast, everywhere." — João Alves

- 🧠 **Knowing what to build** — specs, design, architecture
- 🔍 **Knowing if it works** — testing, verification, observability
- ⚖️ **Knowing when it breaks** — failure modes, edge cases, regressions

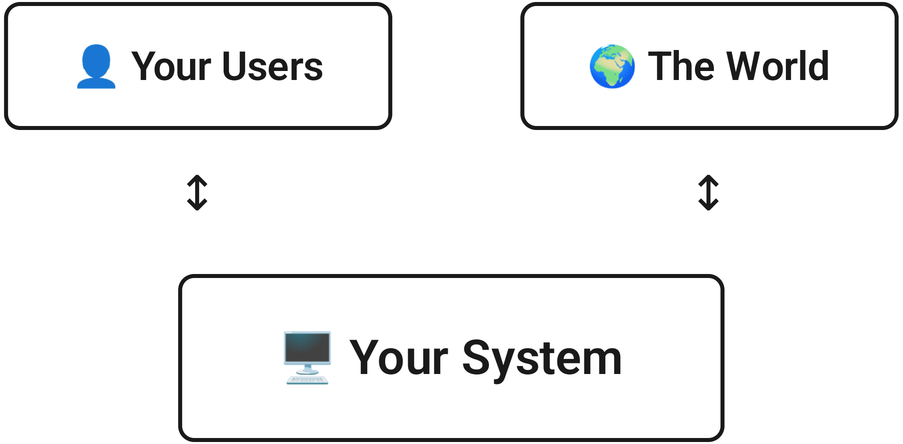
LLMs **amplify expertise, they don’t replace it.**

Let’s focus on quality 🔍

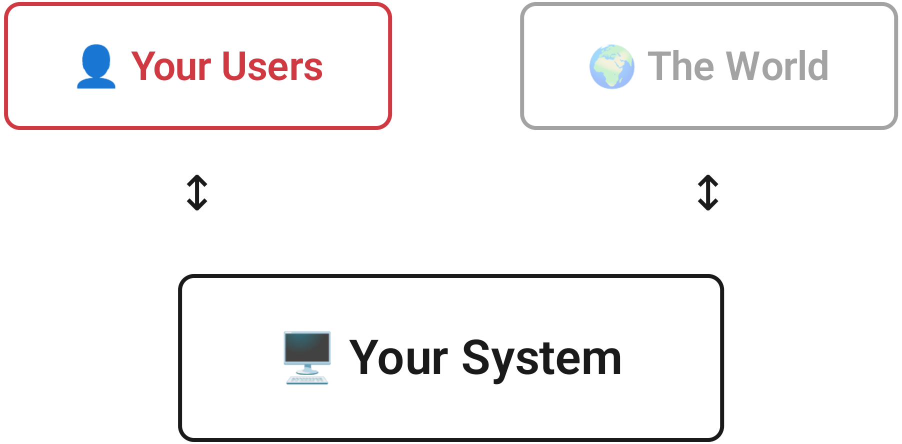
The goal isn’t faster code. It’s **better software**.

We can generate code. But how do we verify it actually works?

Your system interacts with two things 🖥️



Can we improve user-related code? 🤔



👤 Your Users

🌍 The World

↕ ↕

🖥️ Your System

Let’s test our users 😱

Let’s imagine the following e-commerce API that supports:

- 👤 User Type: Guest, Logged-in, Premium
- 💳 Payment Method: Credit Card, PayPal, Apple Pay, Gift Card
- 🚚 Delivery Option: Standard, Express, In-Store Pickup
- 🎁 Promotion Applied: Yes, No
- 📦 Inventory Status: In Stock, Low Stock, Out of Stock
- 💱 Currency: USD, EUR, GBP

To cover all possibilities, we need to write **3 × 4 × 3 × 2 × 3 × 3 = 648 unique test cases** 😱

(just to cover the happy path)

This is why E2E testing is hard 🤯

Let’s imagine the following e-commerce API that supports:

- 👤 User Type: Guest, Logged-in, Premium, **Business**
- 💳 Payment Method: Credit Card, PayPal, Apple Pay, Gift Card, **Bank Transfer**
- 🚚 Delivery Option: Standard, Express, In-Store Pickup, **Same-Day**
- 🎁 Promotion Applied: Yes, No, **Expired Promo**
- 📦 Inventory Status: In Stock, Low Stock, Out of Stock, **Preorder**
- 💱 Currency: USD, EUR, GBP, **JPY**

To cover all possibilities, we need to write **4 × 5 × 4 × 3 × 4 × 4 = 3,840 unique test cases** 😱

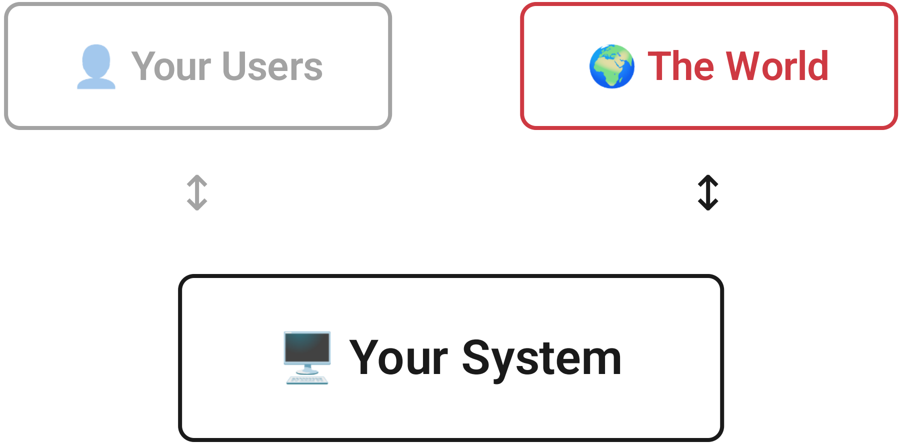
(just to cover the happy path)

We can’t test everything 🤯

- You test what you imagine. Bugs hide in combinations you didn’t.
- Manual or naive testing strategies can’t keep up

We need a way to explore!

Can our code handle the world? 🌍



👤 Your Users

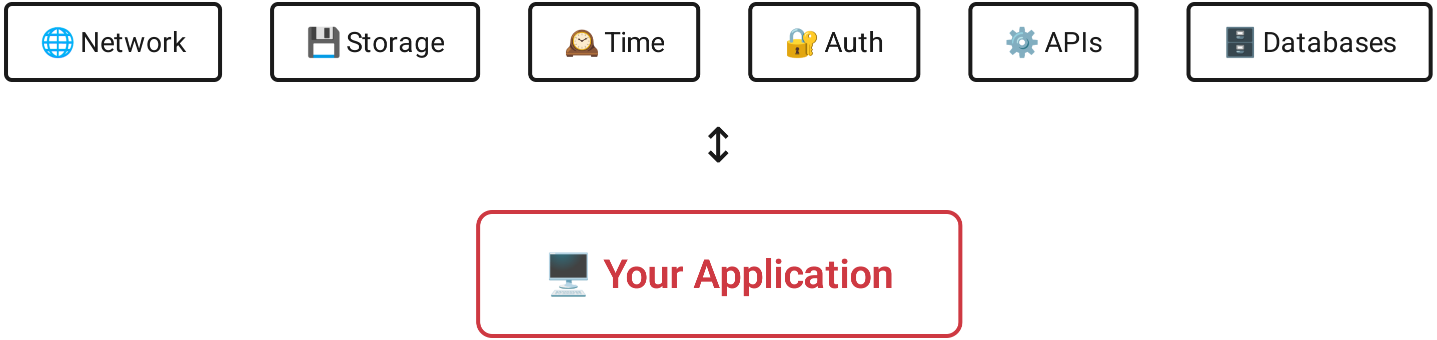
🌍 The World

↕

↕

🖥️ Your System

What can go wrong? 🌍



🌐 Network

💾 Storage

🕰️ Time

🔐 Auth

⚙️ APIs

🗄️ Databases

↕

🖥️ **Your Application**

You probably handle the happy path. What about the other **3,839 combinations**?

What do you believe about your system? 🤔

"The network is reliable" 🌐

An Analysis of Network-Partitioning Failures — Alquraan et al., OSDI 2018

- **80%** of partition failures have catastrophic impact
- **27%** lead to data loss
- **90%** are silent — no log, no alert, nothing 🤫
- **21%** cause permanent damage that persists after the partition heals
- **83%** need 3+ events to manifest — **the sequential luck problem**

"My data is safe on disk" 💾

- *SSD Reliability* FAST 2020
- *Data Corruption in the Storage Stack* FAST 2008

- Silent corruption
  - 0.031% of SSDs per year
  - 1.4% of enterprise HDDs per year

- Misdirected I/O
  - 0.023% of SSDs per year
  - 0.466% of Nearline HDDs per year

"I called fsync, my data is safe" 💾

Can Applications Recover from fsync Failures? — Rebello et al., ATC 2020

> "Although applications use many failure-handling strategies, none are sufficient: fsync failures can cause catastrophic outcomes such as data loss and corruption."

> "All three file systems mark pages clean after fsync fails, rendering techniques such as application-level retry ineffective."

Tested on PostgreSQL, LMDB, LevelDB, SQLite, Redis — **none handle fsync failures correctly**.

"Consensus will recover from crashes" 💾

Protocol-Aware Recovery for Consensus-Based Storage — Alagappan et al., FAST 2018

The researchers injected **2,401 corruption scenarios** into ZooKeeper:

Scenario	Result
Targeted corruptions	Recovers in **46/2,401** cases (1.9%) 💀
Random block corruptions	**~30%** end in data loss or unavailability
Block errors	**~50%** cluster unavailability — restarting loops forever

Why? ZooKeeper truncates corrupted log entries. If the corrupted node then forms a majority with lagging nodes → **committed data is silently lost**.

"I have 3 replicas, I’m safe" 🛡️

Redundancy Does Not Imply Fault Tolerance — Ganesan et al., FAST 2017

> "A single file-system fault can cause catastrophic outcomes such as data loss, corruption, and unavailability."

Tested on 8 systems: Redis, ZooKeeper, Cassandra, Kafka, MongoDB, RethinkDB, CockroachDB, LogCabin.

Kafka: one corrupted log entry on the leader → leader ignores it, instructs followers to do the same → followers hit fatal assertion → **entire cluster unavailable + data loss** 💥

All problems observed at R=1 **persist at R=3**.

"We handle all our errors" ⚠️

Simple Testing Can Prevent Most Critical Failures — Yuan et al., OSDI 2014

> "Almost all catastrophic failures (**92%**) are the result of incorrect handling of non-fatal errors explicitly signaled in software."

> "**35%** of the catastrophic failures are caused by trivial mistakes in error handling logic — ones that simply violate best programming practices."

> "A majority of the production failures (**77%**) can be reproduced by a unit test."

"We don’t have concurrency bugs" ⚠️

TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs — Leesatapornwongsa et al., ASPLOS 2016

> "More than three quarters of the bugs involve some background protocols" — not the foreground protocols developers typically test.

> "DC bugs are triggered mostly by untimely messages (**64%**) and sometimes by untimely faults/reboots (**32%**), and occasionally by a combination of both."

"We just retry on failure" 🔄

Metastable Failures in the Wild — Huang et al., OSDI 2022

> "The sustaining effect keeps the system in the metastable failure state even after the trigger is removed."

> "The most common sustaining effect is due to the retry policy, affecting more than **50%** of the studied incidents."

> "It naturally arises from the optimizations for the common case that lead to sustained work amplification."

22 incidents from 11 organizations — outages: **1.5 to 73.5 hours** ⏰ — at least **4 of 15 major AWS outages** in the last decade.

"We follow the documentation" 📖

Jepsen: MariaDB Galera Cluster 12.1.2

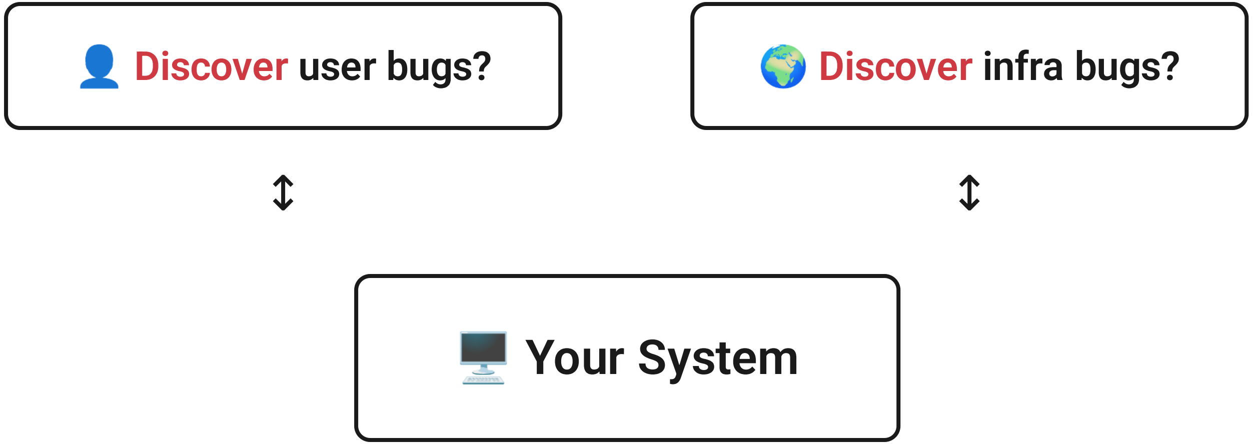
MariaDB claims: "Galera’s tx_isolation is between Serializable and Repeatable Read."

Jepsen found — even in **healthy clusters** with zero faults:

- 💀 **Lost committed transactions**
- 💀 **Lost Updates**
- 💀 **Stale Reads**

> "It appears weaker than Read Uncommitted."

We need to explore 🚀

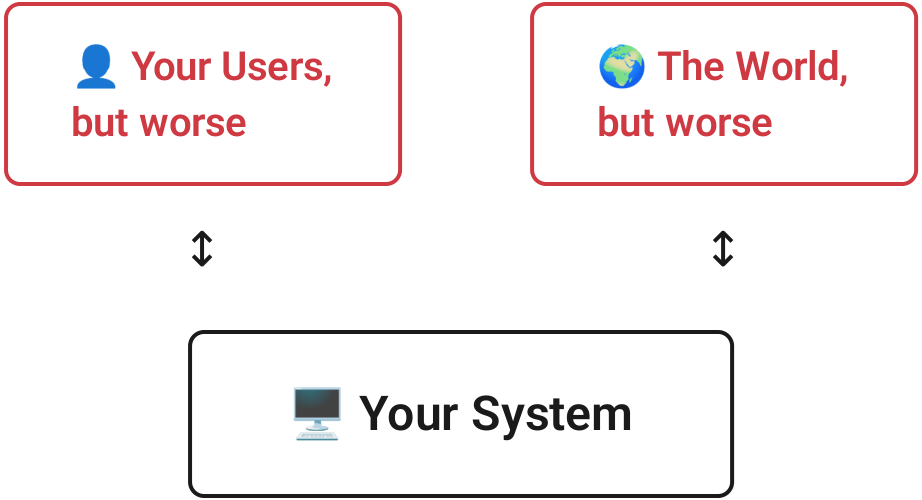


LLMs excel at prevention — give them a spec, they write tests. But discovery requires different infrastructure.

What developers want from tests ✅

- ⚡ **Fast** — not 10 min CI, not waiting for containers
- 🔍 **Debuggable** — not "works on my machine"
- 🏗️ **Full system** — not just isolated units
- 💪 **Robust** — no flaky tests, no `sleep()` in tests

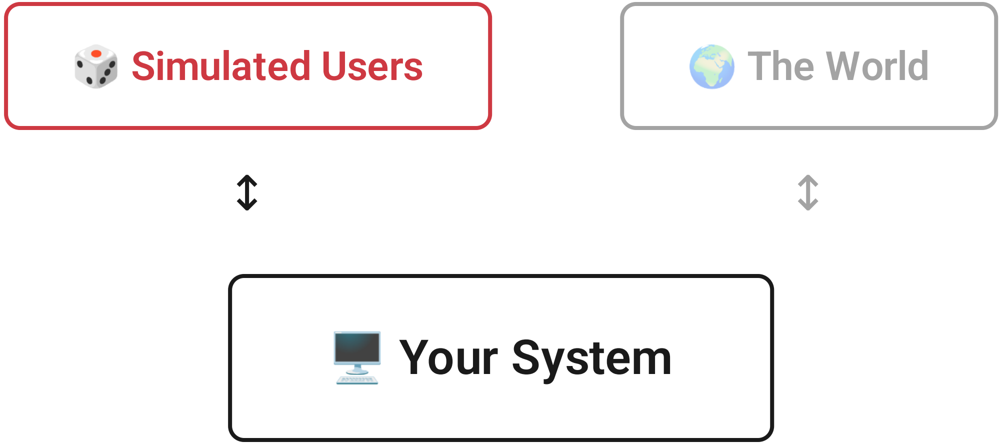
We must test the worst 🦹



If your system survives this, we can have more confidence it will **survive production** 🦸

Let’s tackle these **one at a time**.

How to simulate our users? 🤔



🎲 Simulated Users

🌍 The World

↕

↕

🖥️ Your System

Remember our e-commerce API? 🛒

3,840 test cases — and that was just the happy path.

Writing them by hand? Not an option.

What if we could **generate** them instead?

From tests to test generators 🎲

```java
// ✅ One generator = all combinations
enum UserType { GUEST, LOGGED_IN, PREMIUM, BUSINESS }
enum PaymentMethod { CREDIT_CARD, PAYPAL, APPLE_PAY, GIFT_CARD, BANK_TRANSFER }
// …

Random rand = new Random();
UserType user = pickRandom(rand, UserType.values());
PaymentMethod payment = pickRandom(rand, PaymentMethod.values());
// …
checkout(user, payment, shipping, promo, stock, currency);
```

Add a feature? Add one enum value, not 100 tests.

Properties, not test cases 🧪

```
// Don't assert specific values — assert relationships
for (int i = 0; i < 1000; i++) {
  UserType user = pickRandom(rand, UserType.values());
  assertEquals(user.isAuthenticated(), user.canUseSavedCards());
}
```

Properties look like specs. They compile as code. They hold for **all** inputs.

Property-based testing 🧪

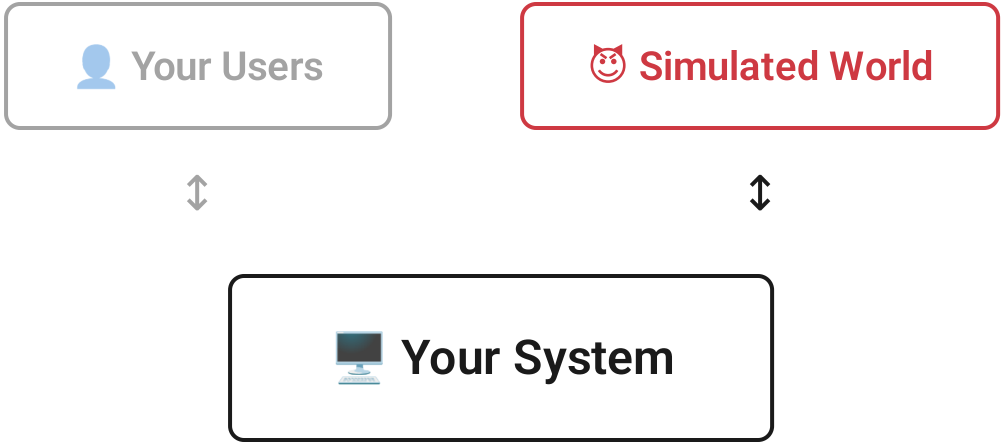
- 🐍 Python: Hypothesis
- ☕ Java: jqwik
- 🦀 Rust: proptest
- λ Haskell: QuickCheck

The recipe:

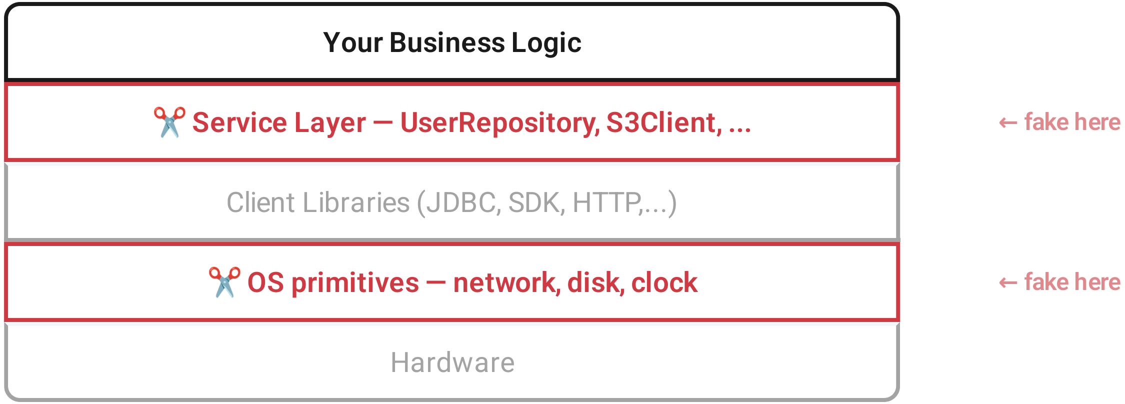
- 🎲 **Randomize inputs** — let the computer explore combinations you'd never write by hand
- 🧪 **Validate properties** — assert relationships that must hold for all inputs

How to simulate the world? 🌍

Replace real dependencies with **fakes** — lightweight, in-memory implementations that actually hold state.



Step 1: Identify fallible parts of your code 🔌



Your Business Logic

✂️ Service Layer — UserRepository, S3Client, ... ← fake here

Client Libraries (JDBC, SDK, HTTP,...)

✂️ OS primitives — network, disk, clock ← fake here

Hardware

Don’t fake PostgreSQL. Fake your **access** to it.

Step 2: Define the boundary 🎭

```java
interface UserRepository {
  void save(User user);
  Optional<User> findById(long id);
  List<User> findAll();
}
```

Your code depends on this interface, not on PostgreSQL.

Two implementations, one interface

🗄️ Production

```
class PostgresUserRepository
    implements UserRepository {
  void save(User user) {
    jdbc.execute(
      "INSERT INTO users ... ", user);
  }
}
```

🎭 Simulation

```
class FakeUserRepository
    implements UserRepository {
  Map<Long, User> store = new HashMap<>();
  void save(User user) {
    store.put(user.id(), user);
  }
}
```

Same interface. One talks to Postgres. One lives in memory. **Your system can’t tell the difference.**

Step 3: Now break everything 💥

```
class FakeUserRepository implements UserRepository {
  Map<Long, User> store = new HashMap<>();
  Random rand;

  void save(User user) {
    if (rand.nextFloat() < 0.5)
      throw new StorageException("Connection lost");
    store.put(user.id(), user);
  }
}
```

Same fake from Step 1. Now it fights back.

Be worse than production 😈

Remember MariaDB Galera? Jepsen found **stale reads in healthy clusters**.

Your fake should be **worse**: 50% stale reads, not 0.1%.

If your system survives this, it **survives production**.

What if we combined both? 🧩

- 🎭 Fakes that **control** the world
- 💥 Chaos that **injects failures** everywhere

Run it all at once, with random inputs, checking properties…

The price of determinism 🔧

Sources of non-determinism you must eliminate:

- 🧵 **Thread scheduling** → single-threaded cooperative execution
- 🎲 **Random numbers** → seeded PRNG
- 🕰️ **System time** → simulated clock
- 📋 **HashMap iteration** → deterministic data structures
- 🌐 **I/O** → simulated through fakes

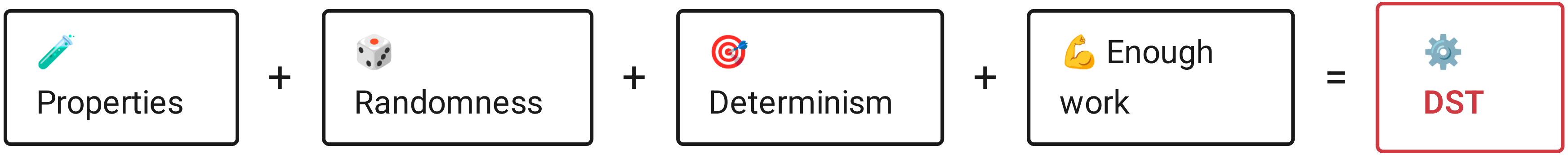
The payoff:

```
u64 seed → entire execution determined
Same seed = same bugs. Every time.
```

A failing seed is a time machine ⏪

Teach a computer to test 🤖

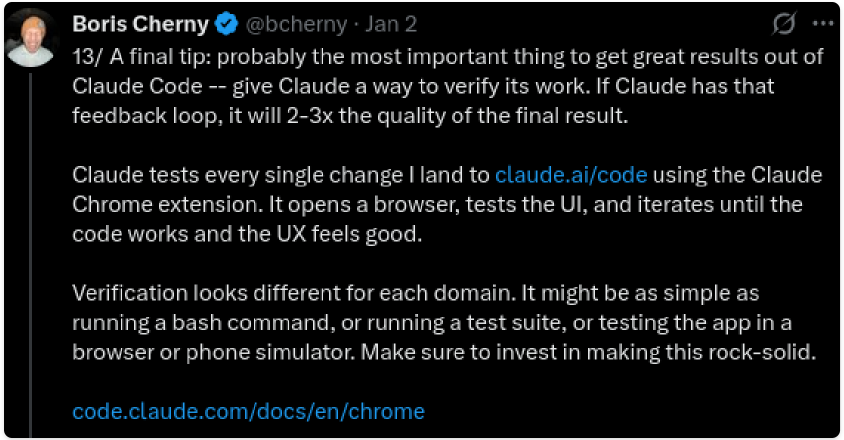
Give a computer a test, it finds a bug. Teach a computer to test, it finds bugs forever.



🧪 Properties + 🎲 Randomness + 🎯 Determinism + 💪 Enough work = ⚙️ **DST**

Deterministic Simulation Testing (DST): Simulated users + simulated world + seeded determinism = reproducible bug discovery at scale.

DST: The ultimate LLM feedback loop 🤖🔁



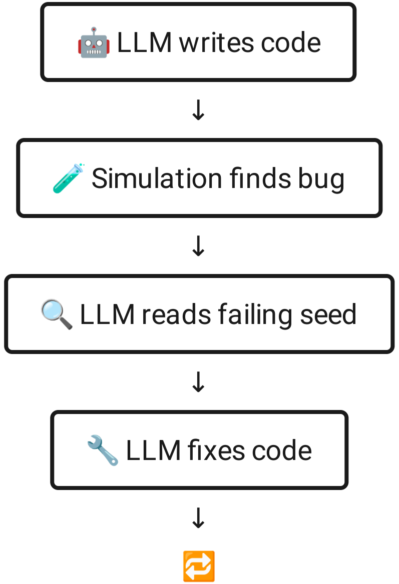
Boris Cherny @bcherny · Jan 2

13/ A final tip: probably the most important thing to get great results out of Claude Code -- give Claude a way to verify its work. If Claude has that feedback loop, it will 2-3x the quality of the final result.

Claude tests every single change I land to claude.ai/code using the Claude Chrome extension. It opens a browser, tests the UI, and iterates until the code works and the UX feels good.

Verification looks different for each domain. It might be as simple as running a bash command, or running a test suite, or testing the app in a browser or phone simulator. Make sure to invest in making this rock-solid.

code.claude.com/docs/en/chrome



🤖 LLM writes code

↓

🧪 Simulation finds bug

↓

🔍 LLM reads failing seed

↓

🔧 LLM fixes code

↓

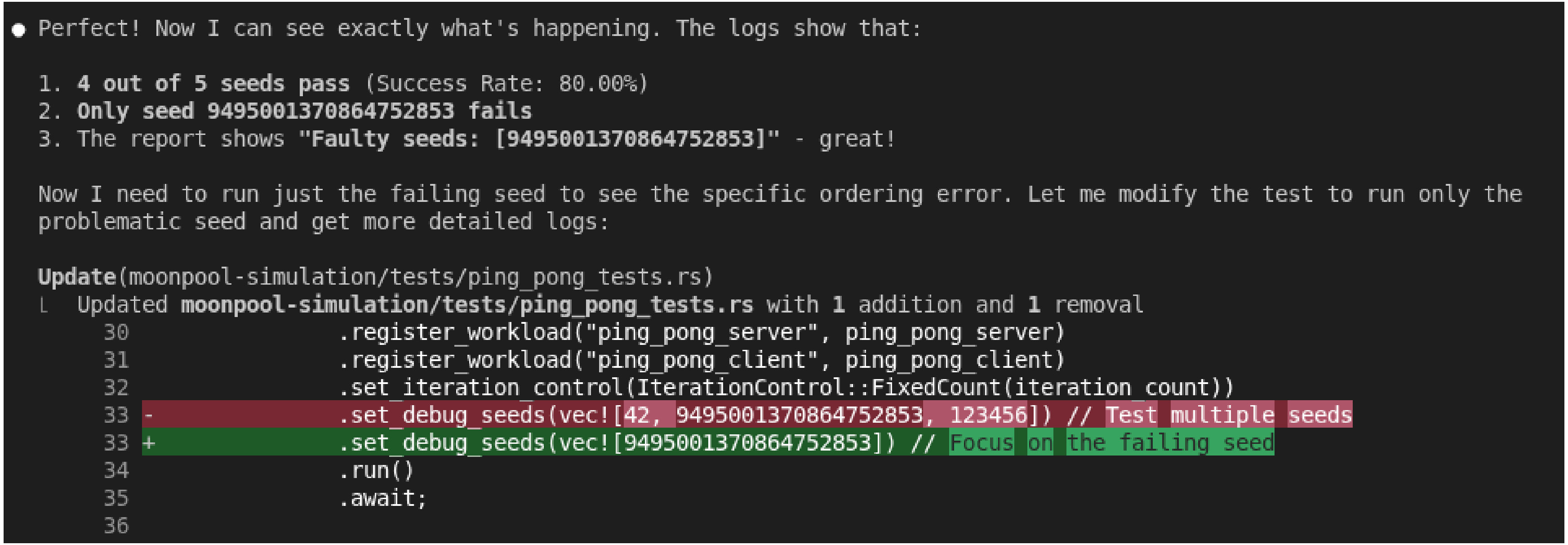
🔁

DST as **a feedback loop** works as well for:

- junior engineers
- AI

because it lets them **discover what they don’t know**.

🤖 Claude fixing software with DST



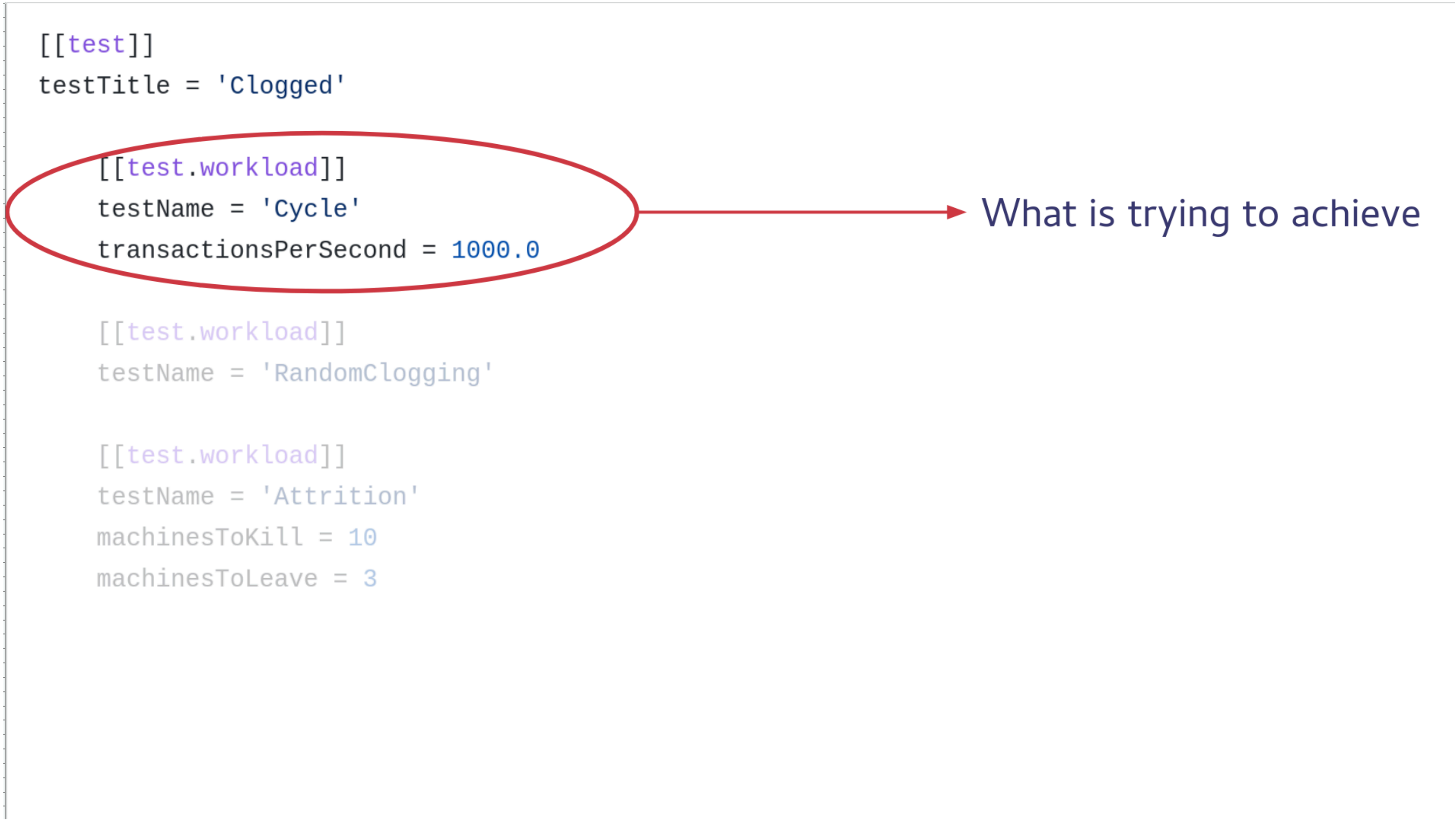
```
● Perfect! Now I can see exactly what's happening. The logs show that:

  1. 4 out of 5 seeds pass (Success Rate: 80.00%)
  2. Only seed 9495001370864752853 fails
  3. The report shows "Faulty seeds: [9495001370864752853]" - great!

  Now I need to run just the failing seed to see the specific ordering error. Let me modify the test to run only the
  problematic seed and get more detailed logs:

  Update(moonpool-simulation/tests/ping_pong_tests.rs)
  ⎿  Updated moonpool-simulation/tests/ping_pong_tests.rs with 1 addition and 1 removal
                             .register_workload("ping_pong_server", ping_pong_server)
                             .register_workload("ping_pong_client", ping_pong_client)
                             .set_iteration_control(IterationControl::FixedCount(iteration_count))
          -                  .set_debug_seeds(vec![42, 9495001370864752853, 123456]) // Test multiple seeds
          +                  .set_debug_seeds(vec![9495001370864752853]) // Focus on the failing seed
                             .run()
                             .await;

```

Clever Cloud / Materia ☁️🦀

- ~90 employees — a **full cloud provider** with our own LB, Linux distro, and orchestrator
- Already have our hands full — then we decided to build a **distributed multi-model database** 🤯
  - Team grew from 1 to 6 persons
- The question isn't *why* — it's **how do you build something this hard with a small team?**
- The answer: **simulation-driven development** — write the workload first, then implement
  - All our stack is under simulation: KV, KMS, ETCD, workflow engine, leader election…

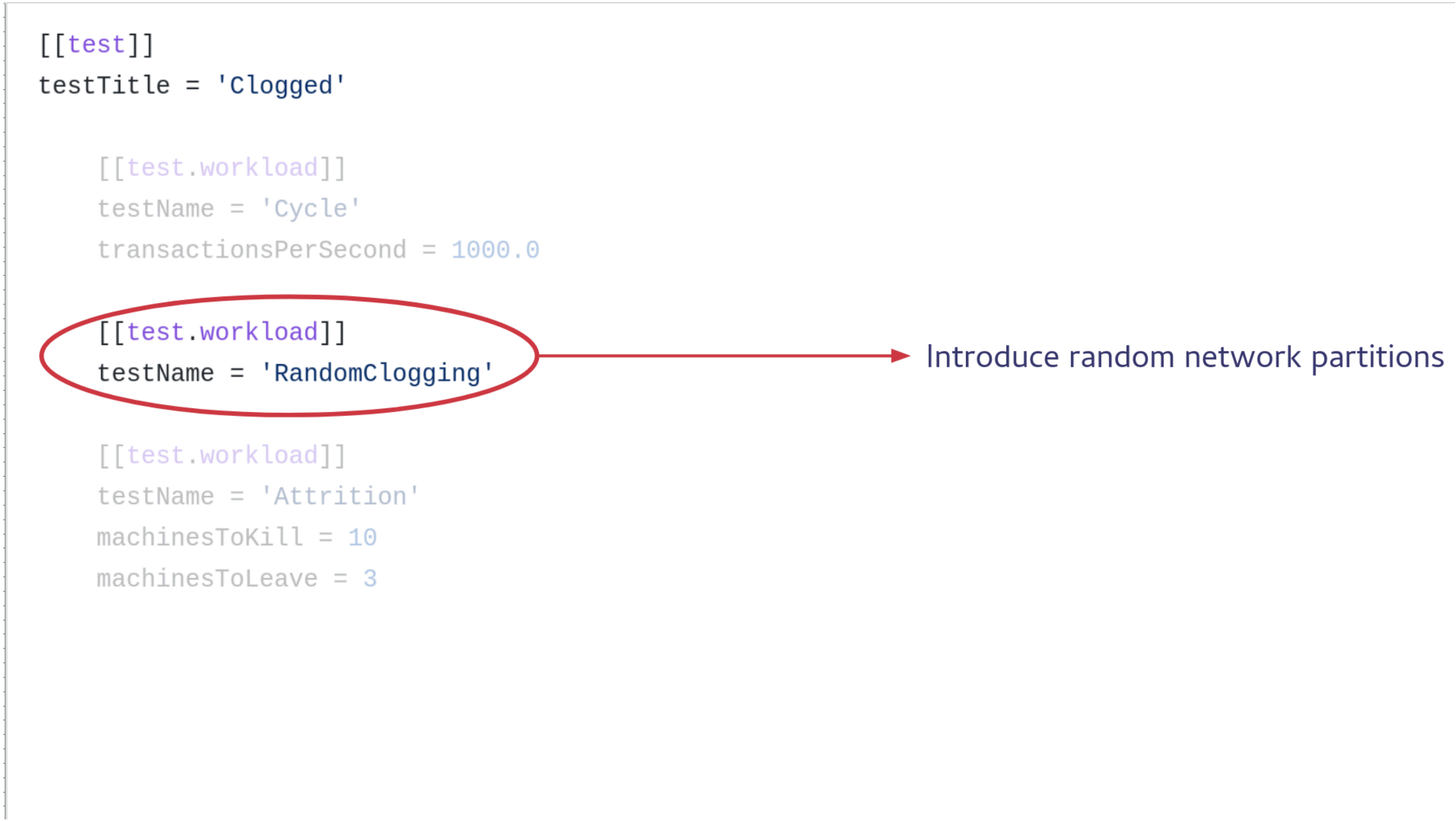
> "We would never have succeeded without simulation." 🦸‍♀️

FoundationDB’s simulation config 🔧



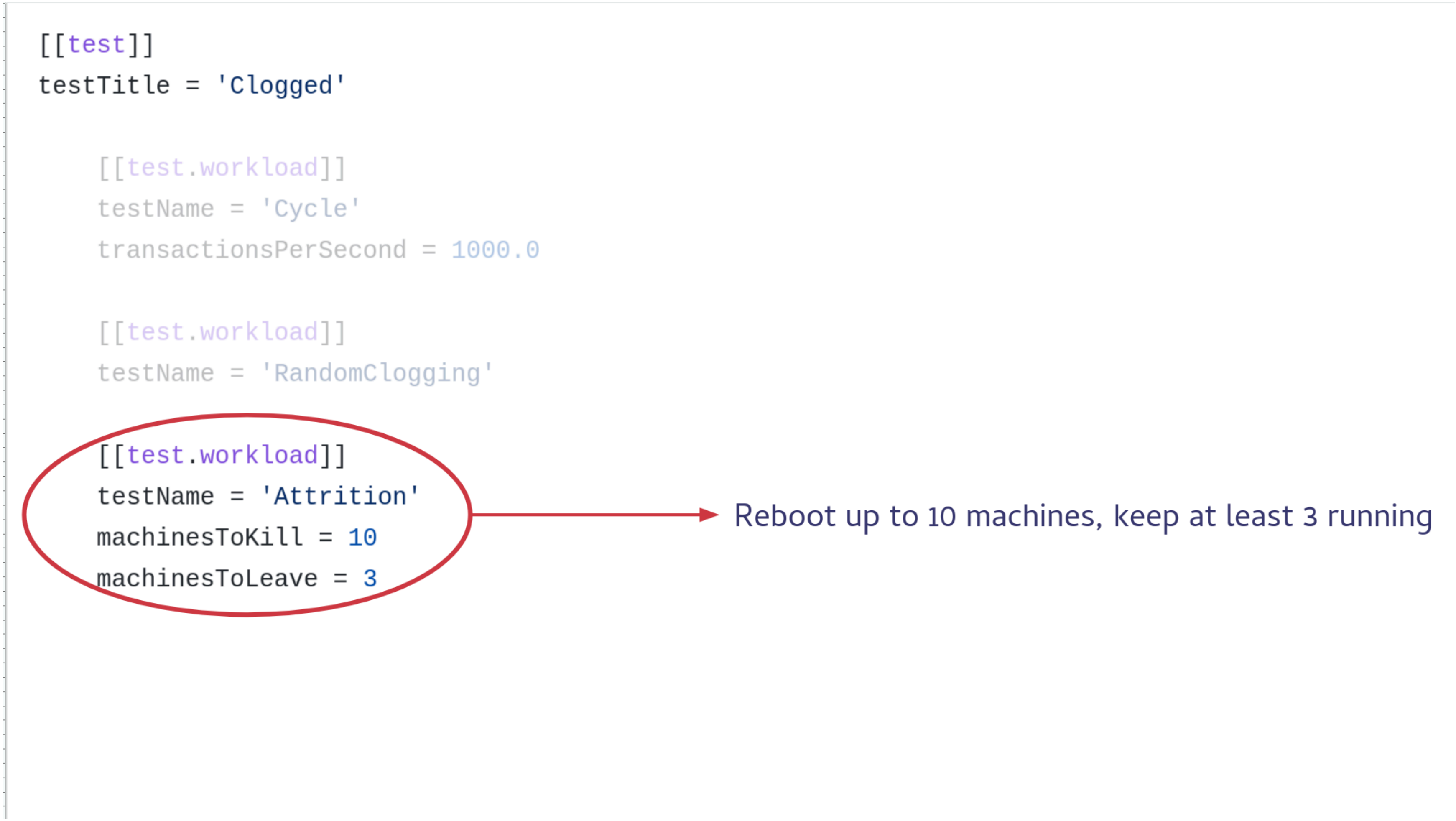
```
[[test]]
testTitle = 'Clogged'

    [[test.workload]]
    testName = 'Cycle'
    transactionsPerSecond = 1000.0

    [[test.workload]]
    testName = 'RandomClogging'

    [[test.workload]]
    testName = 'Attrition'
    machinesToKill = 10
    machinesToLeave = 3
```

What is trying to achieve



```
[[test]]
testTitle = 'Clogged'

    [[test.workload]]
    testName = 'Cycle'
    transactionsPerSecond = 1000.0

    [[test.workload]]
    testName = 'RandomClogging'

    [[test.workload]]
    testName = 'Attrition'
    machinesToKill = 10
    machinesToLeave = 3
```

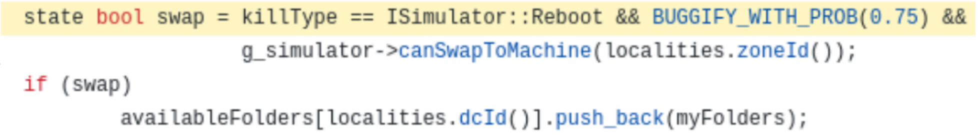
Introduce random network partitions



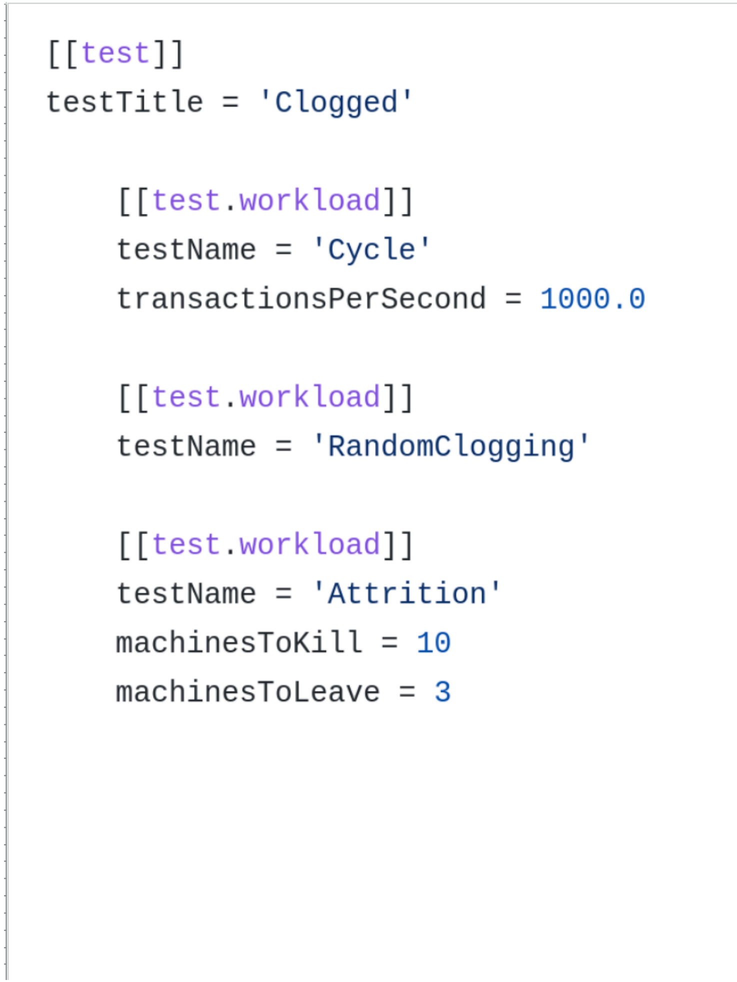
```
[[test]]
testTitle = 'Clogged'

    [[test.workload]]
    testName = 'Cycle'
    transactionsPerSecond = 1000.0

    [[test.workload]]
    testName = 'RandomClogging'

    [[test.workload]]
    testName = 'Attrition'
    machinesToKill = 10
    machinesToLeave = 3
```

Reboot up to 10 machines, keep at least 3 running

```
[[test]]
testTitle = 'Clogged'

    [[test.workload]]
    testName = 'Cycle'
    transactionsPerSecond = 1000.0

    [[test.workload]]
    testName = 'RandomClogging'

    [[test.workload]]
    testName = 'Attrition'
    machinesToKill = 10
    machinesToLeave = 3
```

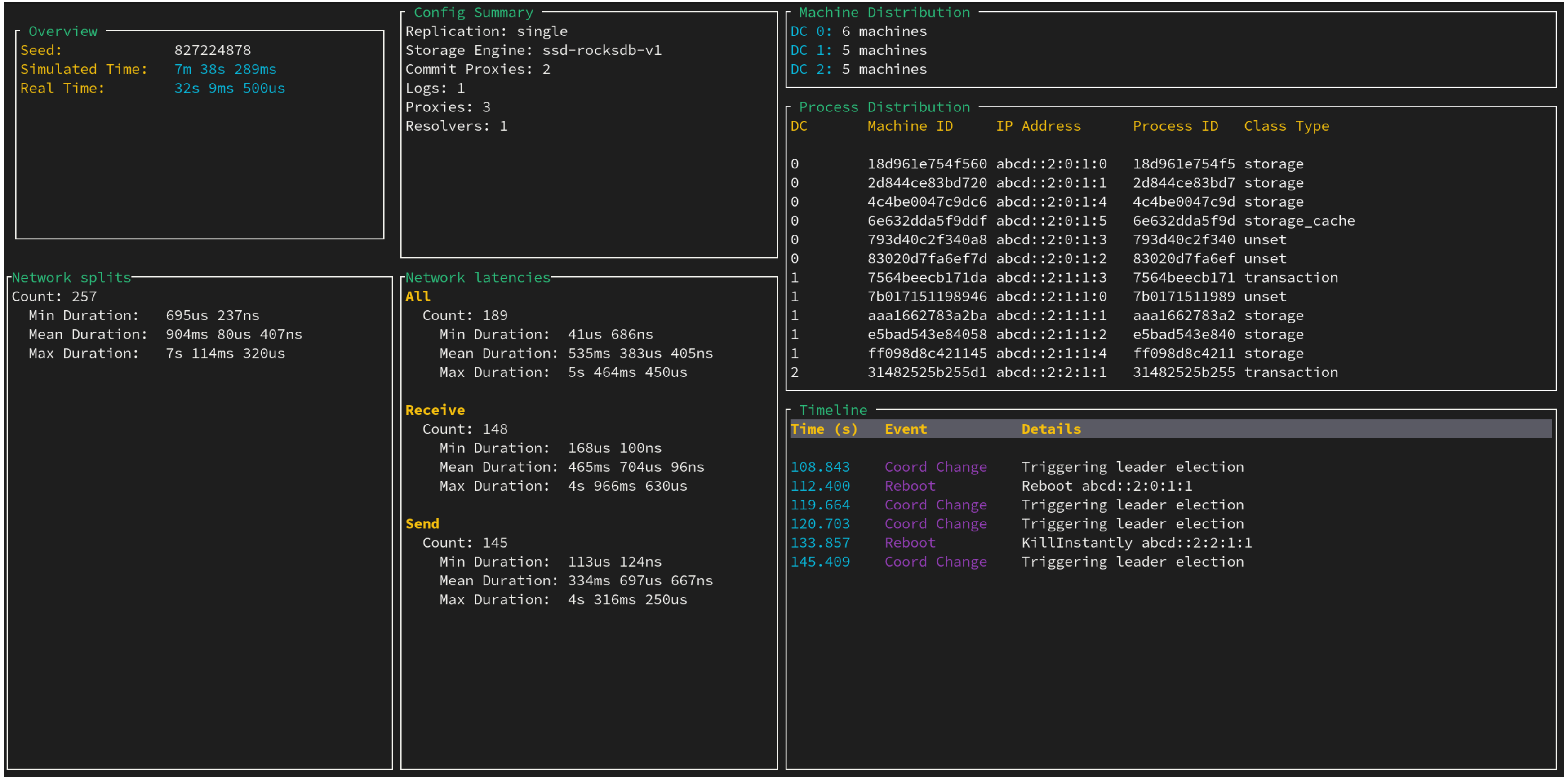


```
state bool swap = killType == ISimulator::Reboot && BUGGIFY_WITH_PROB(0.75) &&
                   g_simulator->canSwapToMachine(localities.zoneId());
if (swap)
        availableFolders[localities.dcId()].push_back(myFolders);
```



```
[[test]]
testTitle = 'Clogged'

    [[test.workload]]
    testName = 'Cycle'
    transactionsPerSecond = 1000.0

    [[test.workload]]
    testName = 'RandomClogging'

    [[test.workload]]
    testName = 'Attrition'
    machinesToKill = 10
    machinesToLeave = 3
```

Will happen **concurrently**



```
Overview
Seed:                827224878
Simulated Time:      7m 38s 289ms
Real Time:           32s 9ms 500us
```

```
Config Summary
Replication: single
Storage Engine: ssd-rocksdb-v1
Commit Proxies: 2
Logs: 1
Proxies: 3
Resolvers: 1
```

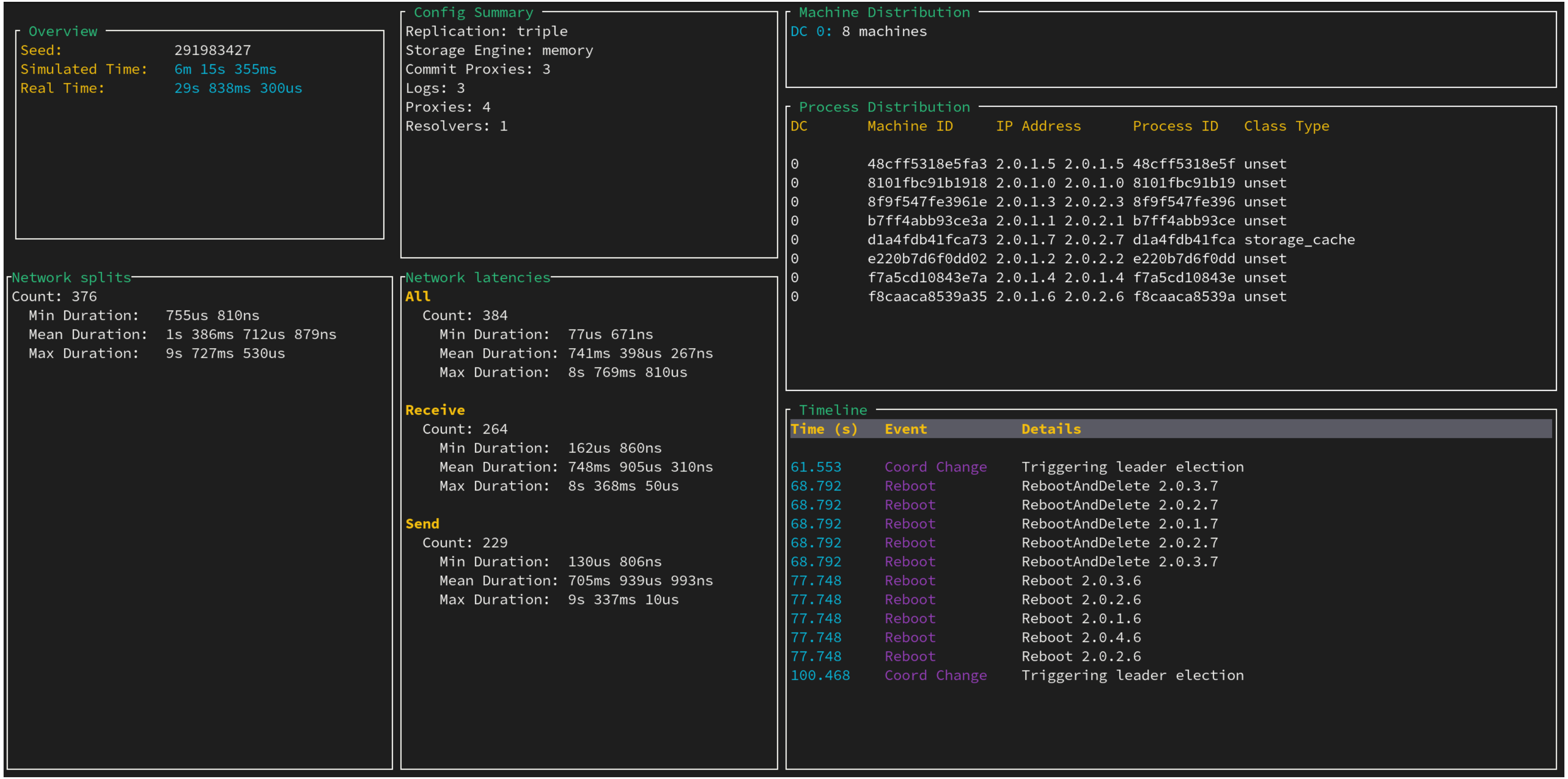
```
Machine Distribution
DC 0: 6 machines
DC 1: 5 machines
DC 2: 5 machines
```

```
Process Distribution
DC        Machine ID      IP Address       Process ID   Class Type

0         18d961e754f560  abcd::2:0:1:0    18d961e754f5 storage
0         2d844ce83bd720  abcd::2:0:1:1    2d844ce83bd7 storage
0         4c4be0047c9dc6  abcd::2:0:1:4    4c4be0047c9d storage
0         6e632dda5f9ddf  abcd::2:0:1:5    6e632dda5f9d storage_cache
0         793d40c2f340a8  abcd::2:0:1:3    793d40c2f340 unset
0         83020d7fa6ef7d  abcd::2:0:1:2    83020d7fa6ef unset
1         7564beecb171da  abcd::2:1:1:3    7564beecb171 transaction
1         7b017151198946  abcd::2:1:1:0    7b0171511989 unset
1         aaa1662783a2ba  abcd::2:1:1:1    aaa1662783a2 storage
1         e5bad543e84058  abcd::2:1:1:2    e5bad543e840 storage
1         ff098d8c421145  abcd::2:1:1:4    ff098d8c4211 storage
2         31482525b255d1  abcd::2:2:1:1    31482525b255 transaction
```

```
Network splits
Count: 257
  Min Duration:    695us 237ns
  Mean Duration:   904ms 80us 407ns
  Max Duration:    7s 114ms 320us
```

```
Network latencies
All
  Count: 189
    Min Duration:  41us 686ns
    Mean Duration: 535ms 383us 405ns
    Max Duration:  5s 464ms 450us

Receive
  Count: 148
    Min Duration:  168us 100ns
    Mean Duration: 465ms 704us 96ns
    Max Duration:  4s 966ms 630us

Send
  Count: 145
    Min Duration:  113us 124ns
    Mean Duration: 334ms 697us 667ns
    Max Duration:  4s 316ms 250us
```

```
Timeline
Time (s)    Event           Details

108.843     Coord Change    Triggering leader election
112.400     Reboot          Reboot abcd::2:0:1:1
119.664     Coord Change    Triggering leader election
120.703     Coord Change    Triggering leader election
133.857     Reboot          KillInstantly abcd::2:2:1:1
145.409     Coord Change    Triggering leader election
```



```
Overview
Seed:                291983427
Simulated Time:      6m 15s 355ms
Real Time:           29s 838ms 300us
```

```
Config Summary
Replication: triple
Storage Engine: memory
Commit Proxies: 3
Logs: 3
Proxies: 4
Resolvers: 1
```

```
Machine Distribution
DC 0: 8 machines
```

```
Process Distribution
DC        Machine ID     IP Address         Process ID   Class Type

0         48cff5318e5fa3 2.0.1.5 2.0.1.5 48cff5318e5f unset
0         8101fbc91b1918 2.0.1.0 2.0.1.0 8101fbc91b19 unset
0         8f9f547fe3961e 2.0.1.3 2.0.2.3 8f9f547fe396 unset
0         b7ff4abb93ce3a 2.0.1.1 2.0.2.1 b7ff4abb93ce unset
0         d1a4fdb41fca73 2.0.1.7 2.0.2.7 d1a4fdb41fca storage_cache
0         e220b7d6f0dd02 2.0.1.2 2.0.2.2 e220b7d6f0dd unset
0         f7a5cd10843e7a 2.0.1.4 2.0.1.4 f7a5cd10843e unset
0         f8caaca8539a35 2.0.1.6 2.0.2.6 f8caaca8539a unset
```

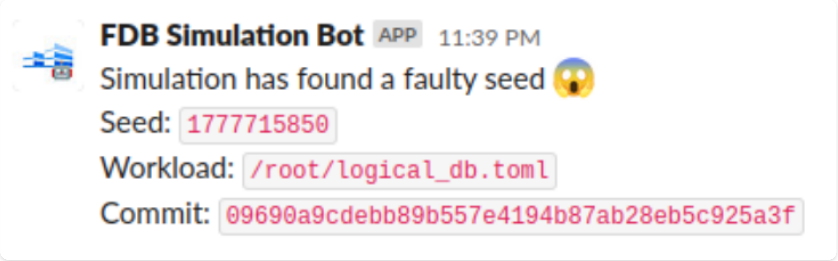
```
Network splits
Count: 376
  Min Duration:    755us 810ns
  Mean Duration:   1s 386ms 712us 879ns
  Max Duration:    9s 727ms 530us
```

```
Network latencies
All
  Count: 384
    Min Duration:   77us 671ns
    Mean Duration:  741ms 398us 267ns
    Max Duration:   8s 769ms 810us

Receive
  Count: 264
    Min Duration:   162us 860ns
    Mean Duration:  748ms 905us 310ns
    Max Duration:   8s 368ms 50us

Send
  Count: 229
    Min Duration:   130us 806ns
    Mean Duration:  705ms 939us 993ns
    Max Duration:   9s 337ms 10us
```

```
Timeline
Time (s)   Event           Details

61.553     Coord Change    Triggering leader election
68.792     Reboot          RebootAndDelete 2.0.3.7
68.792     Reboot          RebootAndDelete 2.0.2.7
68.792     Reboot          RebootAndDelete 2.0.1.7
68.792     Reboot          RebootAndDelete 2.0.2.7
68.792     Reboot          RebootAndDelete 2.0.3.7
77.748     Reboot          Reboot 2.0.3.6
77.748     Reboot          Reboot 2.0.2.6
77.748     Reboot          Reboot 2.0.1.6
77.748     Reboot          Reboot 2.0.4.6
77.748     Reboot          Reboot 2.0.2.6
100.468    Coord Change    Triggering leader election
```

Simulation runs continuously 🔄

- 💻 Engineers run a **few seeds locally** during development
- 🔁 CI runs **more iterations** on every push
- ☁️ Cloud runs simulation **continuously**
- ⏱️ **30 minutes** of simulation = **24 hours** of chaos testing
- 🎲 Faulty seed found? **Replay it locally**, deterministically



FDB Simulation Bot APP 11:39 PM

Simulation has found a faulty seed 😱

Seed: `1777715850`

Workload: `/root/logical_db.toml`

Commit: `09690a9cdebb89b557e4194b87ab28eb5c925a3f`

Claude Code + Simulation in action 🤖🔬

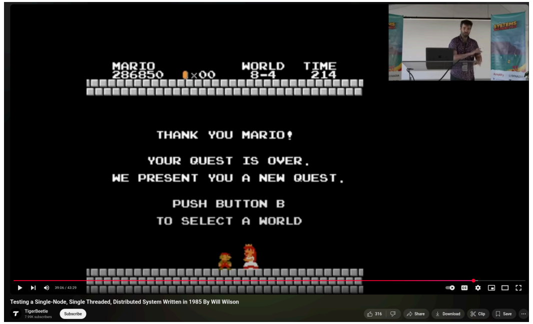
Claude Code autonomously implemented real components under simulation:

- 🦀 **foundationdb-rs** — binding tester with ~219 days of continuous exploration/month
- 🗳️ **Leader election** — generated **13 machine-checkable invariants**, verified under network partitions, process crashes, and clock skew
- 🏗️ **Materia** — implemented workflow engine, index types, query support — and found deep bugs through simulation along the way
- 🔍 **moonpool** — Developing a distributed system simulation framework

What is the state of autonomous testing? 🎮

Antithesis is the leader in autonomous testing — their platform uses **guided random exploration** to find bugs in any software.

Demo: they beat Super Mario Bros using only random inputs — no human, no scripting. Pure guided exploration. 🏆 See Testing a Single-Node, Single Threaded, Distributed System Written in 1985 by Will Wilson.



THANK YOU MARIO!

YOUR QUEST IS OVER.
WE PRESENT YOU A NEW QUEST.

PUSH BUTTON B
TO SELECT A WORLD

Testing a Single-Node, Single Threaded, Distributed System Written in 1985 By Will Wilson

Simulation is not a silver bullet ⚠️

- 📊 **Performance is invisible** — you still need a perf/bench farm
- 🧩 **Your model can be wrong** — if you don’t simulate it, you won’t find it
- 🏭 **Production fakes need chaos too** — your production implementations need their own fault injection
- 🎲 **Rare bugs need smart exploration** — brute force isn’t enough
- ⏰ **Bug-finding latency** — a bug can hide in a seed for months

Testing must evolve 🧬

Remember the HDFS incident? Network partition + disk full + restart = NullPointerException.

That exact combination? **It’s a seed in a simulation.** Found in seconds. Fixed before production. **No 3am wake-up call.** 😴

LLMs generate code faster than ever. DST catches the bugs they introduce. Together: **autonomous discovery** 🤖🧪

The feedback loop works for junior engineers and AI alike — it lets them discover what they don’t know.

The tools exist. The techniques are proven. Testing must evolve from prevention to discovery.

The spectrum of adoption 📈

Start anywhere. Each level adds value.

Level	What to do	What you get
1 🎲	Random workload generation	Test unusual combinations
2 🧪	Property-based testing	Flush out your system spec
3 🎭	Fakes	Fast, deterministic tests
4 😈	Fault-injectable fakes	Discover edge cases
5 ⚙️	Seed-driven DST	Reproducible bugs, autonomous discovery

Thank you! 🙏

Testing: Prevention vs Discovery

Pierre Zemb — @PierreZ · pierrezemb.fr

Questions? 💬