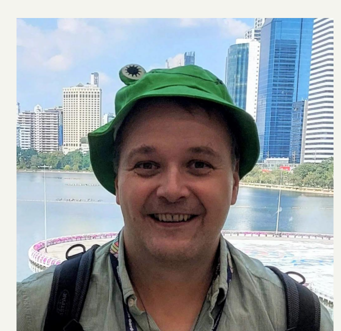
Find distributed systems bugs before production *with executable specifications*



Igor Konnov

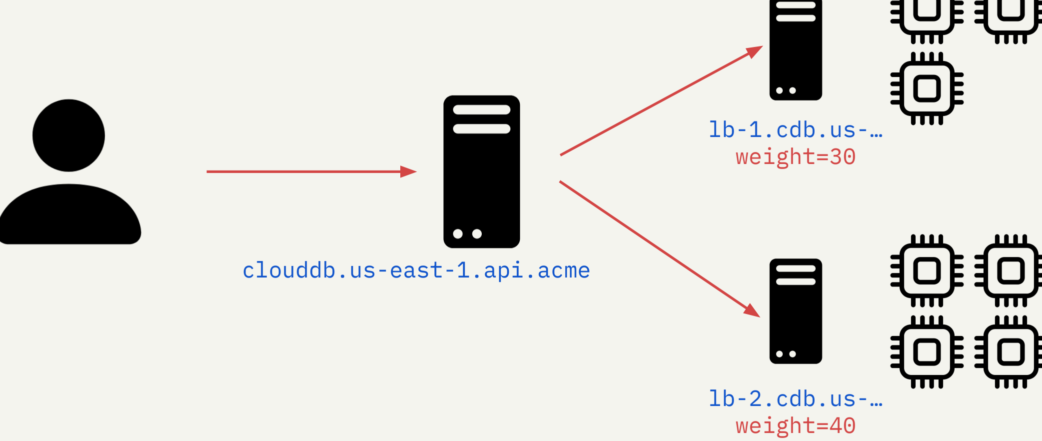
Thomas Pani

What we bring today

How to **find bugs before production** in distributed systems
(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical

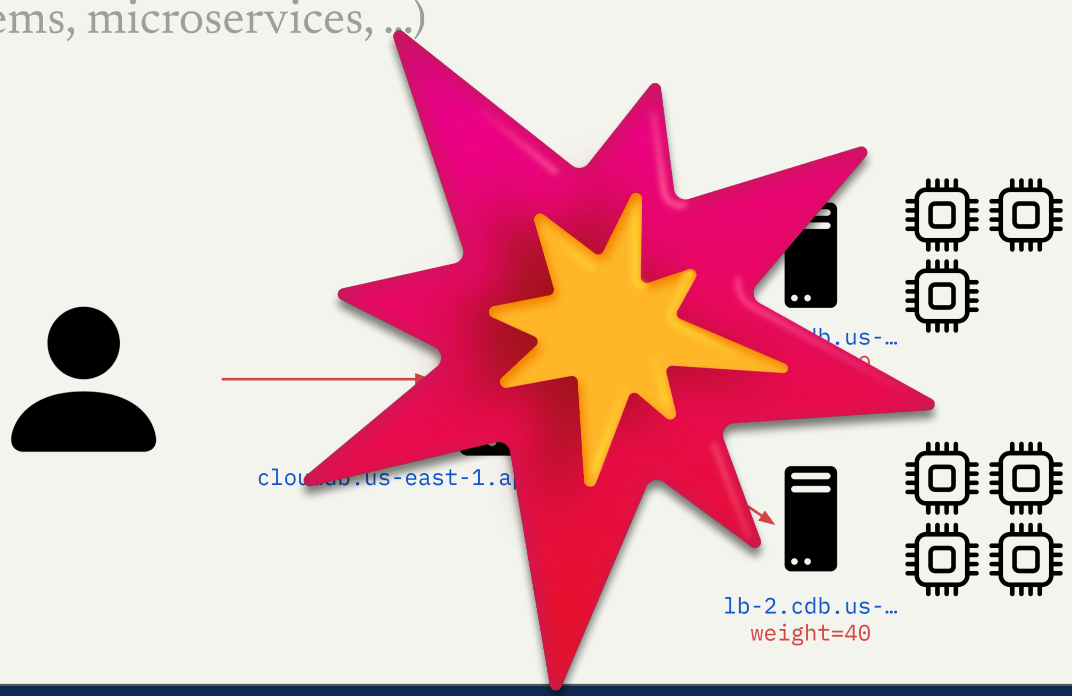


What we bring today

How to **find bugs before production** in distributed systems
(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical



About us

1) We help teams with **reliability of distributed systems**
    a) manual audit & review
    b) tool-based ← *today*
2) R&D on **practical (“lightweight”) formal verification**

Igor Konnov

konnov

konnov-phd

konnov.phd

protocols-made-fun.com

Thomas Pani

thpani

thpani

blltprf.xyz

About us

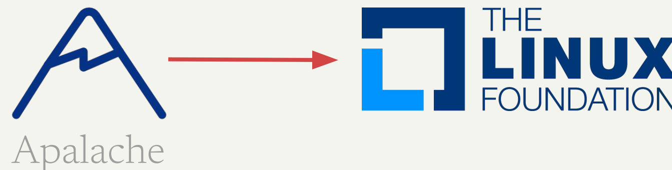
1) We help teams with **reliability of distributed systems**
   a) manual audit & review
   b) tool-based ← *today*
2) R&D on **practical ("lightweight") formal verification**

Tooling:

TLA$^+$ (specification language)

Apalache (model checker for TLA$^+$)

Quint (improved DevEx for TLA$^+$)



About us

1) We help teams with **reliability of distributed systems**
   a) manual audit & review
   b) tool-based ← *today*
2) R&D on **practical ("lightweight") formal verification**

Tooling:

TLA⁺ (specification language)

Apalache (model checker for TLA⁺)

Quint (improved DevEx for TLA⁺)

models in Alloy, proofs in Lean 4, ...

About us

1) We help teams with **reliability of distributed systems**
   a) manual audit & review
   b) tool-based ← *today*
2) R&D on **practical (“lightweight”) formal verification**

Tooling:

TLA⁺ (specification language)

Apalache (model checker for TLA⁺)

Quint (improved DevEx for TLA⁺)

models in Alloy, proofs in Lean 4, …

← *today*

A faulty distributed system

duration:
~14.5 hrs

cause:
latent race condition

affected:
Amazon retail, Prime Video,
Roblox, Fortnite, Pinterest, Snapchat,
Duolingo, Canva, Strava, Flickr, Peloton,
Zoom, Slack, Reddit, Signal, Lyft, Venmo,
Lloyds Bank, Halifax, HMRC, Bank of Scotland, ...

financial loss (est.):
$500M–$650M (US)

A faulty distributed system

duration:
~14.5 hrs

cause:
latent race condition

affected:
Amazon retail, Prime Video,
Roblox, Fortnite, Pinterest, Snapchat,
Duolingo, Canva, Strava, Flickr, Peloton,
Zoom, Slack, Reddit, Signal, Lyft, Venmo,
Lloyds Bank, Halifax, HMRC, Bank of Scotland, ...

financial loss (est.):
$500M–$650M (US)

Our faulty distributed system: ACME CloudDB

Not 1:1 copy, but the **same root cause**

Our faulty distributed system: ACME CloudDB

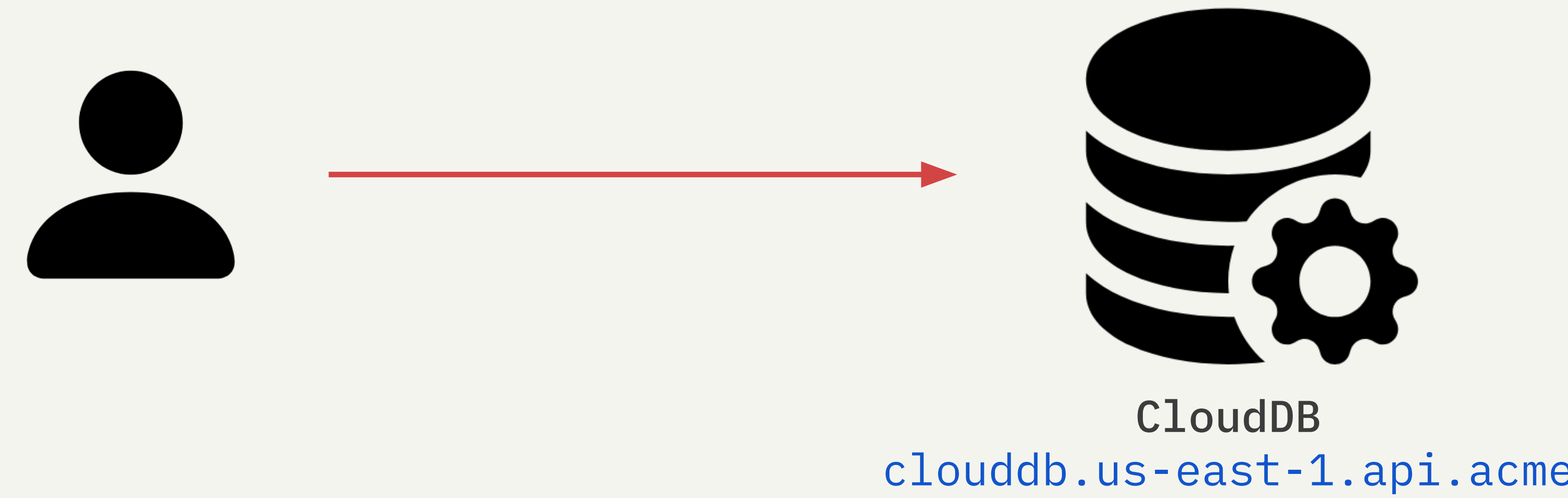
Not 1:1 copy, but the **same root cause**

Workshop Goal: *"shift left"* = find the bug before

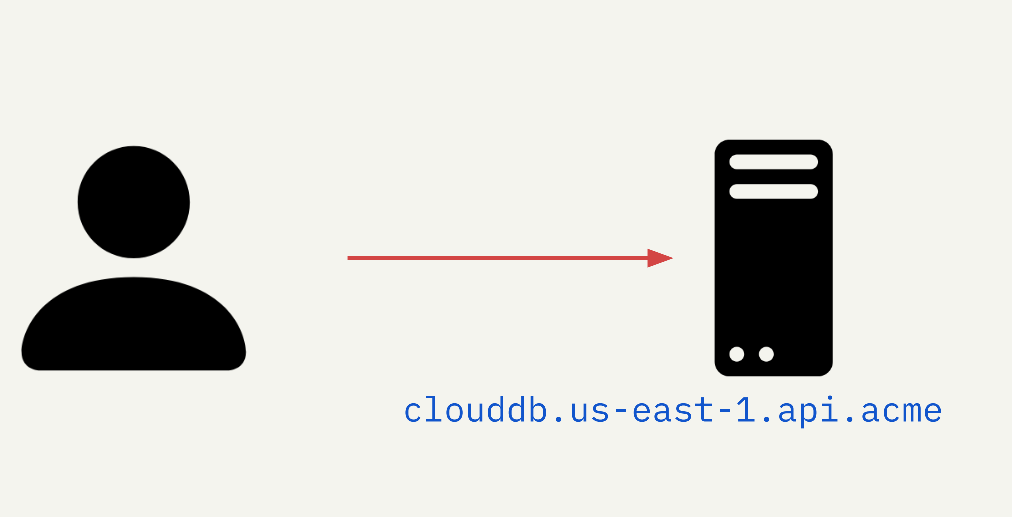
- it reaches production
- causes an outage
- costs $$$

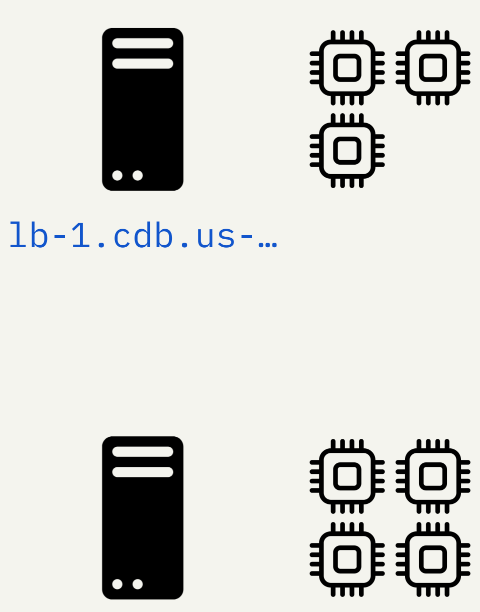
Discuss how to test this *kind of system*.

Our faulty distributed system: ACME CloudDB

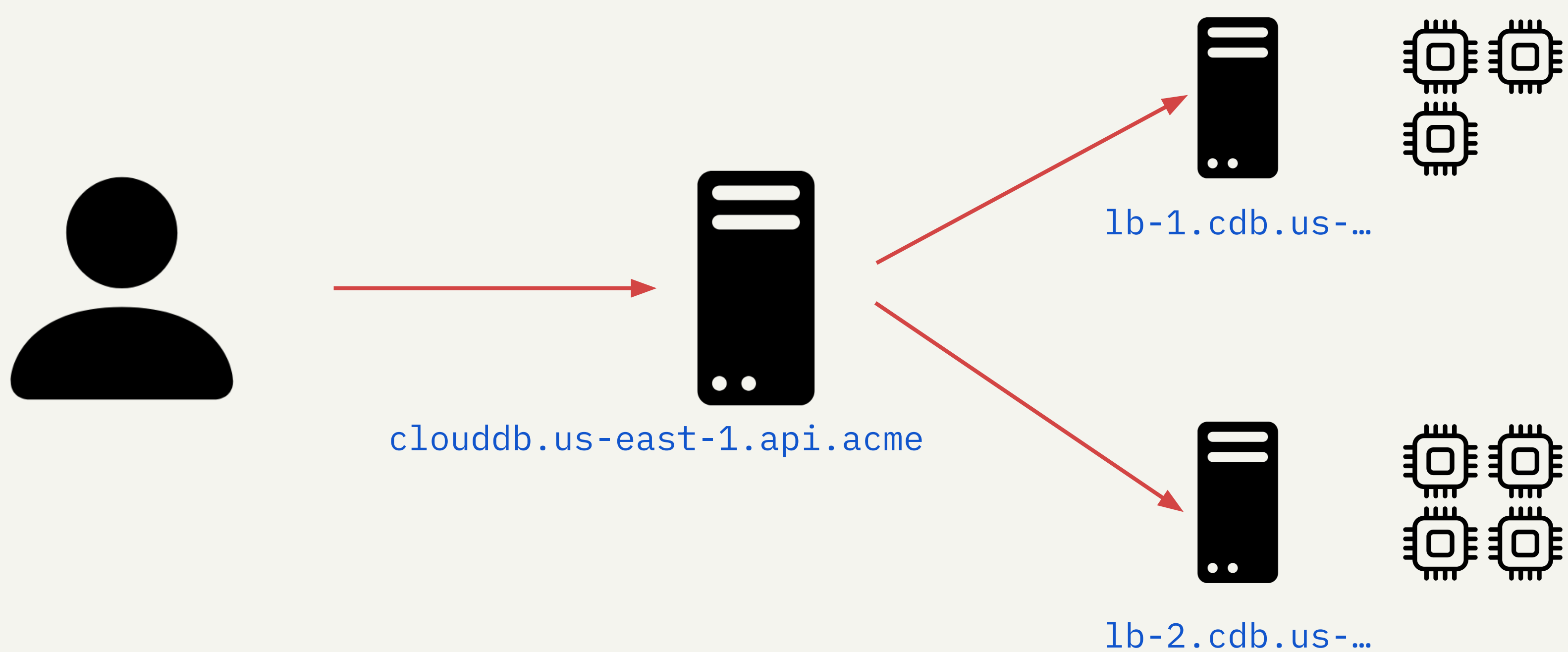


Our faulty distributed system: ACME CloudDB

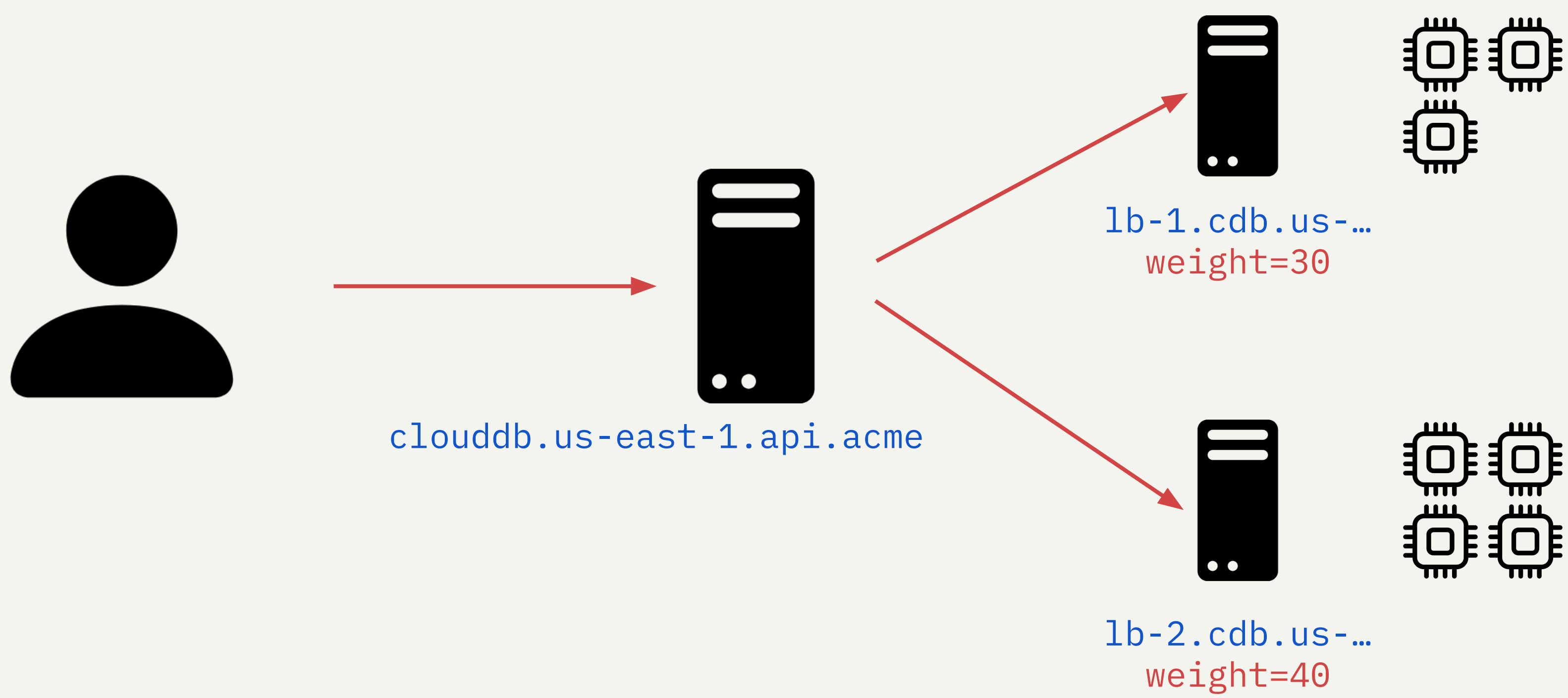




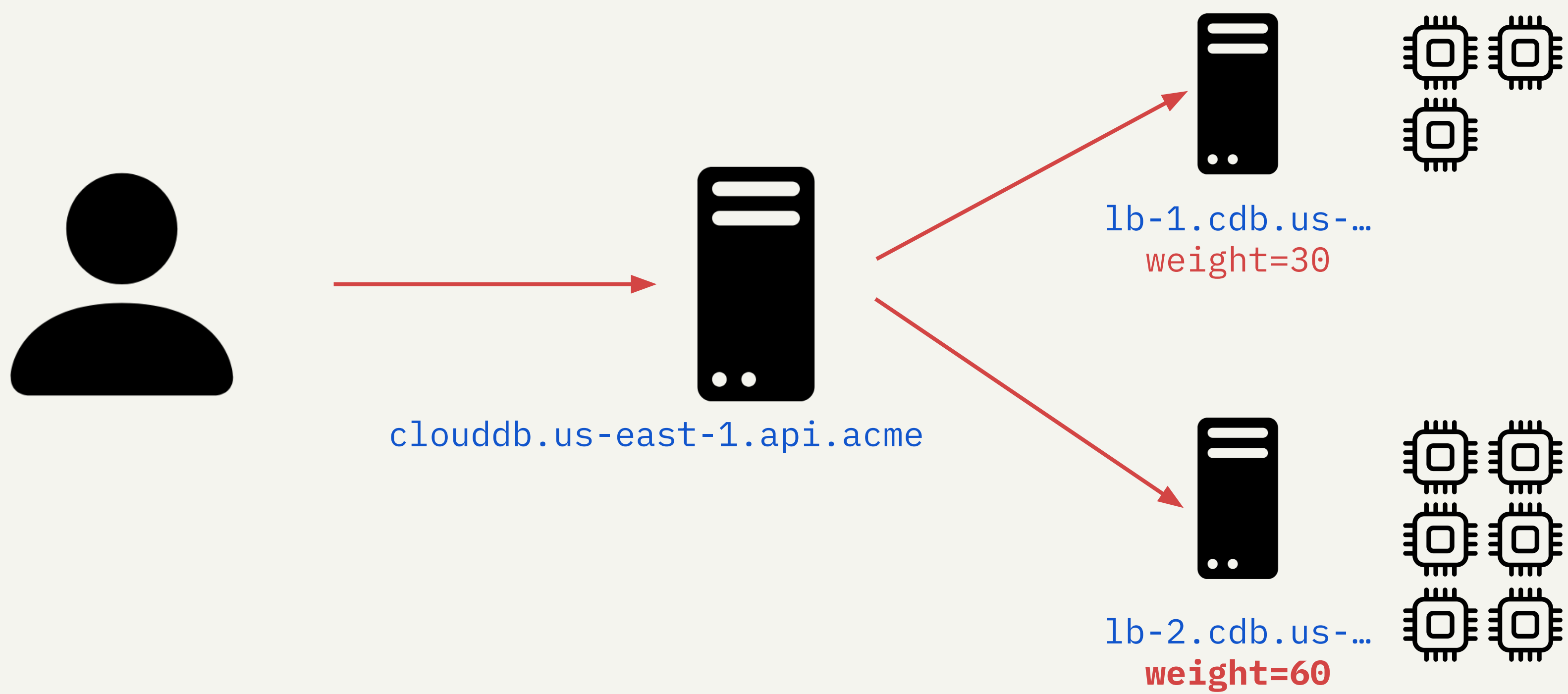
Our faulty distributed system: ACME CloudDB



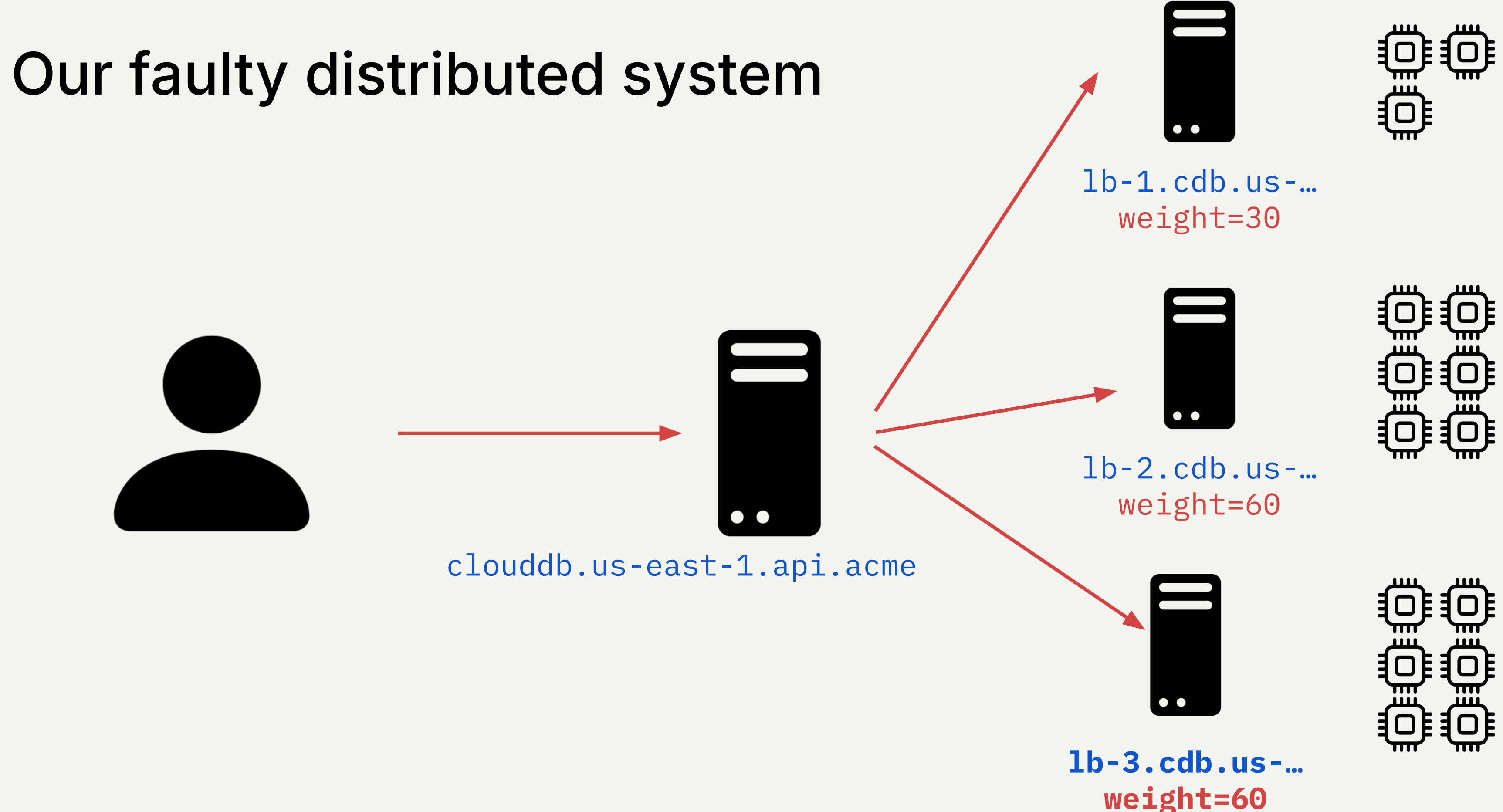
Our faulty distributed system: ACME CloudDB



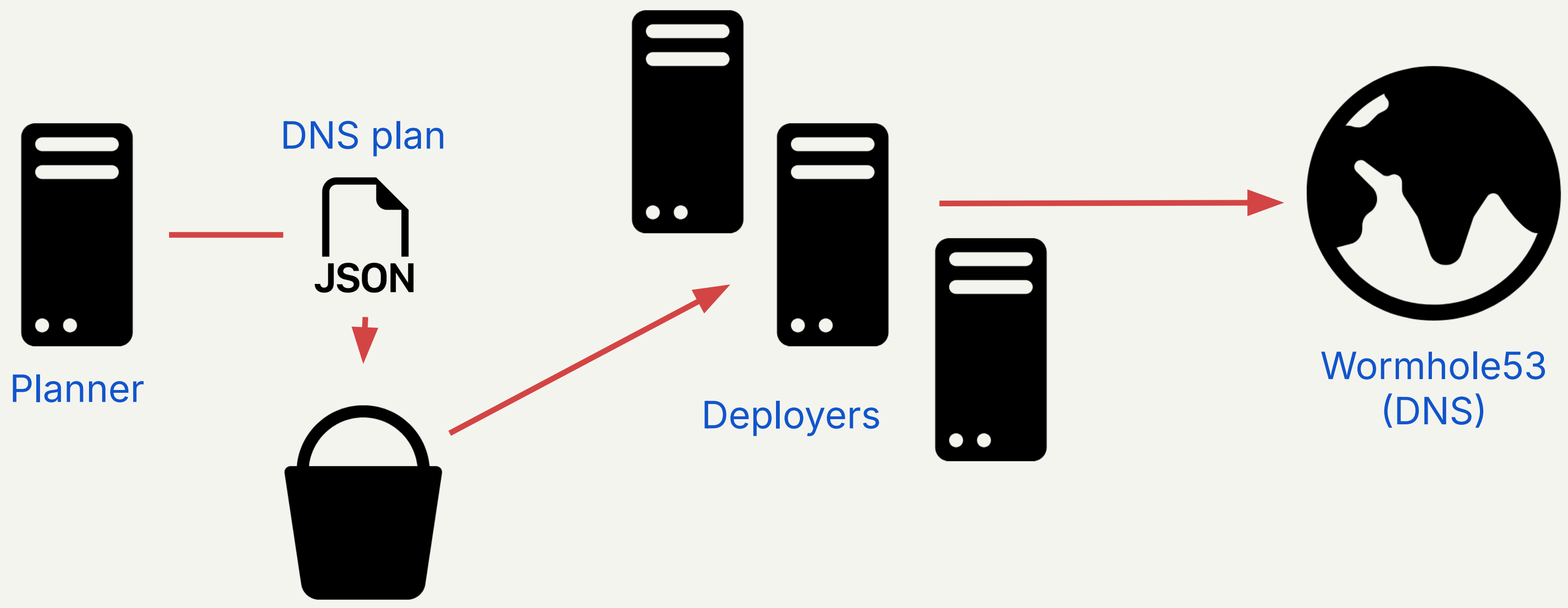
Our faulty distributed system: ACME CloudDB



Our faulty distributed system



Our faulty distributed system: ACME DNS control plane



Planner: Generate plans (JSON)

`clouddb.us-east-1.api.acme`

`plan-001.cdb.api.acme`

`lb-1.cdb.us-…`	`lb-2.cdb.us-…`
`10.0.0.1`	`10.0.0.2`
`weight=30`	`weight=40`

Deployers: Deploy DNS plans

`clouddb.us-east-1.api.acme`

`plan-001.cdb.api.acme`

`lb-1.cdb.us-…`
`10.0.0.1`
`weight=30`

`lb-2.cdb.us-…`
`10.0.0.2`
`weight=40`

Deployers: Deploy DNS plans

Wormhole53
(DNS)

```
clouddb.us-east-1.api.acme

plan-001.cdb.api.acme

lb-1.cdb.us-…      lb-2.cdb.us-…
  10.0.0.1           10.0.0.2
 weight=30          weight=40

plan-002.cdb.api.acme

lb-1.cdb.us-…      lb-2.cdb.us-…      lb-3.cdb.us-…
  10.0.0.1           10.0.0.2           10.0.0.3
 weight=30          weight=60          weight=60
```

Deployers: Deploy DNS plans

Wormhole53 (DNS)

clouddb.us-east-1.api.acme

plan-001.cdb.api.acme

plan-002.cdb.api.acme

lb-1.cdb.us-…
10.0.0.1
weight=30

lb-2.cdb.us-…
10.0.0.2
weight=40

lb-1.cdb.us-…
10.0.0.1
weight=30

lb-2.cdb.us-…
10.0.0.2
weight=60

lb-3.cdb.us-…
10.0.0.3
weight=60

Deployers: Garbage-collect old DNS plans

clouddb.us-east-1.api.acme

plan-001.cdb.api.acme

plan-002.cdb.api.acme

lb-1.cdb.us-…
10.0.0.1
weight=30

lb-2.cdb.us-…
10.0.0.2
weight=40

lb-1.cdb.us-…
10.0.0.1
weight=30

lb-2.cdb.us-…
10.0.0.2
weight=60

lb-3.cdb.us-…
10.0.0.3
weight=60

Deployers: Garbage-collect old DNS plans

clouddb.us-east-1.api.acme

plan-002.cdb.api.acme

lb-1.cdb.us-…
10.0.0.1
weight=30

lb-2.cdb.us-…
10.0.0.2
weight=60

lb-3.cdb.us-…
10.0.0.3
weight=60

Hands-on Spec: ACME Cloud DNS control plane

Goal: quick demo of the SUT

github.com/thpani/testing-distsys-devconf26

How do we test this?

Distributed Systems fail outside test coverage

Production bugs

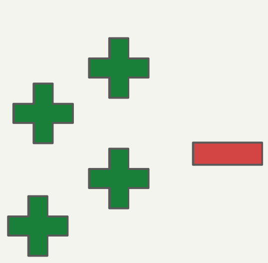
- are not on the tested paths (duh)
- appear as *multiple, unforeseen* messages / events / timeouts / retries / ...

Bad "fix": think harder. write more tests.

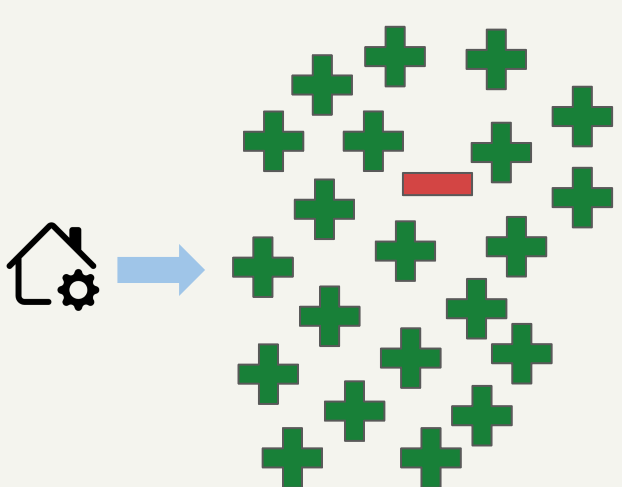
Good fix: build a machine that produces the paths we are not thinking about.

Testing things

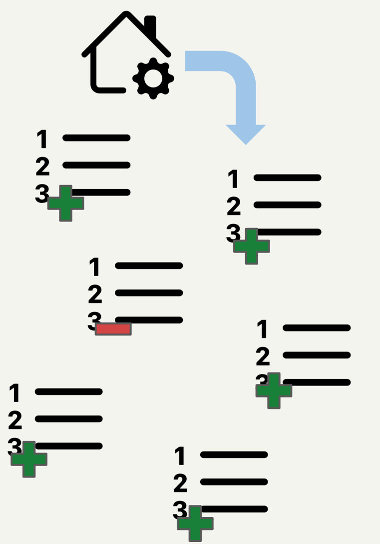
A. example-based
table-driven testing



B. input generation
property-based (PBT) / fuzzing



C. behavioral testing
[next slide]



Testing things

A. example-based
table-driven testing

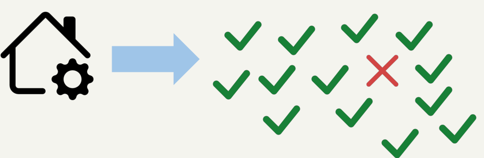
```
a.deposit(100)
a.withdraw(30)
a.balance == 70 ✓

a.deposit(50)
a.withdraw(0)
a.balance == 50 ✓

a.deposit(50)
a.deposit(50)
a.balance == 100 ✗
```

B. input generation
property-based (PBT) / fuzzing

```
for all init >= 0, amt > 0:
  a = Account(init)
  a.deposit(amt)
  a.withdraw(amt)
  a.balance == init
```



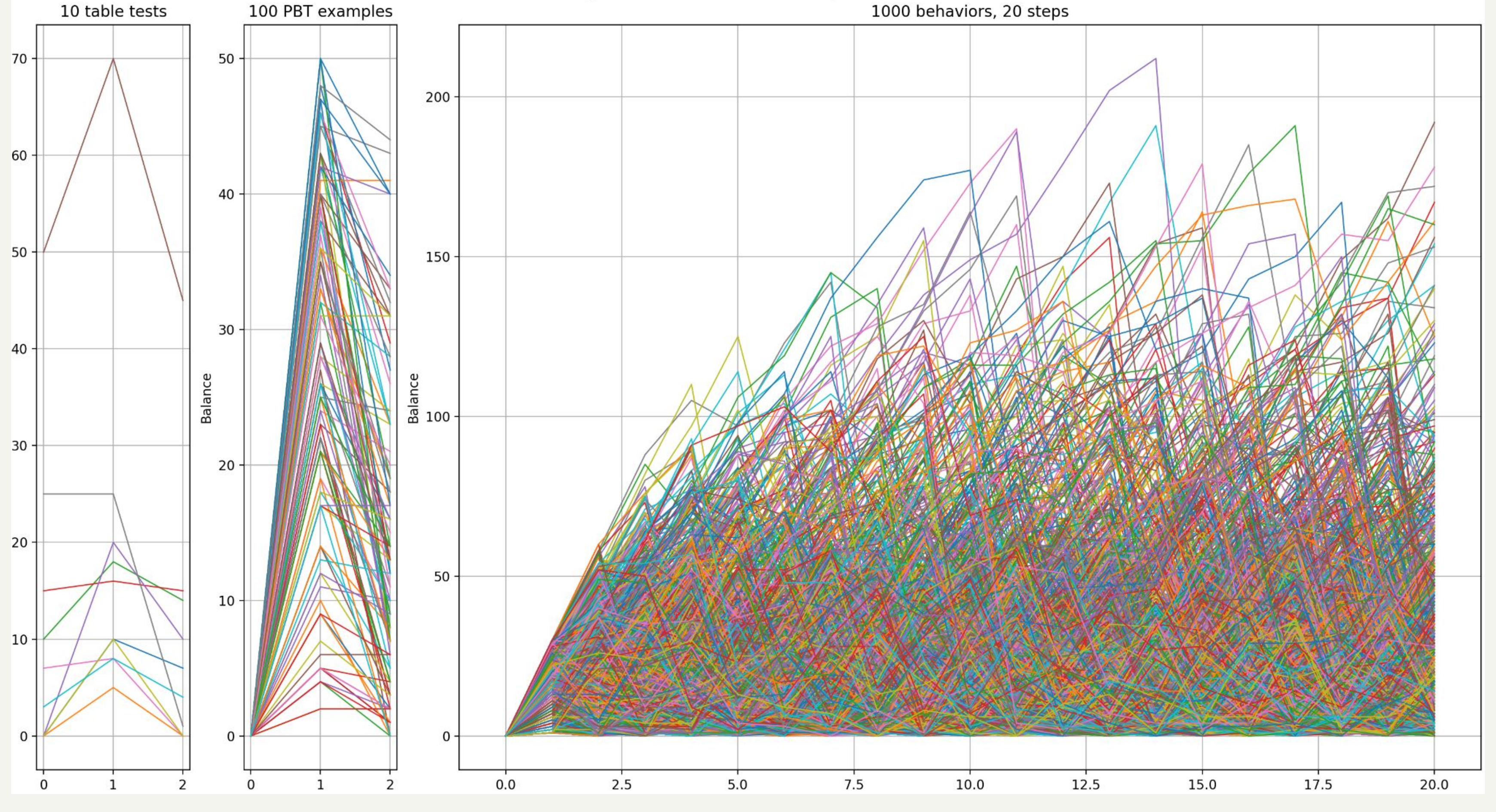
C. behavioral testing
[next slide]

```
init:
  a = Account(0)
  state = 0

deposit(amt):
  a.deposit(amt)
  state += amount

withdraw(amt):
  a.withdraw(amt)
  if amount <= state:
    state -= amount

always:
  a.balance == state
```


10 table tests
100 PBT examples
1000 behaviors, 20 steps
Balance
Balance

Testing systems: *Behavioral testing*

production system

Fault injection

Chaos testing

Deterministic simulation testing

Testing systems: *Behavioral testing*

production system

Fault injection

Chaos testing

Deterministic simulation testing

Testing systems: *Behavioral testing*

design-time

Executable specifications

shift left

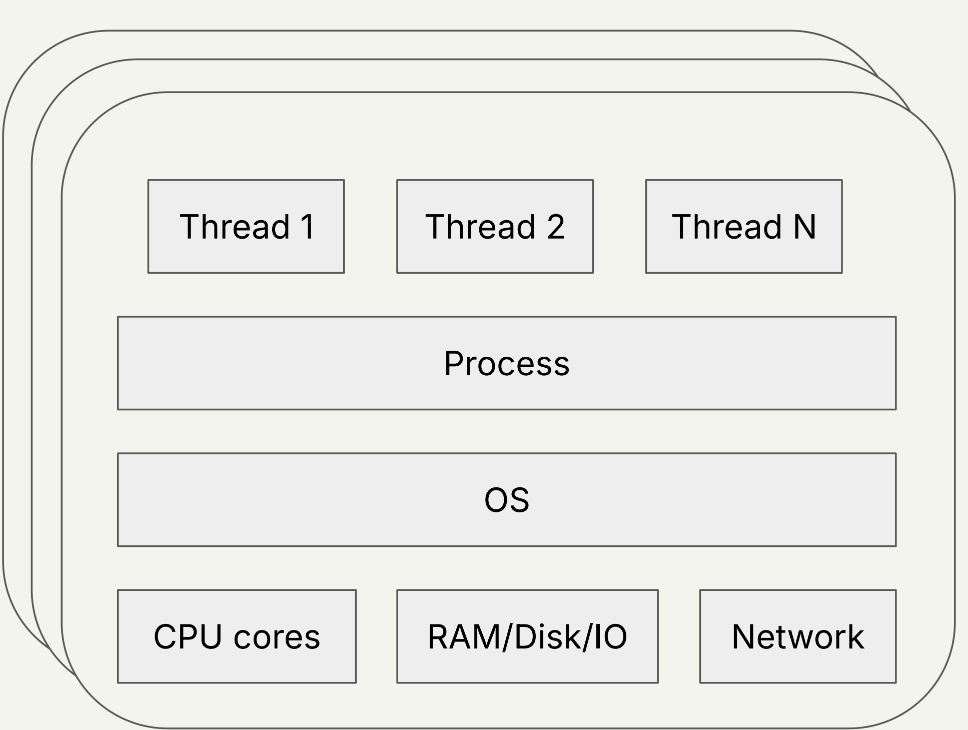
production system

Fault injection

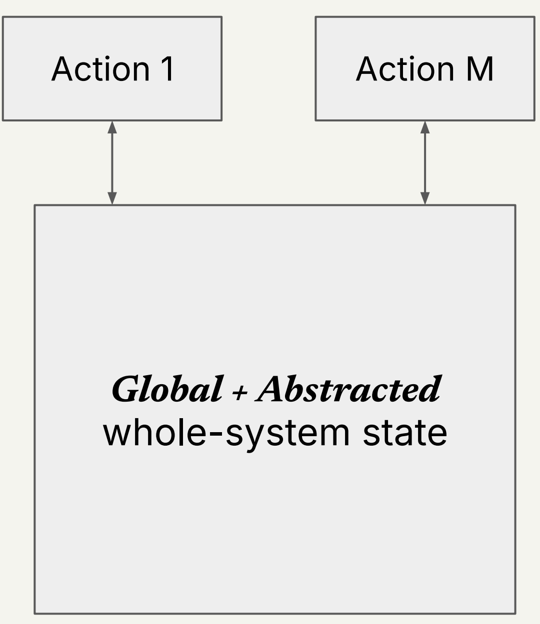
Chaos testing

Deterministic simulation testing

Why spec?



“Less physical”



What are executable specs?

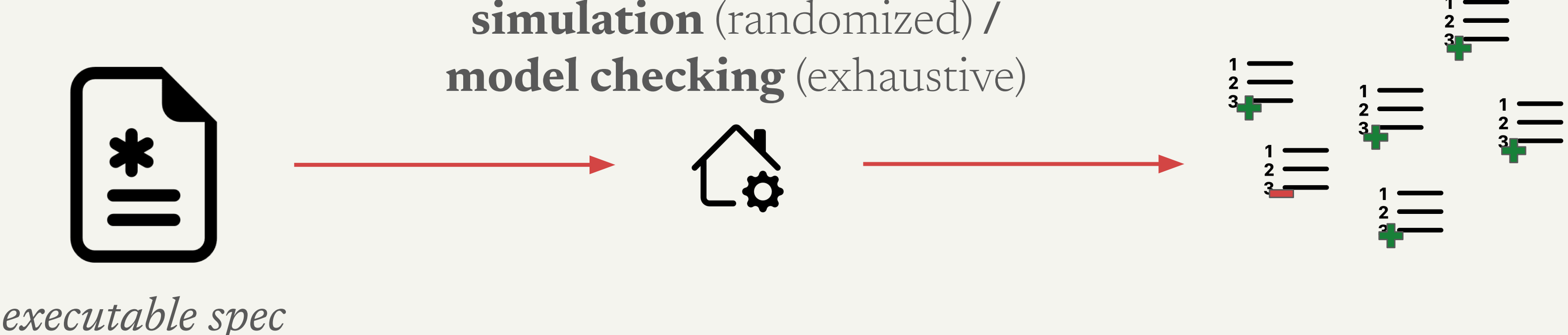
- Any high-level code that lets you focus on the hard problems
- De-focus from component boundaries & implementation details

  (PL idioms, memory, libraries, physical deployment)
- It could be just “high-level” Python (e.g., Ethereum EELS)

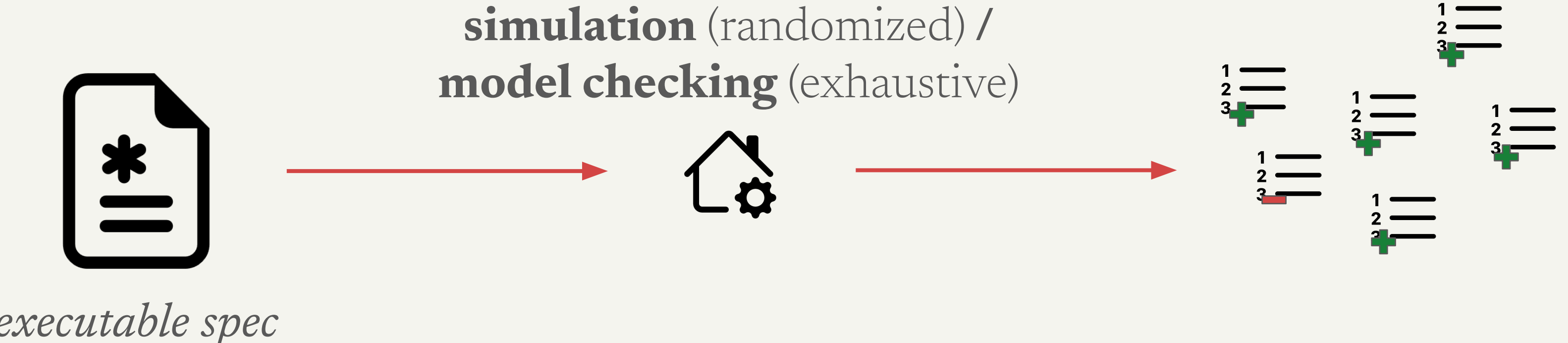
```
def reconcile(state):
    state.active_pod_ids -= set(p.id for p in state.pods_marked_for_deletion)
```

- **Obvious goodies**: replay steps, run simulations
- **Push it further**: Python DSL → fast simulations, tests, model checking, proofs

Executable specifications: *Search*



Executable specifications: *Search*



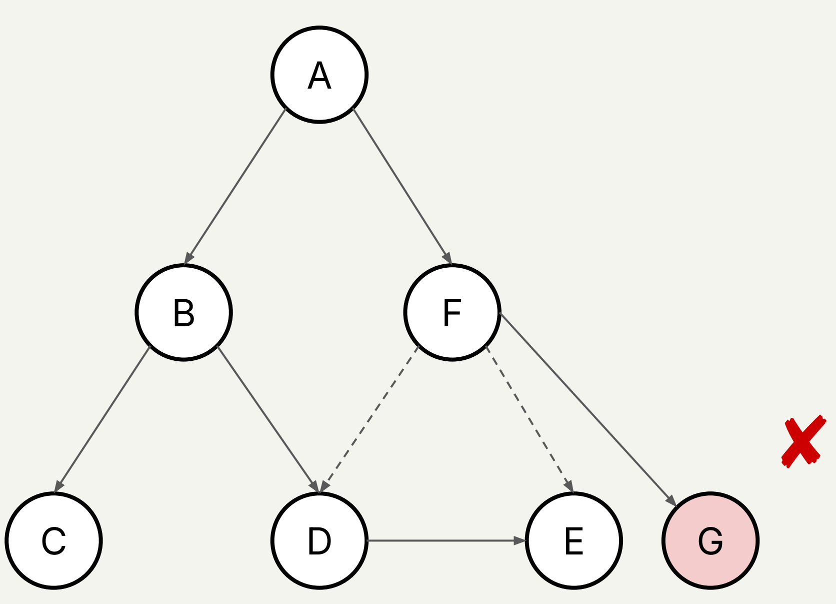
Incredibly hard to do at the implementation-level!

Hands-on Spec: ACME Cloud DNS control plane

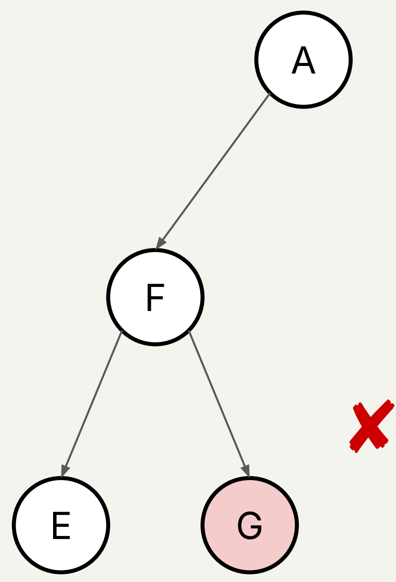
Goal: find the bug *at the spec-level* (→ later: system-level)

github.com/thpani/testing-distsys-devconf26

Depth-first search

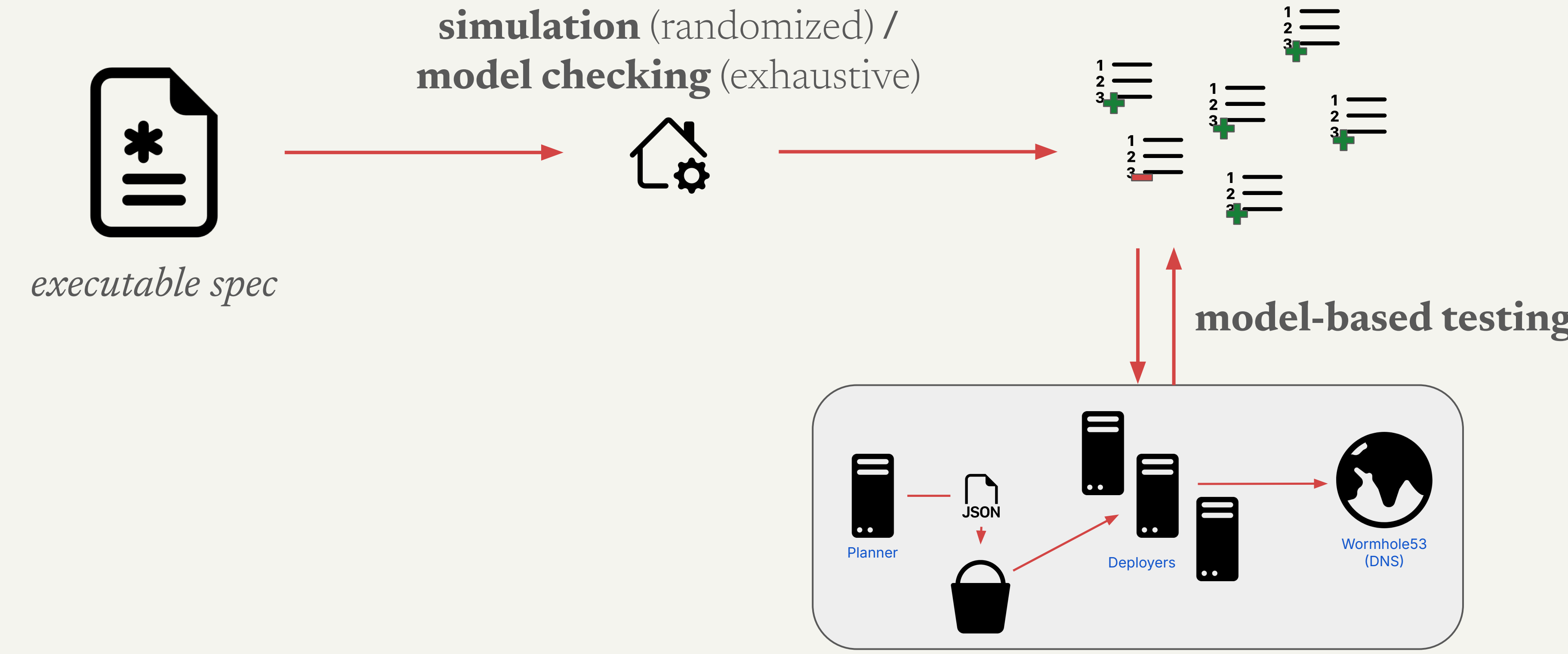


DFS with shuffle



Executable specifications: *Transfer to SUT*

Executable specifications: *Transfer to SUT*



Hands-on Spec: ACME Cloud DNS control plane

Goal: reproduce the bug *in the implementation*

github.com/thpani/testing-distsys-devconf26

Beyond one trace

Player 1 (Wunderspec)

Player 2 (SUT)

1. Find a trace to the bug
2. Execute the whole trace → ✓ / ✗

Multi-shot / Interactive game:

1. Choose next transition with inputs
2. Execute the transition. Respond.
3. Evaluate the response. Goto 1 ↑

Beyond one trace: advanced techniques

Interactive Symbolic Testing of TFTP with TLA+ and Apalache

Extracting formal specifications from Apache ZooKeeper with AI tools and Apalache

at protocols-made-fun.com

Let’s wrap this up

“Specification”-driven development

Recent trend:

- write technical specifications in Markdown
- ask LLMs to generate the code and tests → manual review

Our take:

- write functional parts as executable specs
- ask LLMs to generate the code → test it against the executable spec

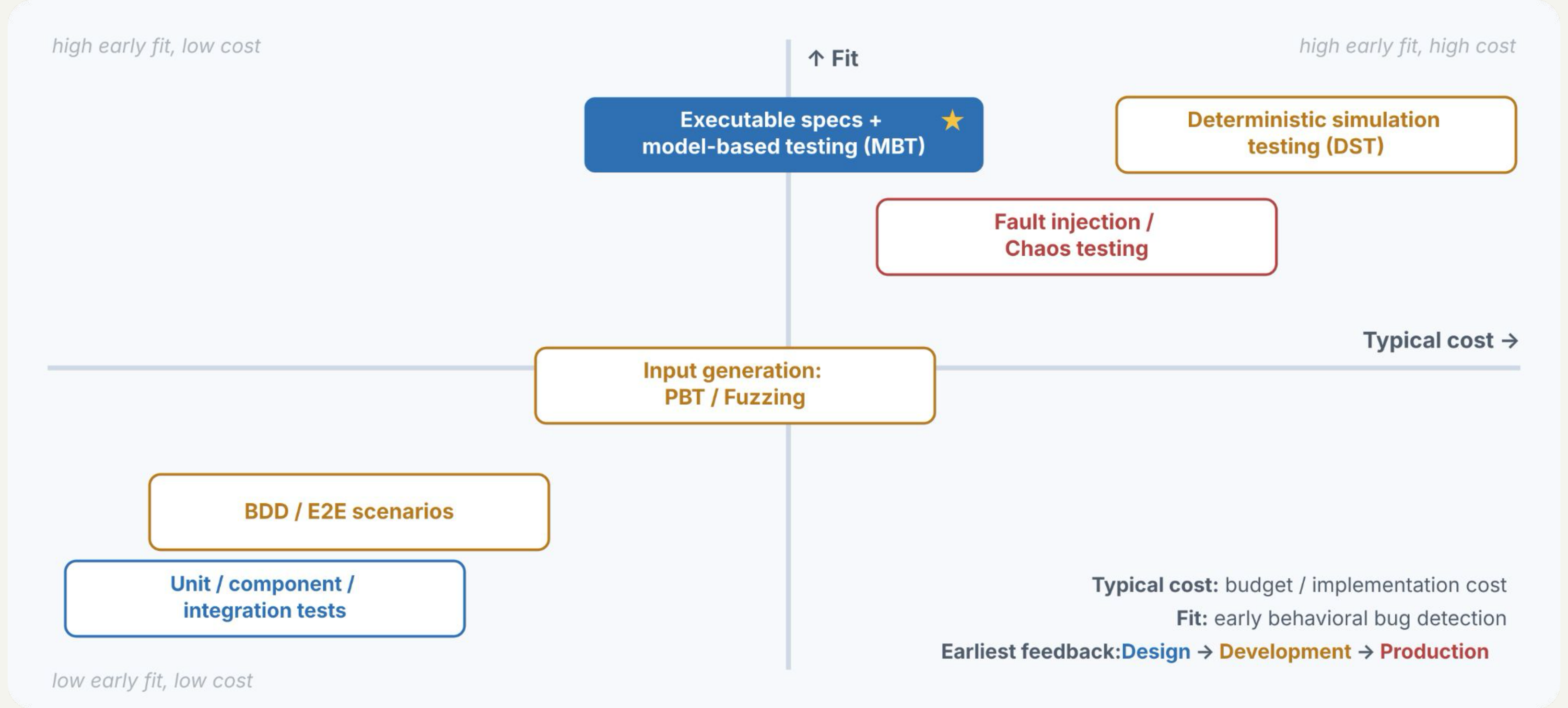
Use both!

When is this useful? If your system has these symptoms:

1. manages complex state
   - implements a state machine
   - contains a control loop
2. is distributed
   - composes multiple components
   - components can crash
   - incidents are hard to reproduce
   - bugs depend on time / ordering
3. has complex failure modes
   - incidents are hard to reproduce
   - has important system invariants
   - production bugs cost $$$$

… but no executable reference

Testing distributed systems



Executable specifications help reveal *behaviors* and *generate tests* that your current test suite misses.

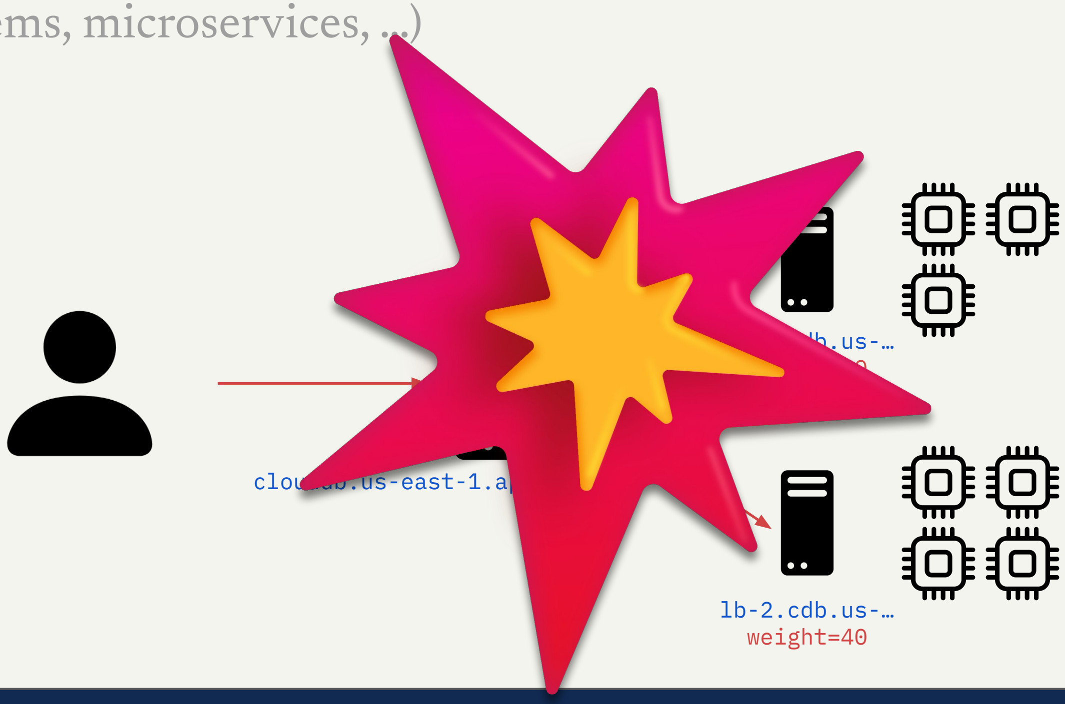
Before outages cost $$$ in production.

What we did today

How to **find bugs before production** in distributed systems
(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical

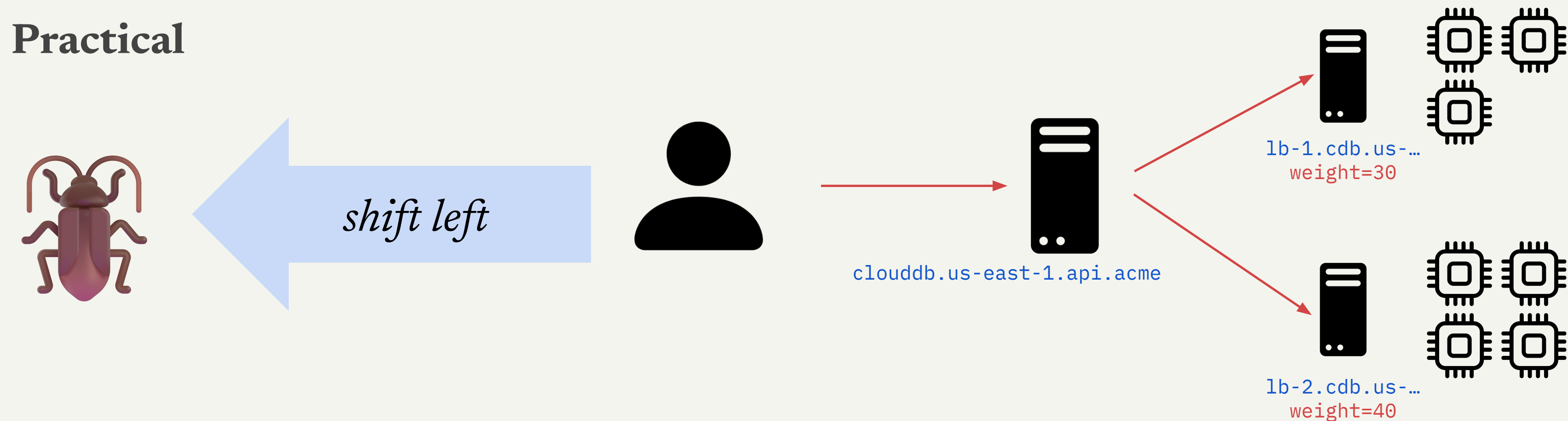


What we did today

How to **find bugs before production** in distributed systems
(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical

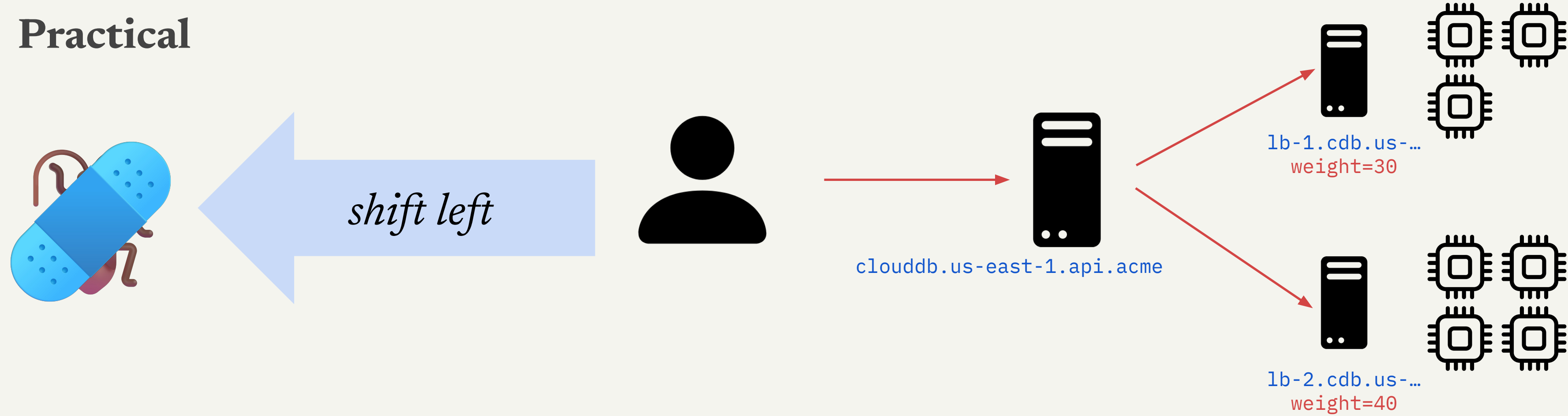


What we did today

How to **find bugs before production** in distributed systems
(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

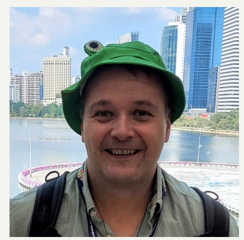
Practical



Find out if executable models fit *your system*

Get the workshop materials, testing landscape, pilot project details, and a 5-bullet system assessment.

Pilots: 1-2 focused projects after DevConf
Best fit: control planes, schedulers, databases, protocols, stateful cloud infra



Igor Konnov

Thomas Pani

wunderspec/wunderspec

demo@wunderspec.com

→ wunderspec.com/devconf