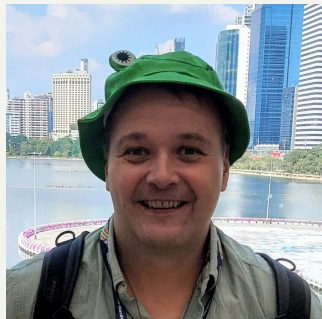


Find distributed systems bugs before production *with executable specifications*



Igor Konnov



Thomas Pani



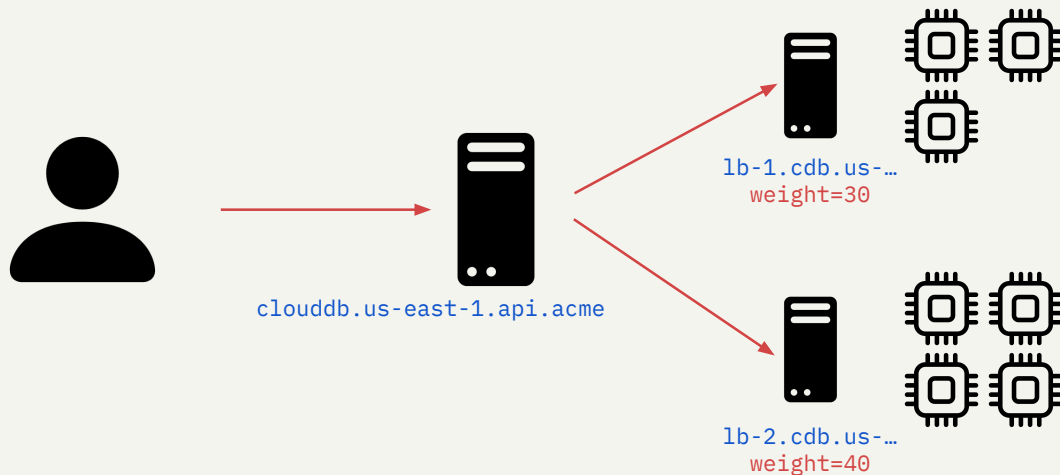
What we bring today

How to **find bugs in distributed systems** *before* production

(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical, interactive

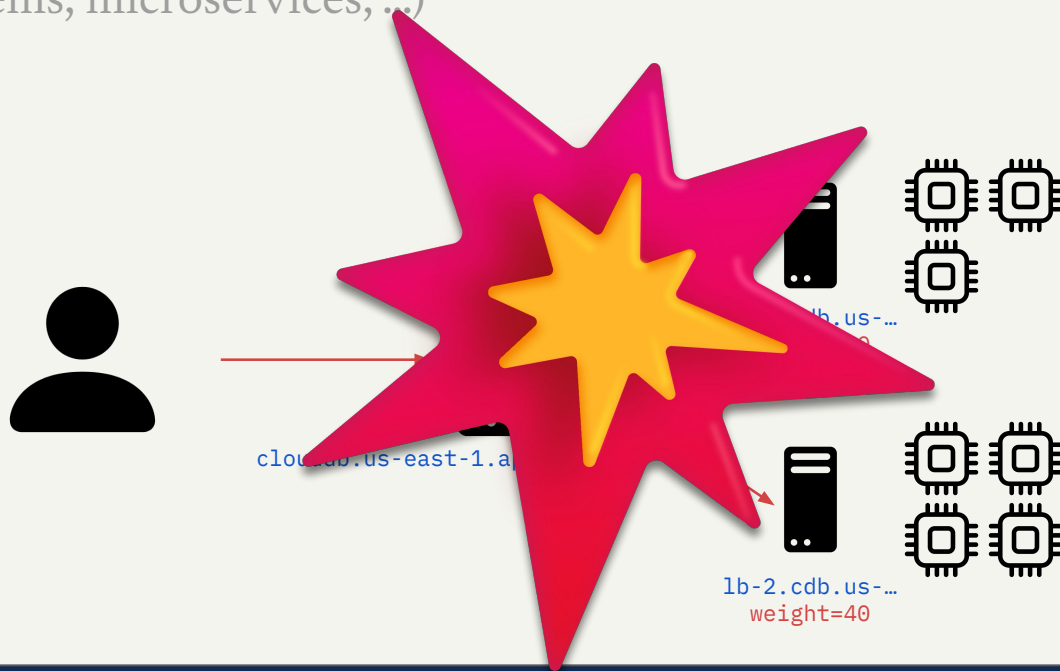


What we bring today

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

Focus: ideas, not tools

Practical, interactive



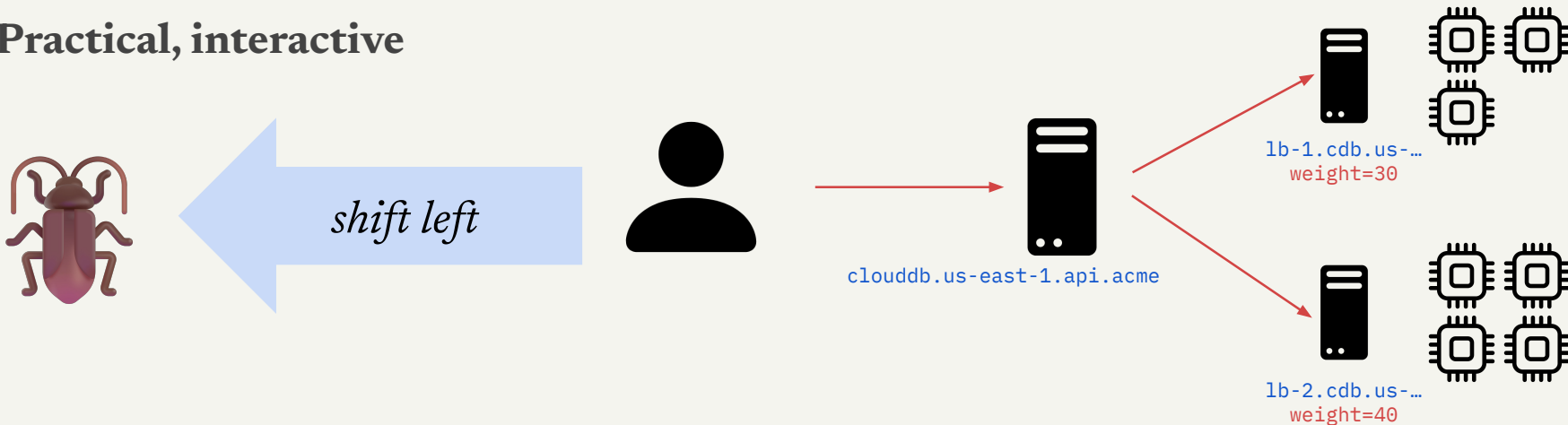
What we bring today

How to **find bugs in distributed systems** *before* production

(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical, interactive



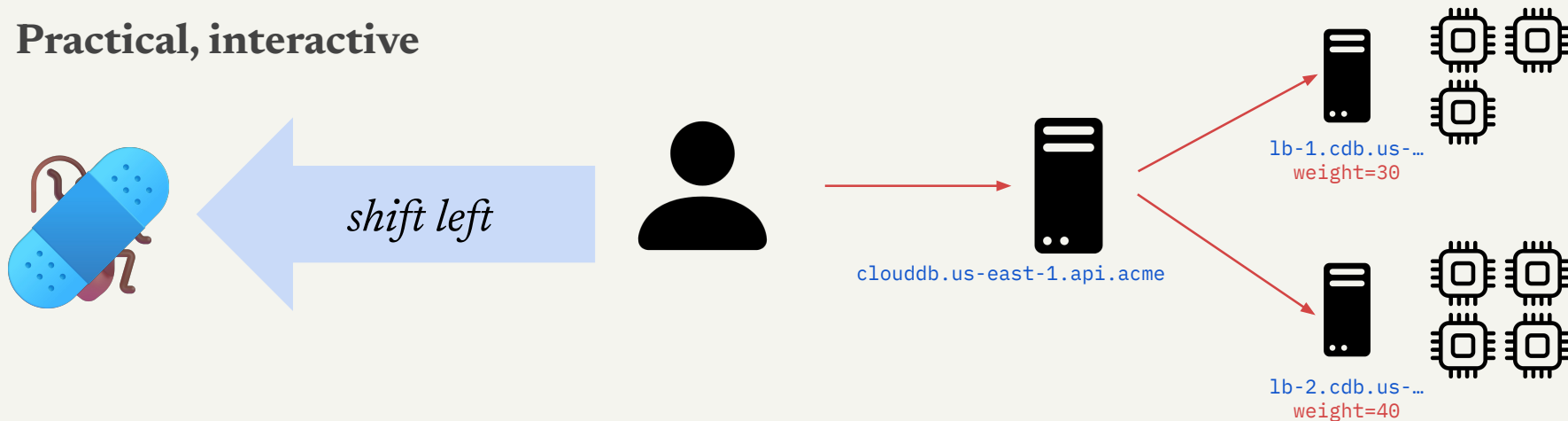
What we bring today

How to **find bugs in distributed systems** *before* production

(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical, interactive

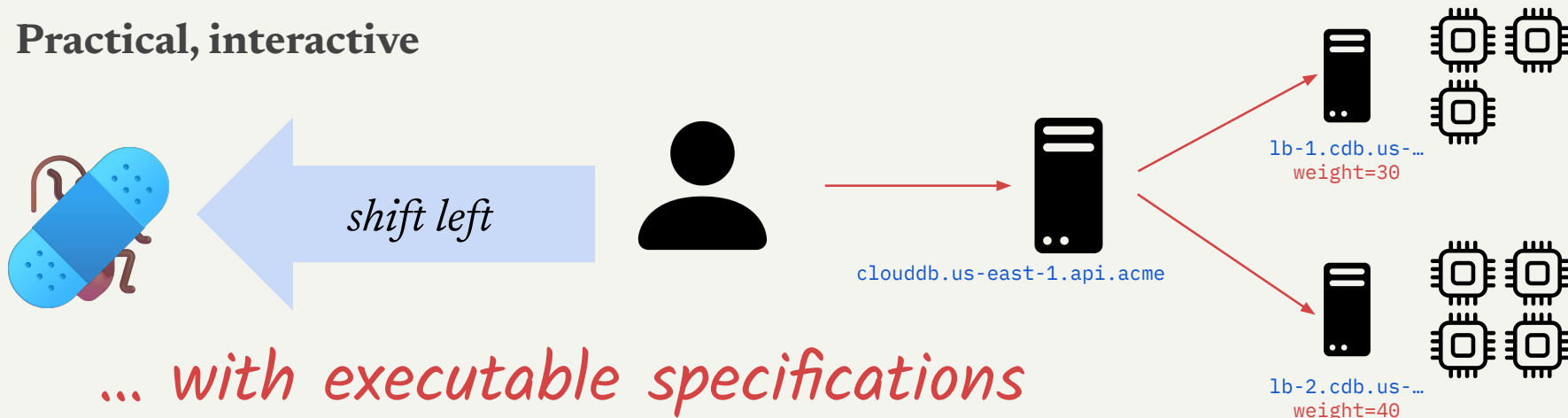


What we bring today

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

Focus: ideas, not tools

Practical, interactive



What are executable specs?

Any high-level code that lets you focus on the hard problems

Could just be “high-level” Python: (we will show you something better)

```
def reconcile(desired, observed):  
    unmanaged = {k: v for k, v in observed.config.items()  
                  if k not in desired.managed_keys}  
  
    return {**observed, "config": unmanaged | desired.config}
```

Use it as: test case generator, oracle, input to LLMs, documentation,, ...

"Specification"-driven development

Recent trend:

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

Our take:

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

LLMs and executable specs reinforce each other! → feedback loop

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



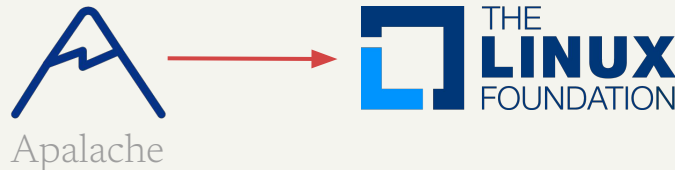
bltprf.xyz

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

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

Apalache (model checker for TLA⁺)

Quint (improved DevEx for TLA⁺)



← *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)



DATA 53

DynamoDB: Resilience & lessons from the Oct 2025 service disruption

Craig Howard
Senior Principal Engineer
Amazon DynamoDB



Our faulty distributed system: ACME CloudDB

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

Our faulty distributed system: ACME CloudDB

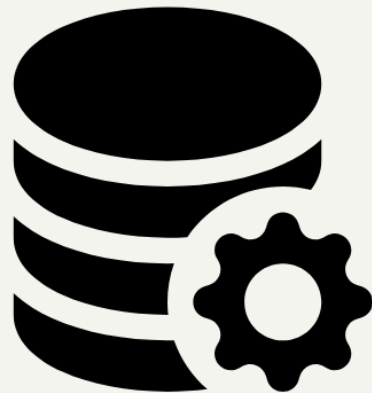
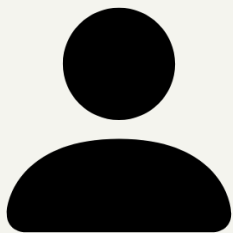
Not 1:1 copy, but the **same root cause**

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

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

Discuss how to test this *kind of system*.

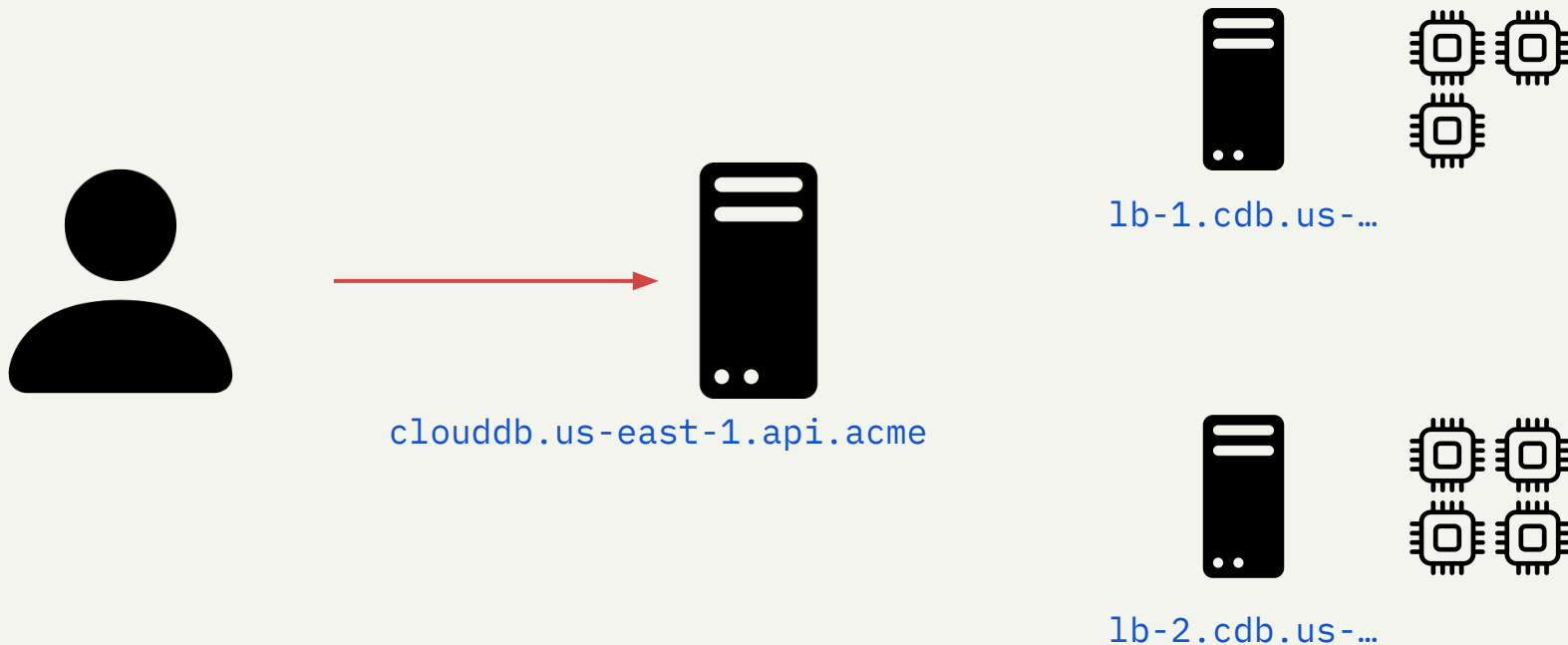
Our faulty distributed system: ACME CloudDB



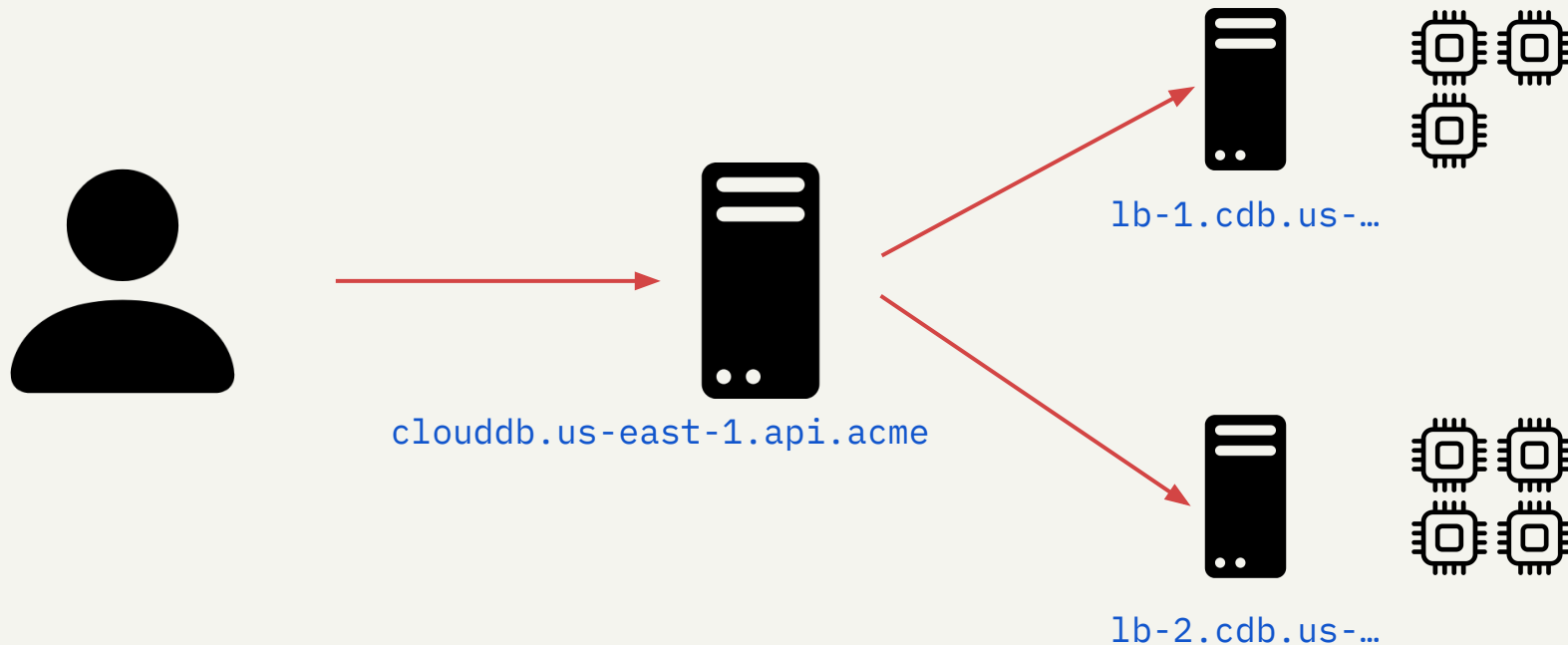
CloudDB

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

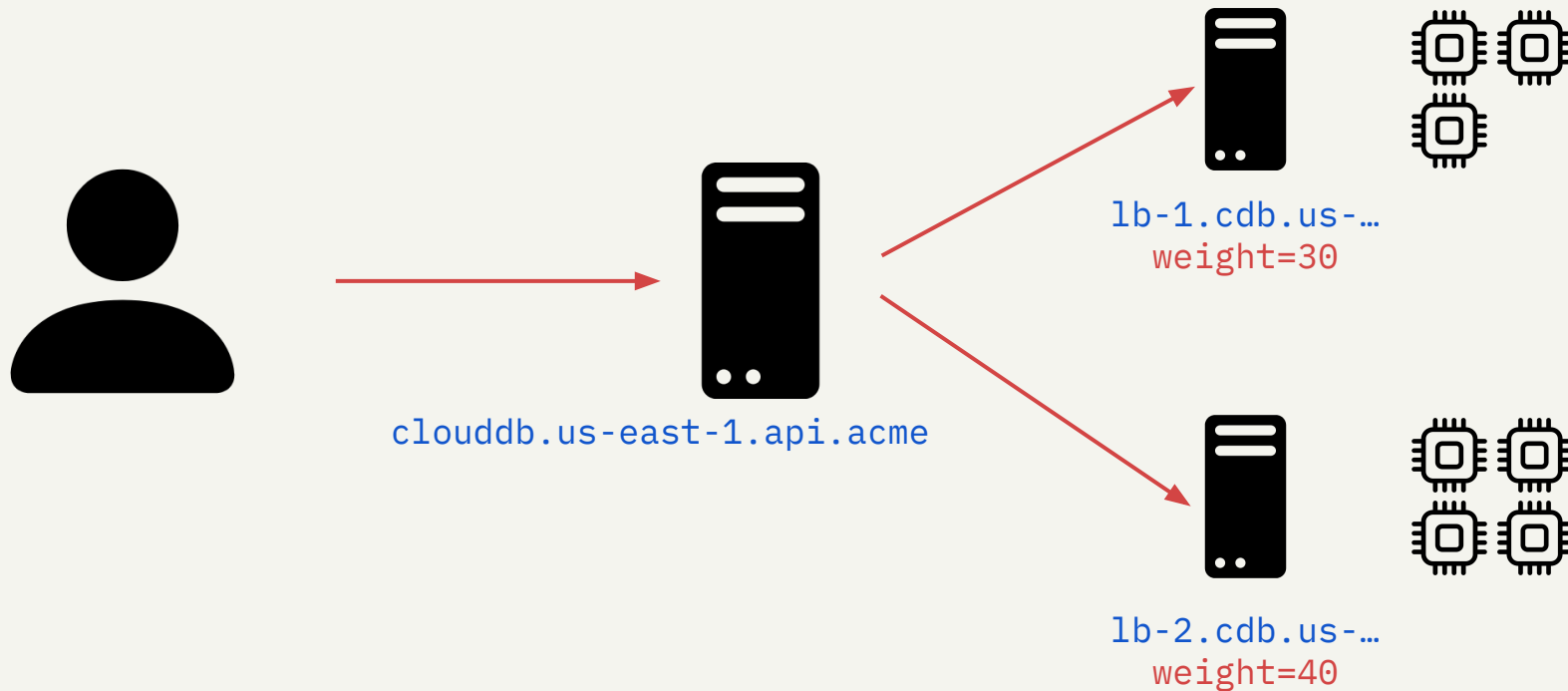
Our faulty distributed system: ACME CloudDB



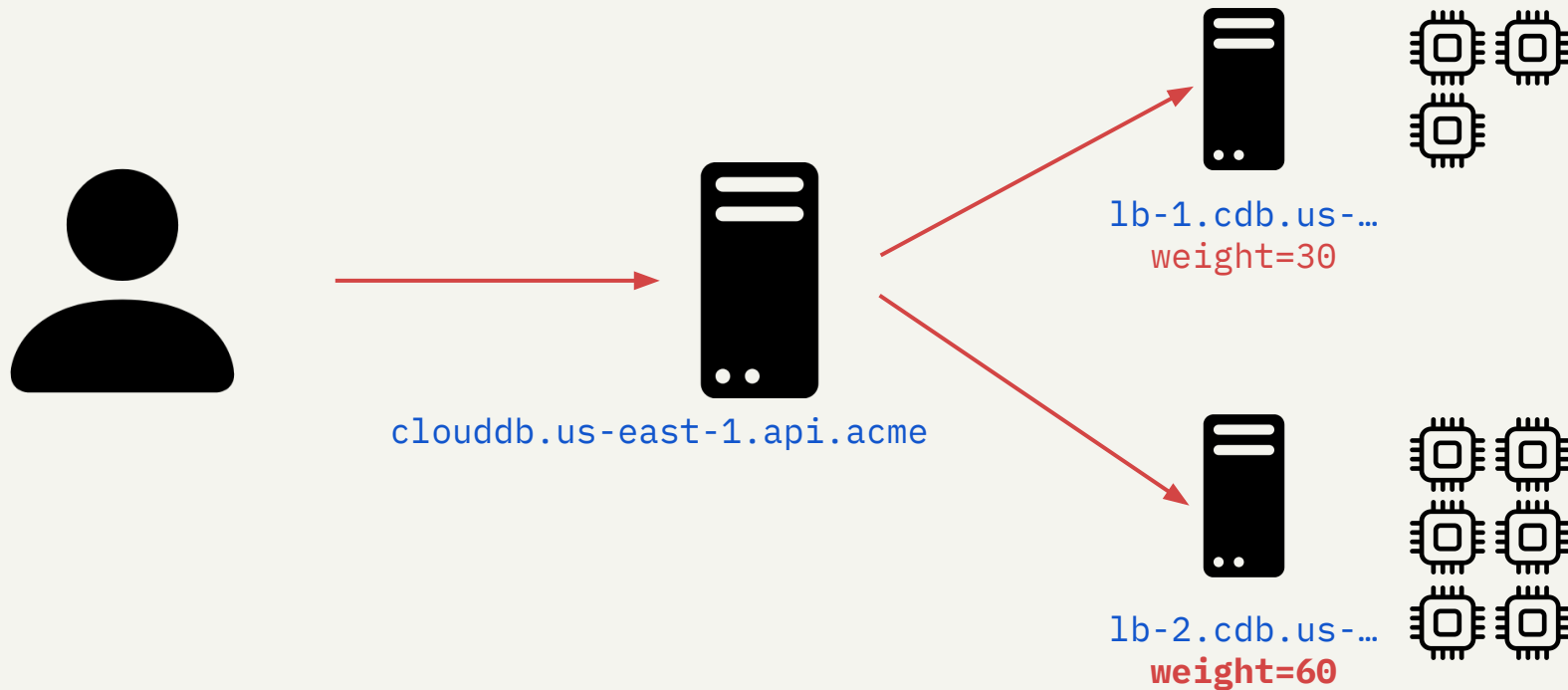
Our faulty distributed system: ACME CloudDB



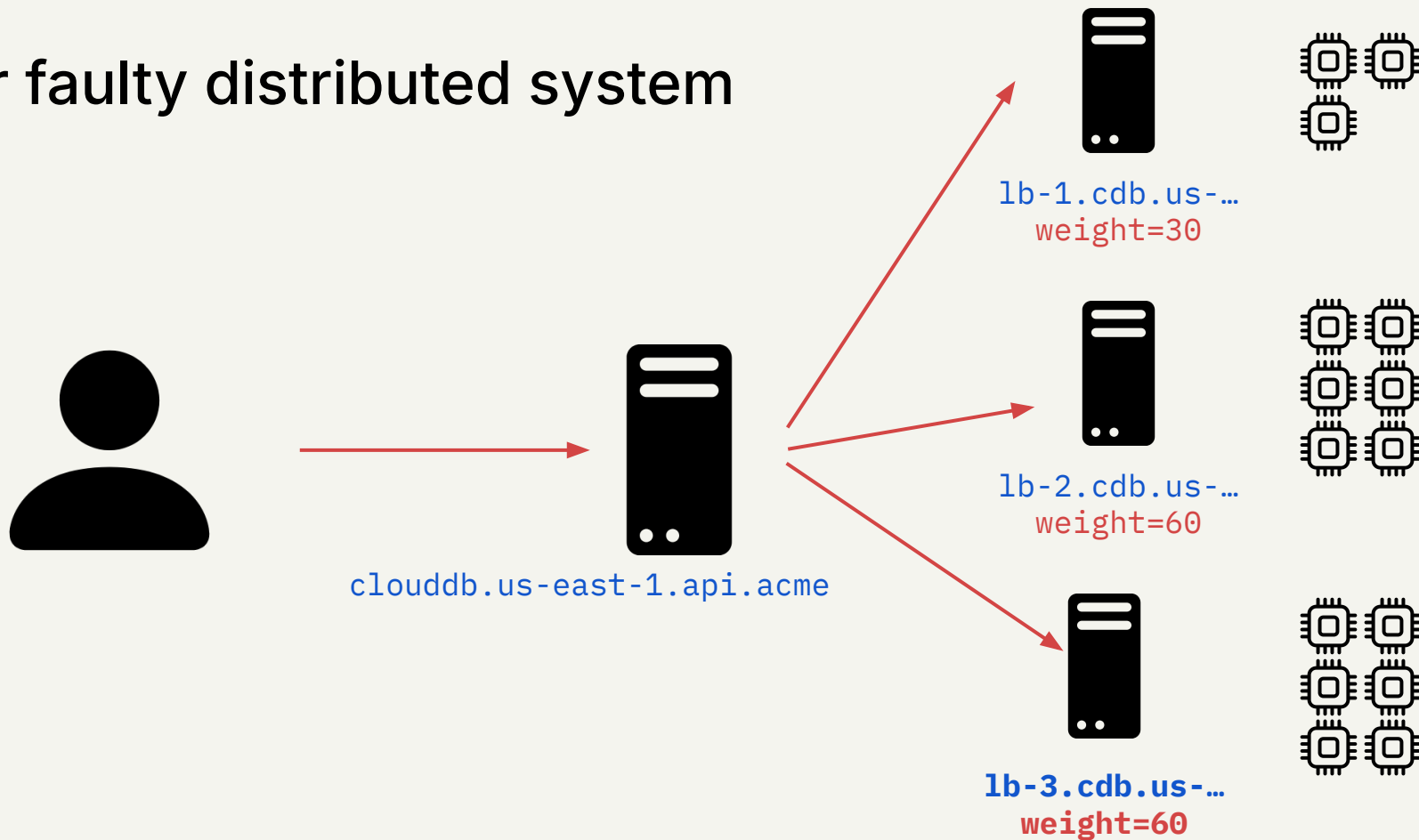
Our faulty distributed system: ACME CloudDB



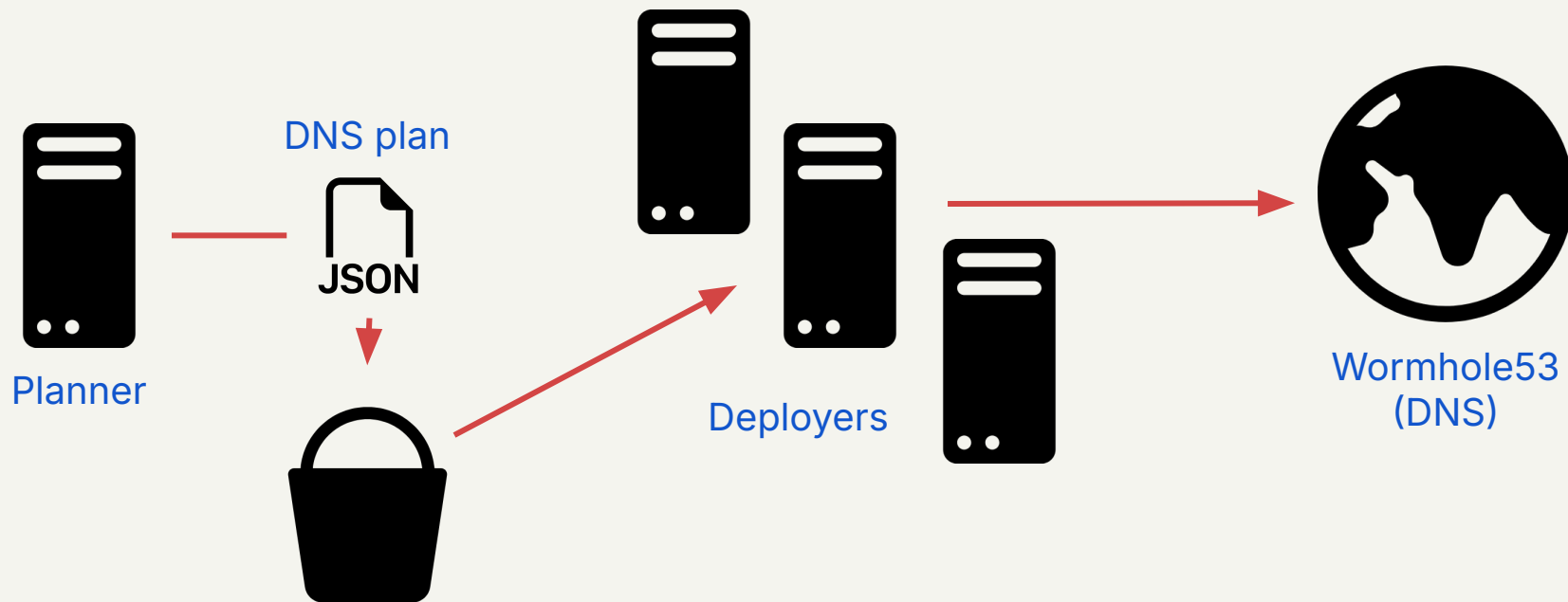
Our faulty distributed system: ACME CloudDB



Our faulty distributed system



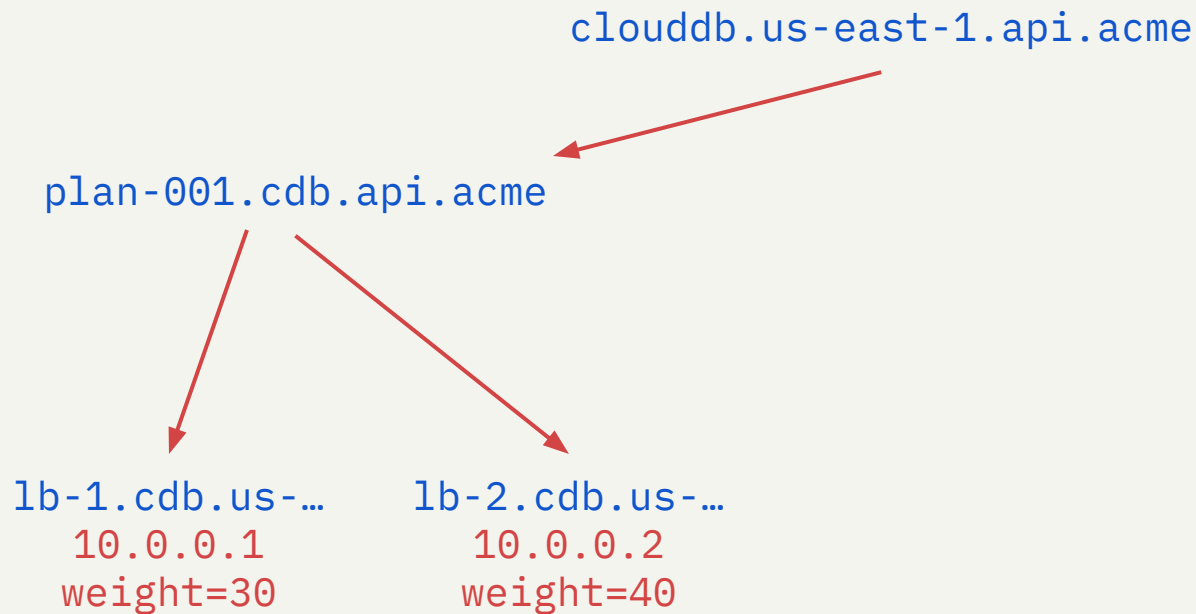
Our faulty distributed system: ACME DNS control plane



Planner: Generate plans (JSON)



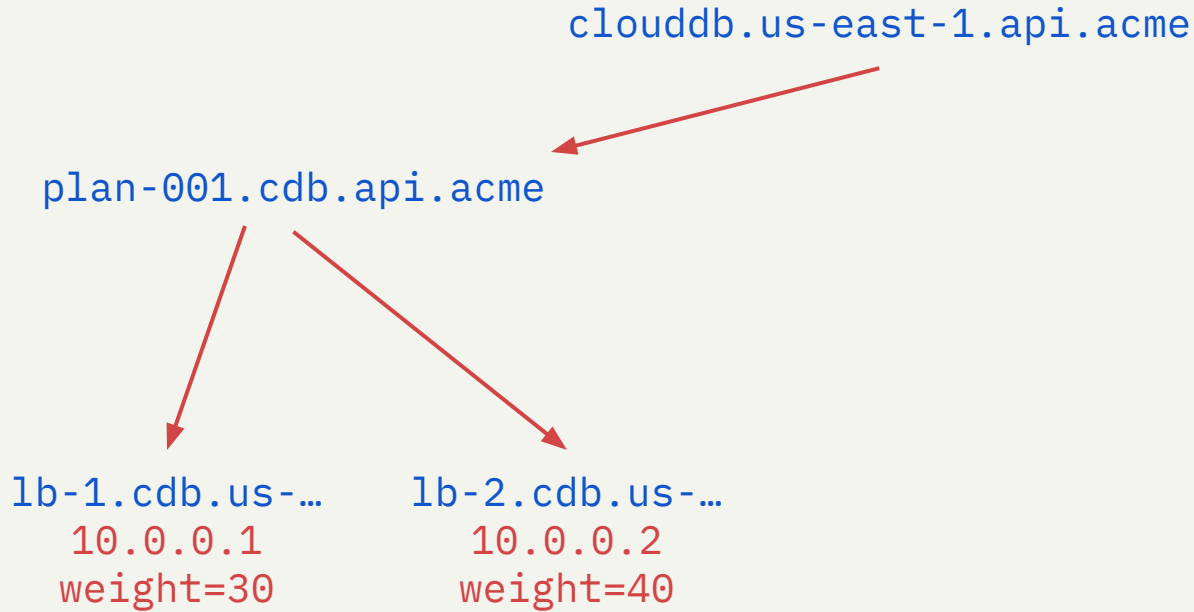
Wormhole53
(DNS)



Deployers: Deploy DNS plans



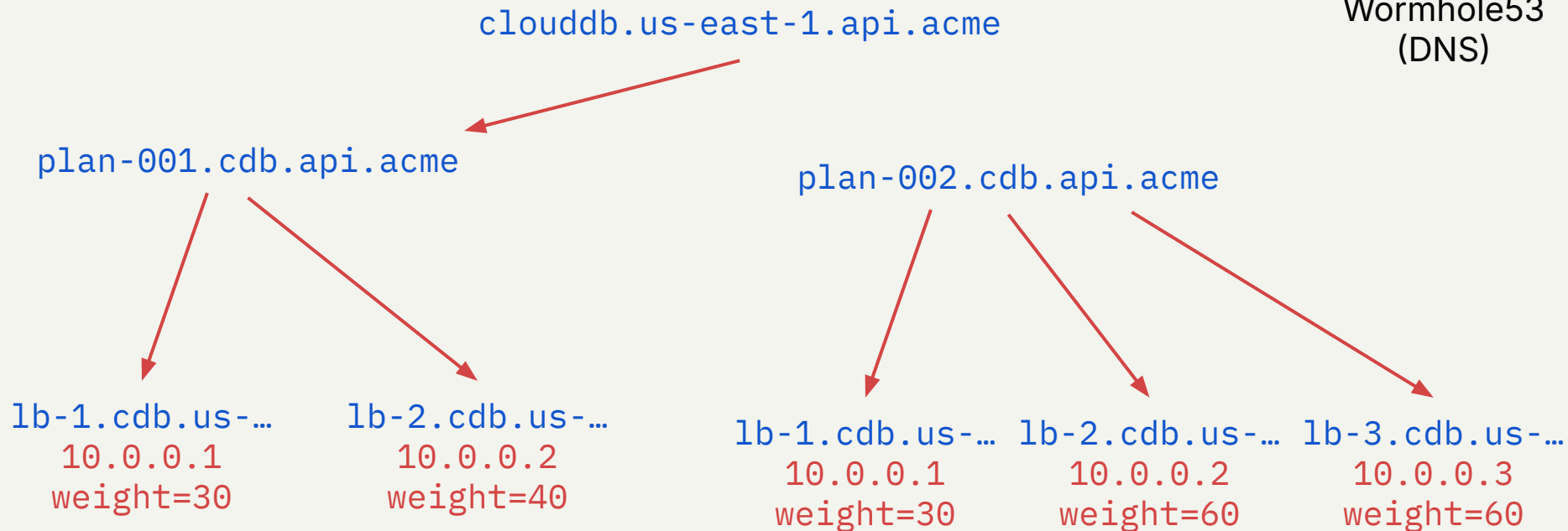
Wormhole53
(DNS)



Deployers: Deploy DNS plans



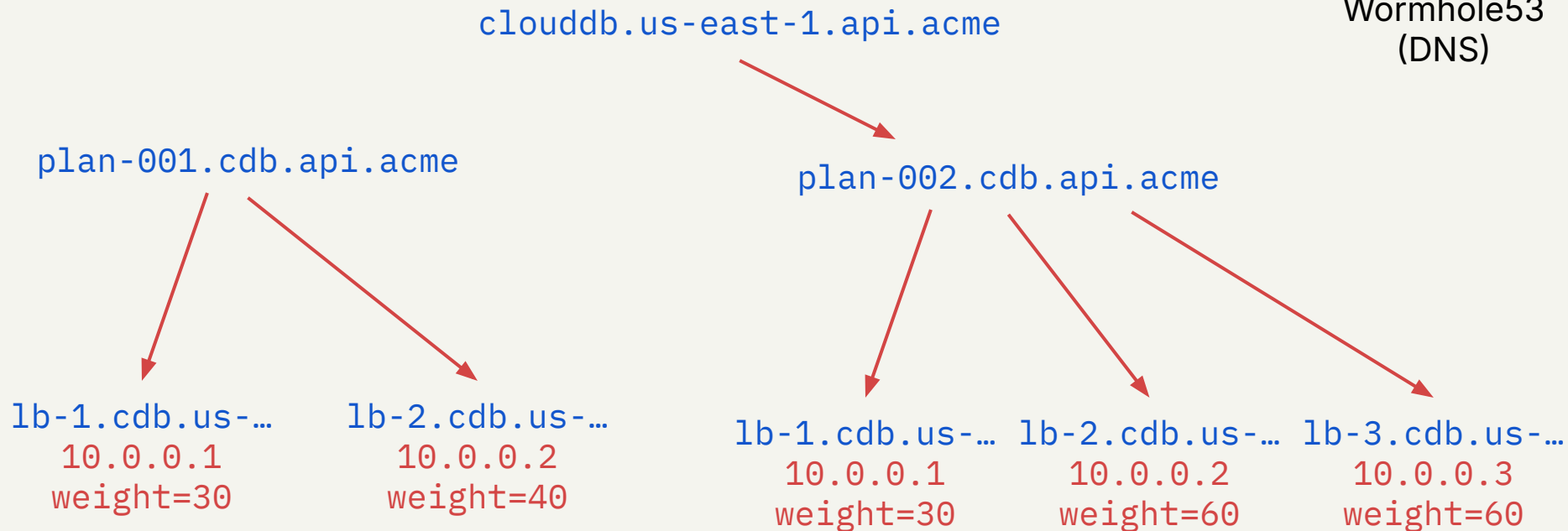
Wormhole53
(DNS)



Deployers: Deploy DNS plans



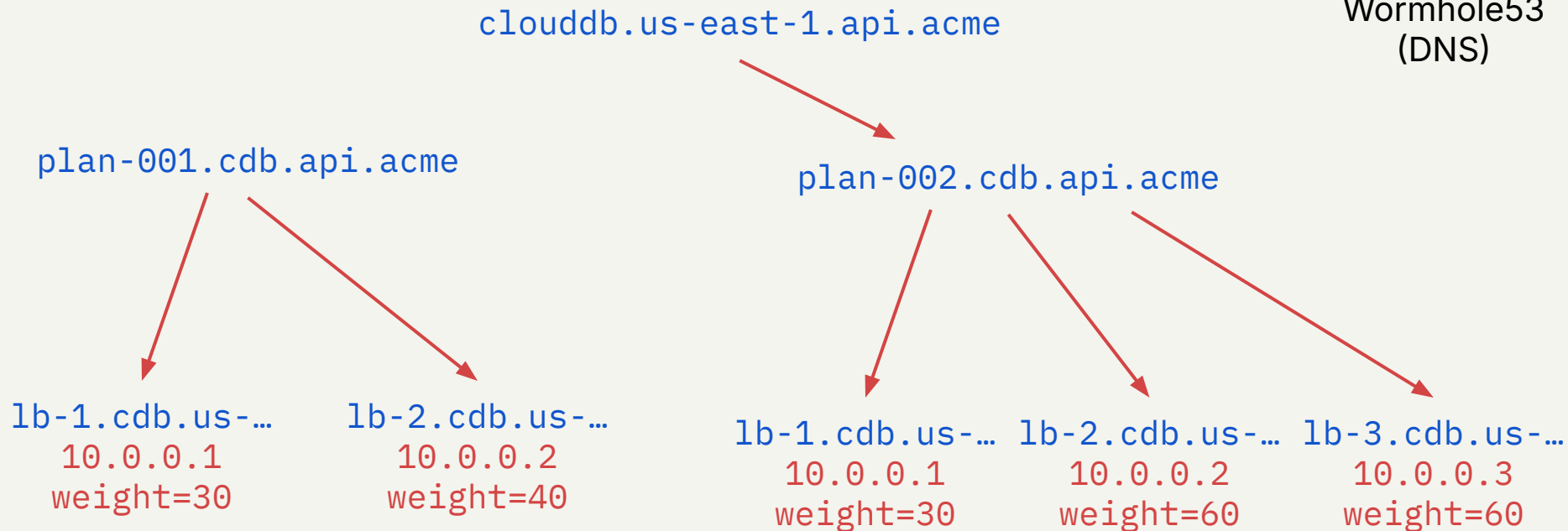
Wormhole53
(DNS)



Deployers: Garbage-collect old DNS plans



Wormhole53
(DNS)



Deployers: Garbage-collect old DNS plans



Wormhole53
(DNS)

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 kind of system?

Distributed Systems fail outside test coverage

Production bugs

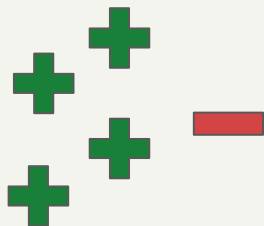
- are not on the tested paths (duh)
- appear as *multiple, unforeseen* messages / events / timeouts / retries / ...

Bad “fix”: think harder. write more tests.

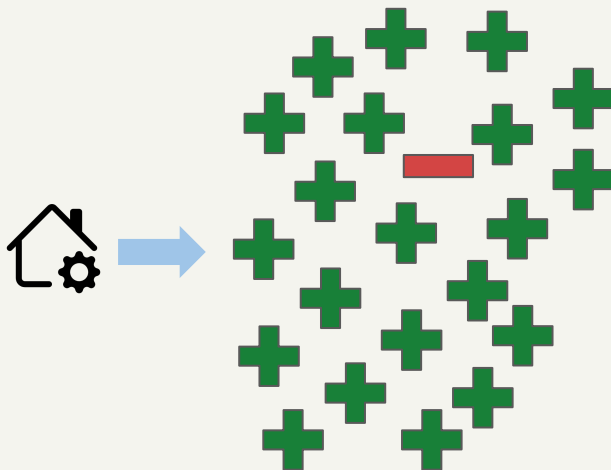
Good fix: build a machine that produces the paths we are not thinking about.

Testing things

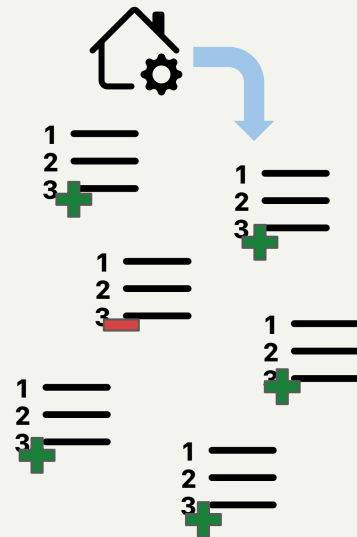
A. example-based
table-driven testing



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]*

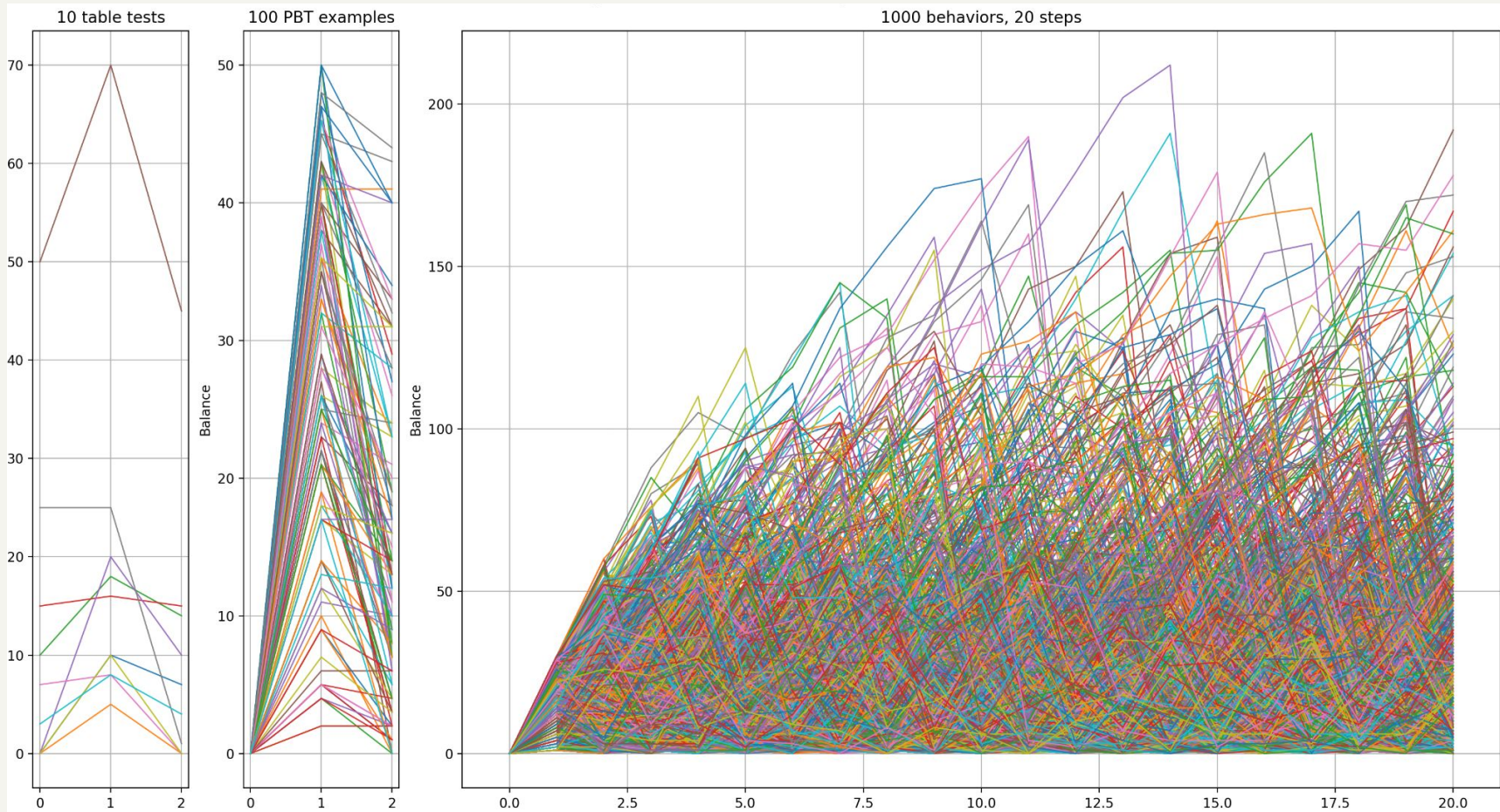
```
init:
  a = Account(0)
  state = 0
```

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

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

```
always:
  a.balance == state
```





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

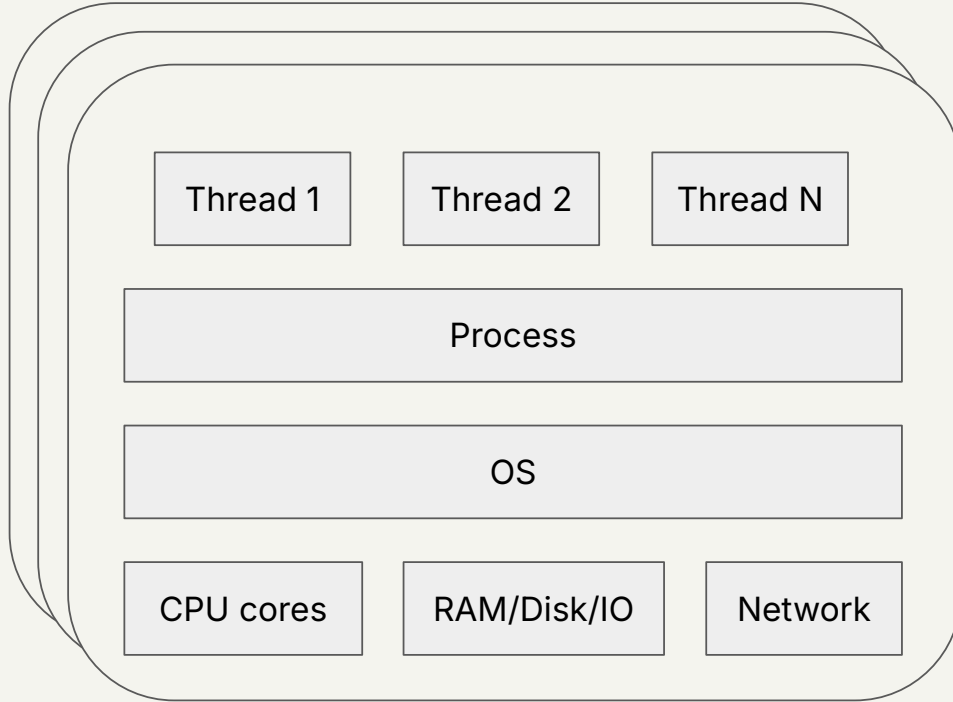
Fault injection

Chaos testing

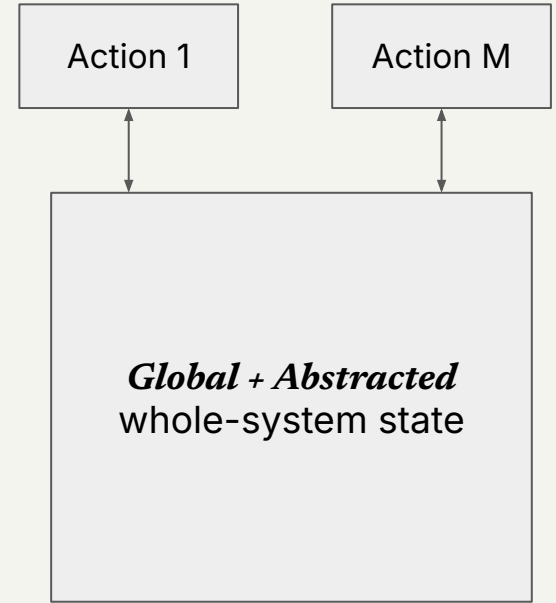
*Deterministic
simulation testing*



Why spec?



“Less physical”



What are executable specs?

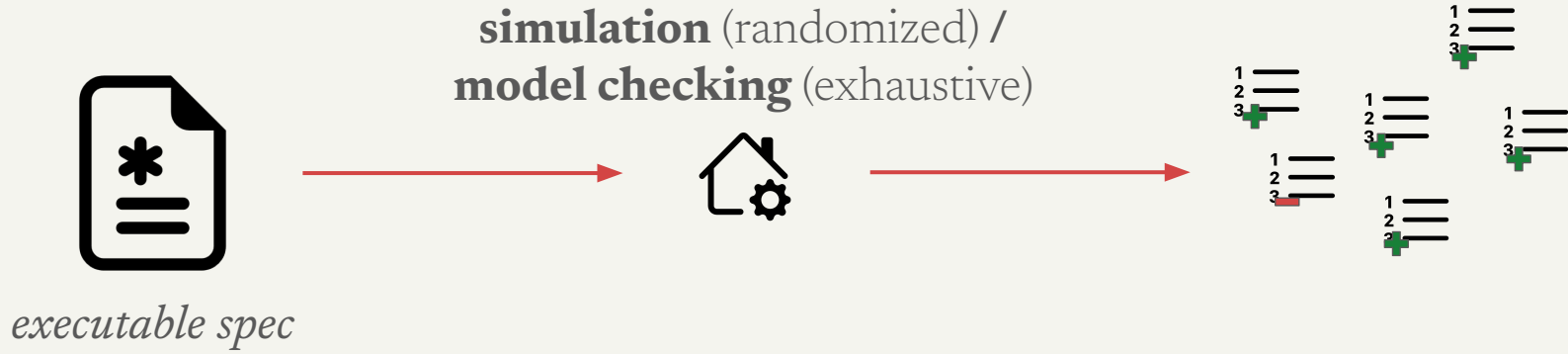
- Any high-level code that lets you focus on the hard problems
- De-focus from component boundaries & implementation details (PL idioms, memory, libraries, physical deployment)
- It could be just “high-level” Python (e.g., Ethereum EELS)

```
def reconcile(desired, observed):  
    unmanaged = {k: v for k, v in observed.config.items()  
                  if k not in desired.managed_keys}  
  
    return {**observed, "config": unmanaged | desired.config}
```

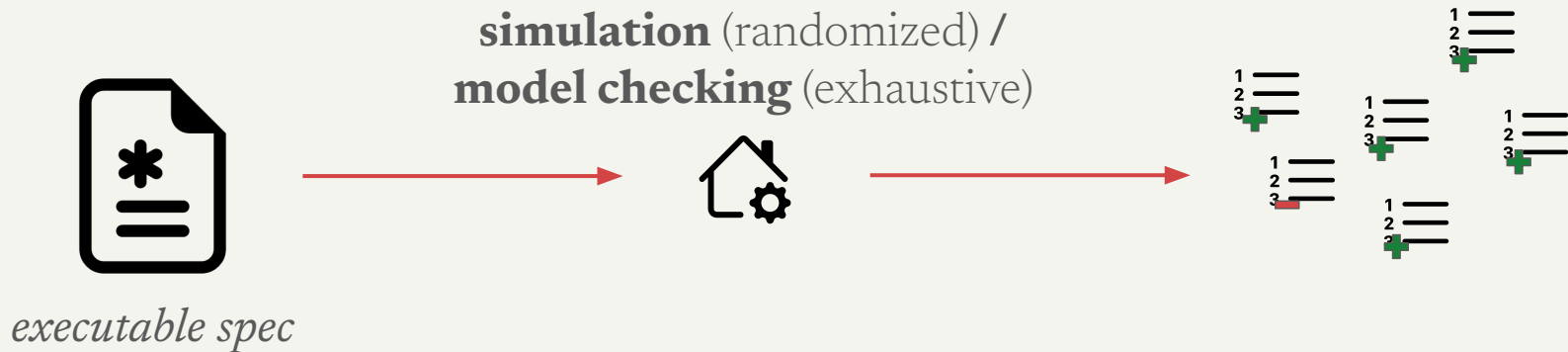
Obvious goodies: replay steps, run simulations

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!**



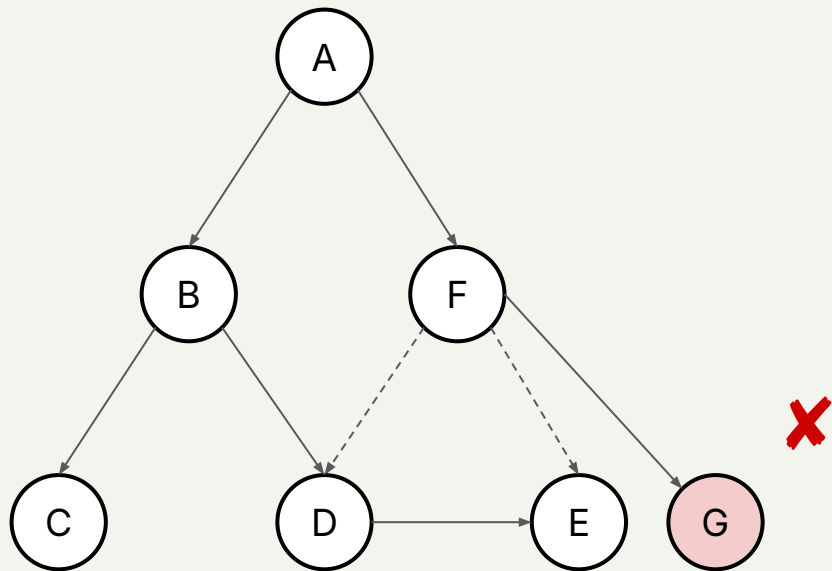
Hands-on Spec: ACME Cloud DNS control plane

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

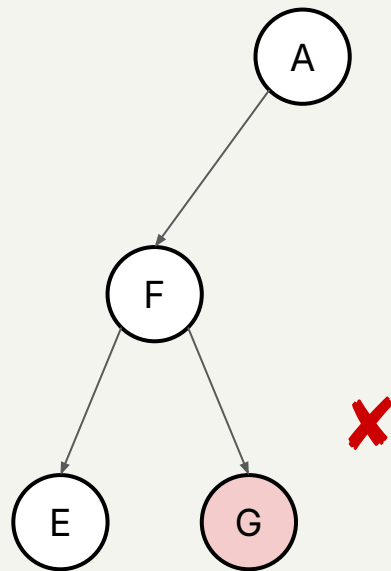


github.com/thpani/testing-distsys-devconf26

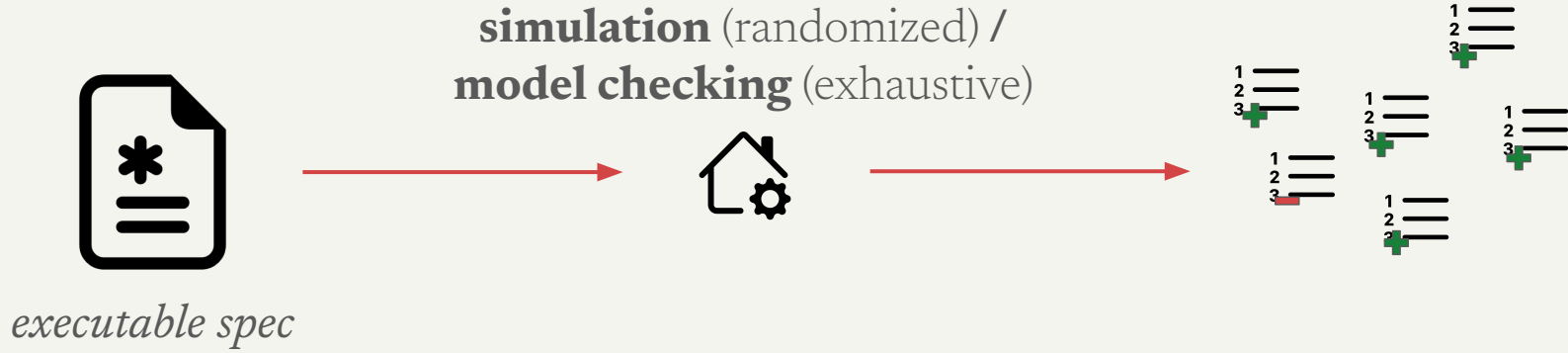
Depth-first search



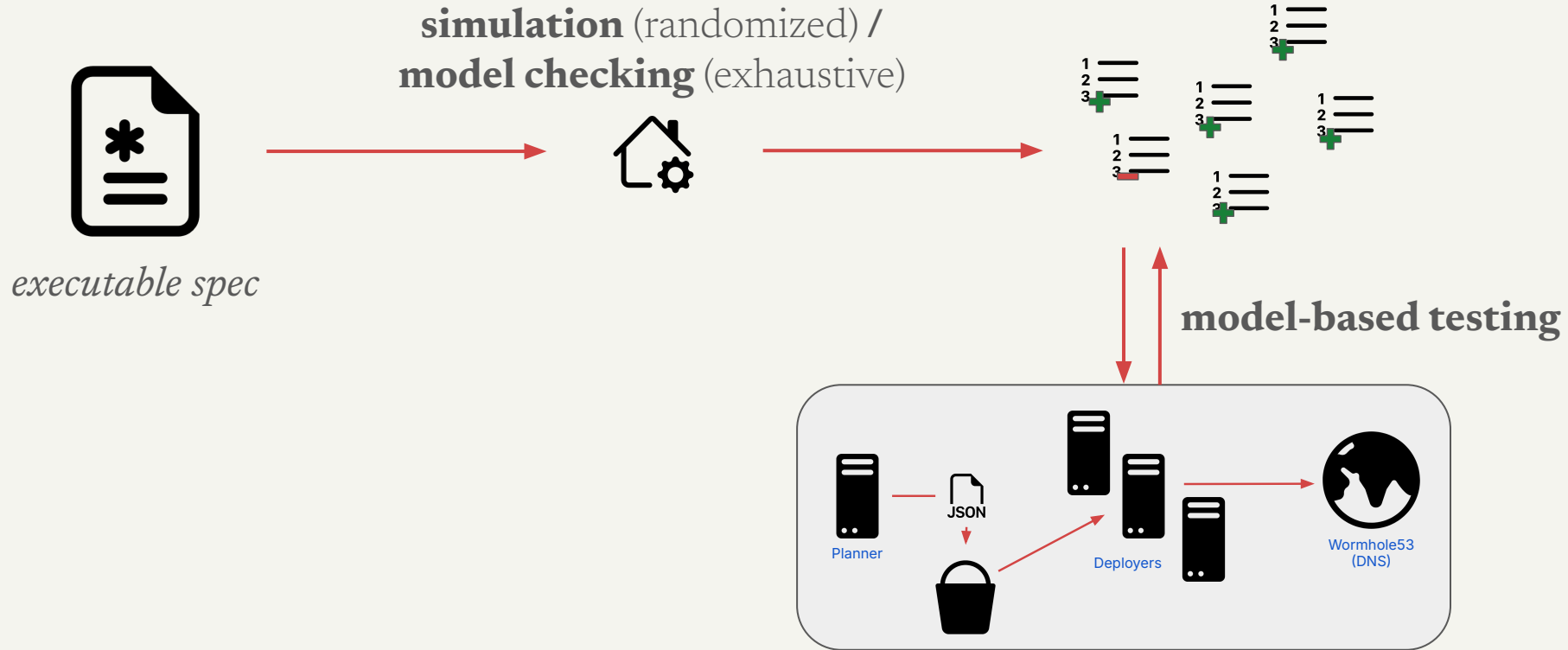
DFS with shuffle



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)

1. Find a trace to the bug

Player 2 (SUT)

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

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

Before outages cost \$\$\$ in production.

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

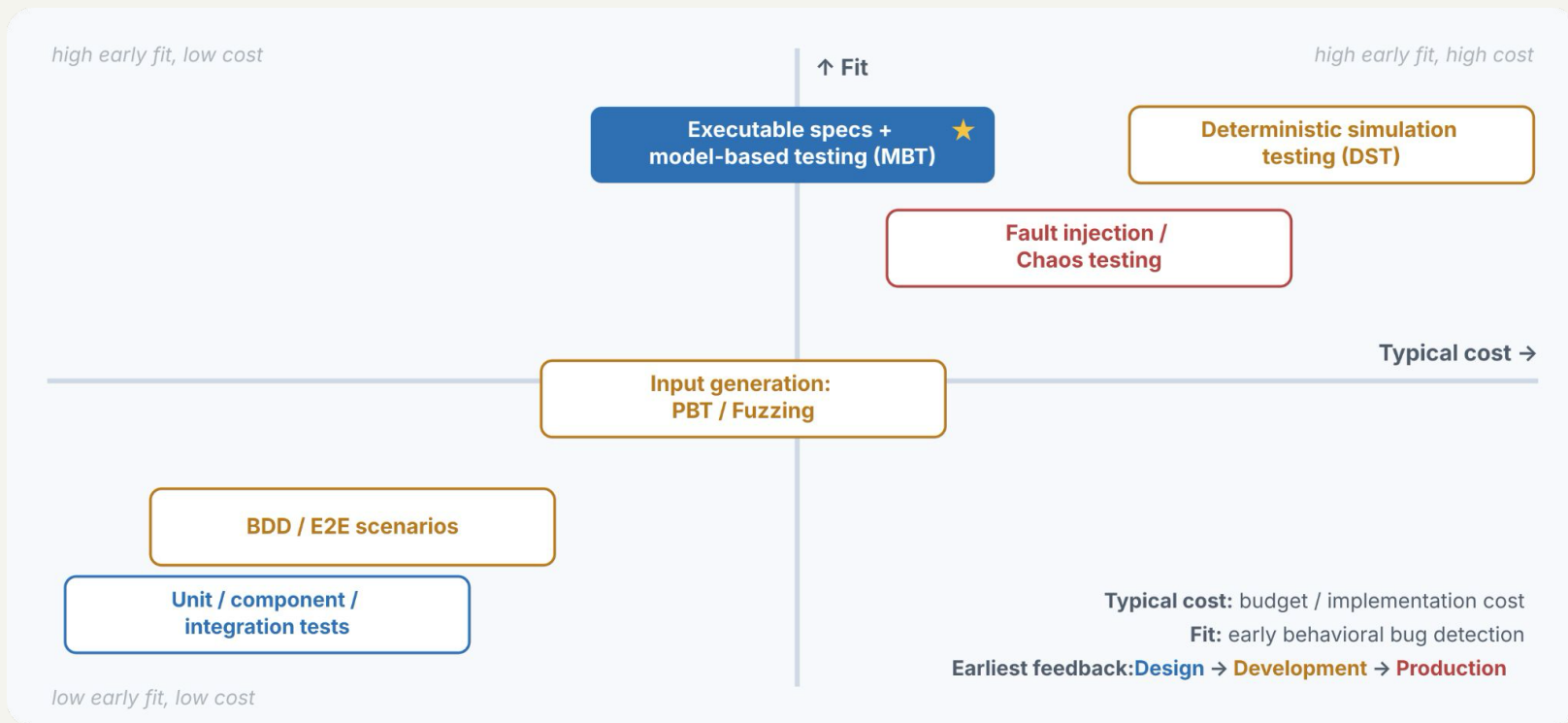
incidents are hard to reproduce

has important system invariants

production bugs cost \$\$\$\$

*... but no
executable
reference*

Testing distributed systems



"Specification"-driven development

Recent trend:

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

Our take:

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

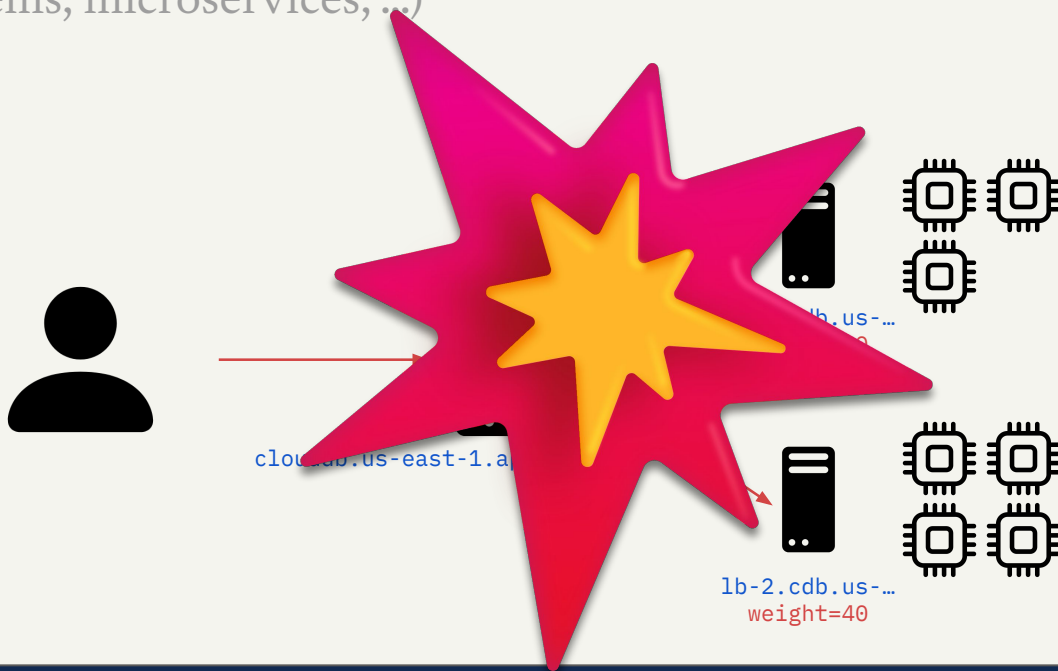
LLMs and executable specs reinforce each other! → feedback loop

What we did today

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

Focus: ideas, not tools

Practical, interactive



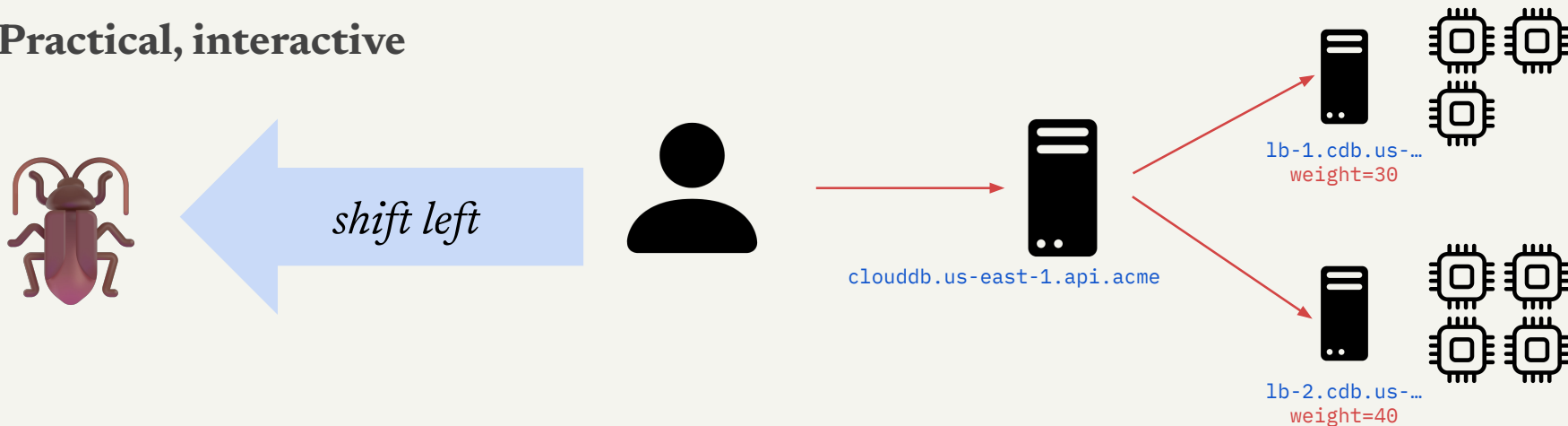
What we did today

How to **find bugs in distributed systems** *before* production

(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical, interactive



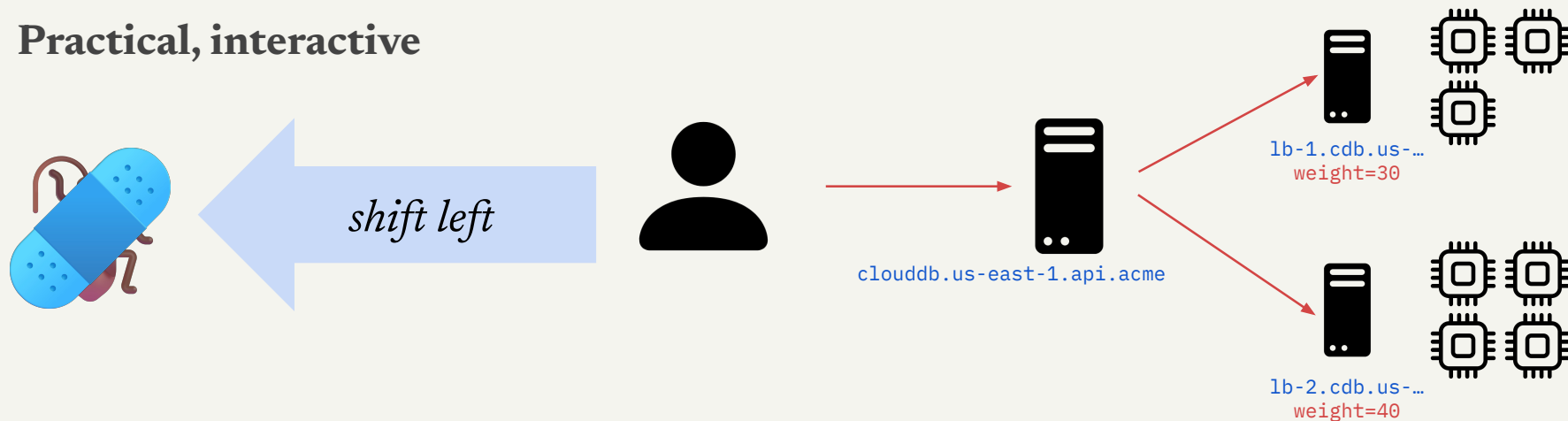
What we did today

How to **find bugs in distributed systems** *before* production

(e.g., controllers, cloud-native systems, microservices, ...)

Focus: ideas, not tools

Practical, interactive

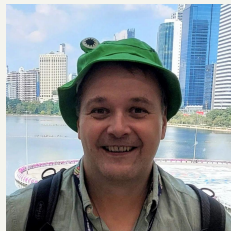


Find out if executable specifications fit *your system*

Scan the QR for workshop materials, a short system assessment, testing positioning, and pilot project details.

Pilots: 1-2 focused projects after DevConf

Best fit: control planes, schedulers, databases, protocols, stateful cloud infra



Igor Konnov



Thomas Pani



 [wunderspec/wunderspec](https://github.com/wunderspec/wunderspec)

 demo@wunderspec.com

→ wunderspec.com/devconf