SOSP 2026

RPCShield

Defending Microservices Against Cascading Failures

Milind Chabbi, Sonal Mahajan, Ivan Beschastnikh, René Just, Yuxin Wang, Yufan Xu, Elton Pinto, Seemanta Saha, Manu Sridharan, Abhishek Jha, Sandeep Koushik Sheshadri, Mayank Bansal

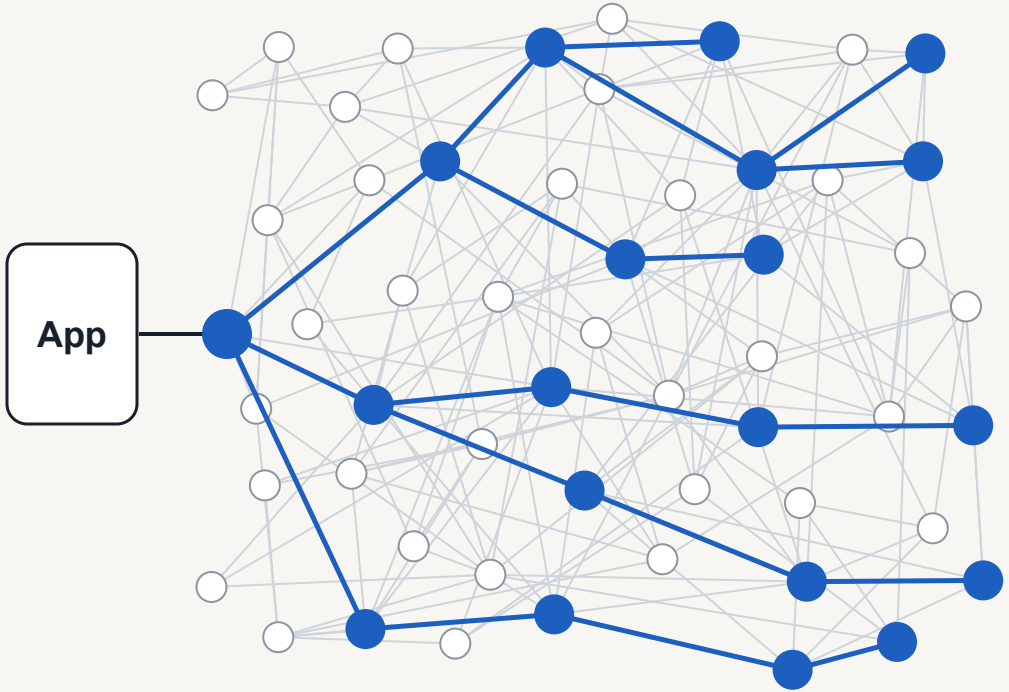
Uber Technologies · University of British Columbia · University of Washington · Georgia Tech · UC Riverside

github.com/uber-research/rpcshield

Uber

MOTIVATION

Every tap in the app fans out into many RPCs



One request fans out at many levels

6,000+

microservices

10s of trillions

RPCs per week

1.8M

outbound RPC call sites

MOTIVATION

Not all services are equally critical

	Role	Services
T0	Infrastructure	151
T1	User-facing	1,490
T2	Business-critical	965
T3	Critical tooling	1,787
T4	Employee tooling	491
T5	Auxiliary workloads	1,311

Critical: T0–T2 · **Lower: T3–T5**

~50%

of all RPCs go from a **more-critical** to a **less-critical** tier

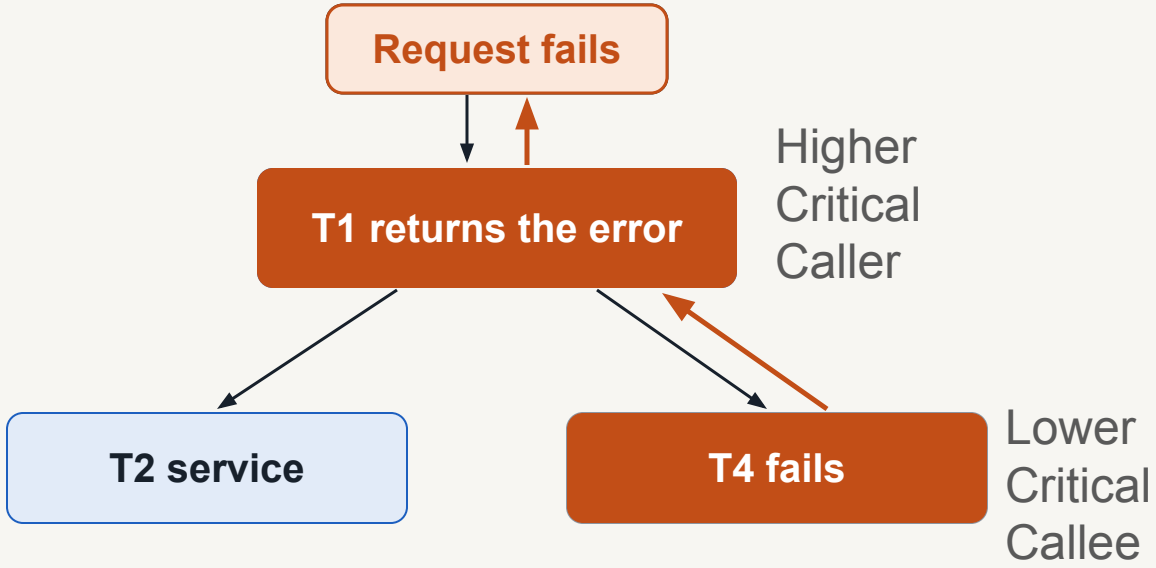
T1 alone sends **~66%** of that traffic

MOTIVATION

A low-tier failure can take down a critical service

Why would T4 fail?

- Failover preemption
- A bad deploy
- Overload, flaky API



Fail-close: the caller fails because a dependency failed

THE PROBLEM

Fail-close vs. fail-open: one line apart

Fail-close

```
func GetFare(req) (*Fare, error) {
  promo, err := promo.Get(req) // RPC
  if err != nil {
    return nil, err
  }
  return price(req, promo), nil
}
```

Promo down → fare request fails

Fail-open

```
func GetFare(req) (*Fare, error) {
  promo, err := promo.Get(req) // RPC
  if err != nil {
    promo = NoPromo
  }
  return price(req, promo), nil
}
```

Promo down → fare without promo

Identical in normal operation; the difference shows only when the dependency fails.

43

highest-severity, customer-facing outages at Uber were caused by fail-close dependencies (2020–2023)

≈ 10 every year

Industry-wide problem: cross-system errors are a leading cause of cloud incidents (Google, Azure, and AWS) — Tang et al., EuroSys ’23

THE PROBLEM

Why fail-close is hard to catch

Latent

Invisible until the dependency fails

Everywhere

~1.8M call sites, deep in call paths

Always changing

Any diff can regress a safe path

We need detection that is…

- In code
- Before deployment
- At fleet scale
- On every diff
- Attributed to code paths

THE SOLUTION

RPCShield

- ✅ In code
- ✅ Before deployment
- ✅ At fleet scale
- ✅ On every diff
- ✅ Attributed to code paths

✅ github.com/uber-research/rpcshield

KEY INSIGHT 1

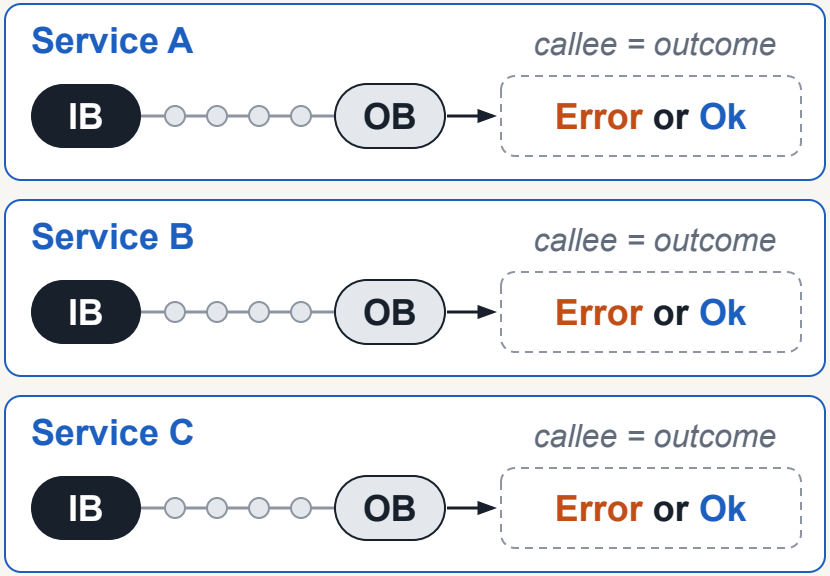
Fail-close is a per-service property

Whether an endpoint fails depends only on **its service's own code**.

A callee enters only as its **outcome**: Error or Ok.

So we can…

- analyze each service alone
- in parallel, in CI
- re-analyze only what a diff touches



One independent analysis per service

KEY INSIGHT 2

What does “caused by” mean?

```
1  func handler() (*S, error) {
2    r1, err1 := rpcA() // Primary
3    if err1 != nil {
4      r2, err2 := rpcB() // Fallback
5      if err2 != nil {
6        return nil, ErrNotFound
7      }
8      return r2, nil
9    }
10   return r1, nil
11 }
```

A fallback pattern from our production corpus

Both errors reach line 6. Which dependency is to blame?

Error reachability (taint):

blames **rpcA and rpcB**

Intuition:

blame **rpcB only**: rpcA already has a fallback

KEY INSIGHT 2

A counterfactual (“but-for”) definition

```
1  func handler() (*S, error) {
2    r1, err1 := rpcA() // Primary
3    if err1 != nil {
4      r2, err2 := rpcB() // Fallback
5      if err2 != nil {
6        return nil, ErrNotFound
7      }
8      return r2, nil
9    }
10   return r1, nil
11 }
```

A fallback pattern from our production corpus

Inbound RPC c is **fail-close** on outbound RPC **e** if, in some failure scenario:

1 e fails and c returns Error

2 had e and e **succeeded**, c returns Ok

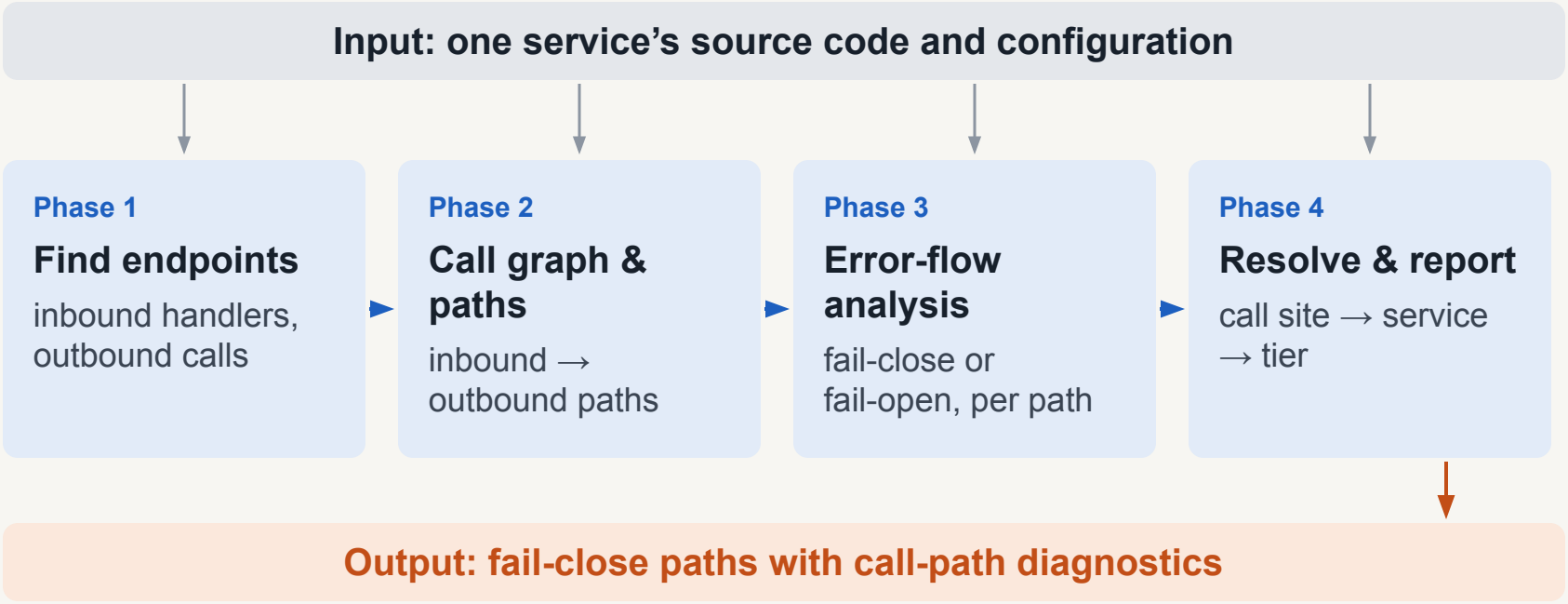
3 no RPC after e fails in either case

Scenario = an oracle fixing each RPC outcome

Intuition: isolate the cause of the failure to a single RPC

DESIGN

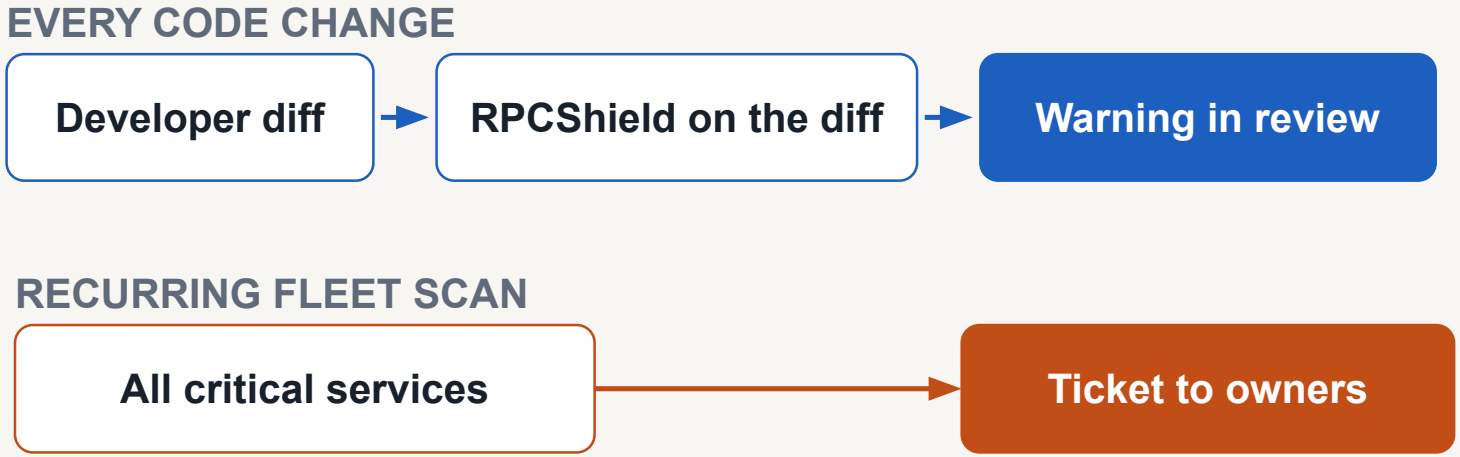
A four-phase pipeline, run per service



DEPLOYMENT

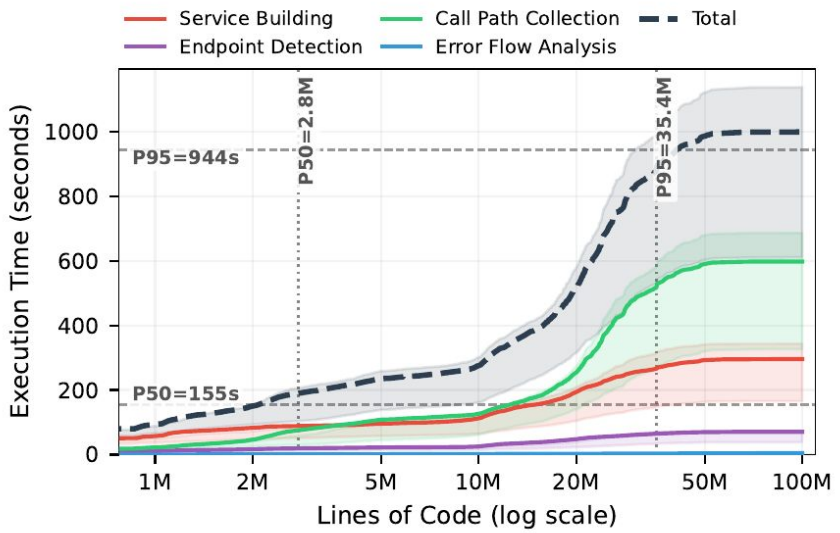
Built for production: runs on every diff

3.8K Fail-close issues found



INTRINSIC EVALUATION

Fast enough for CI, even at 35M lines



3,944 services · median Go service 2.8M lines, P95 35.4M

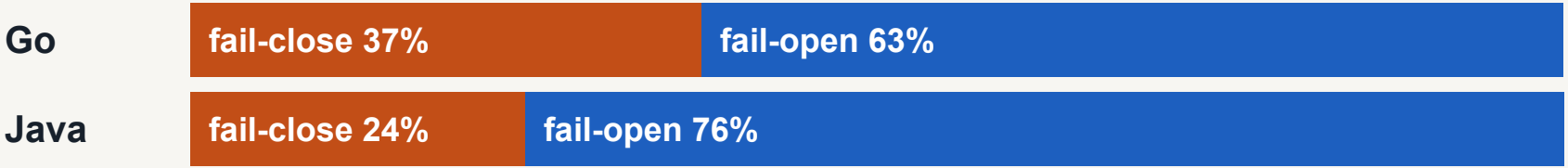
PER-SERVICE ANALYSIS TIME

	median	P95
Go	**2.5 min**	**16 min**
Java	**1 min**	**3 min**

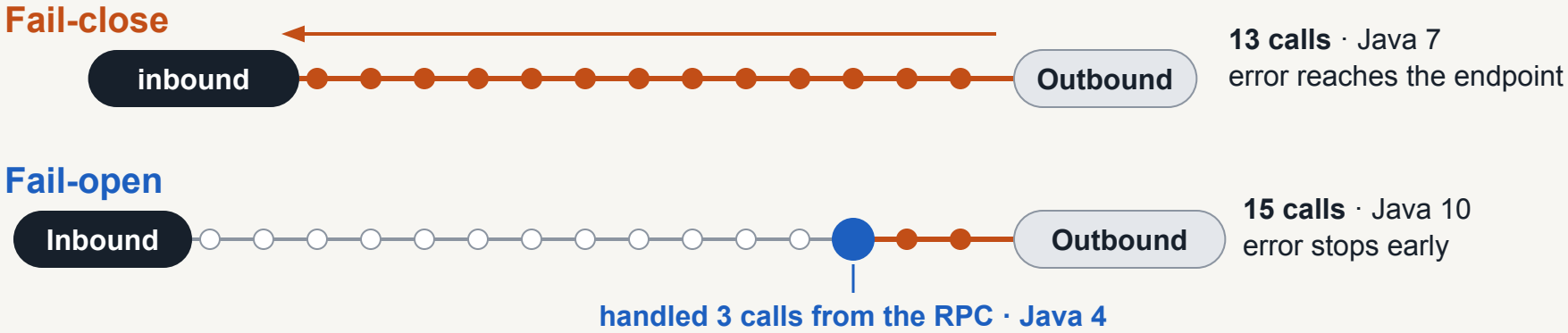
Error-flow analysis itself takes seconds.

What the paths look like: fail-close vs. fail-open

PATH SPLIT (Java: synchronous paths)

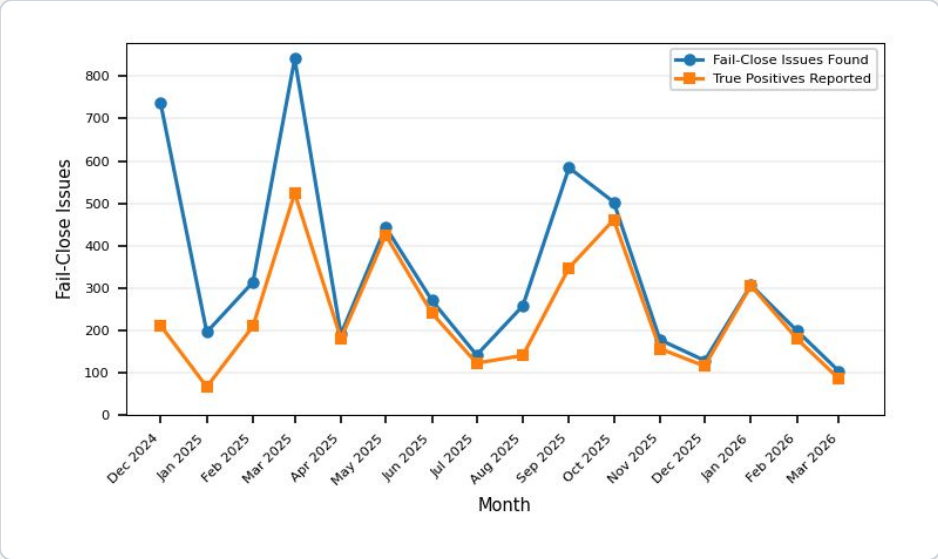


PATH LENGTH AND WHERE THE ERROR STOPS (median calls, Go · Java)



EXTRINSIC EVALUATION

Precise enough that owners trust it



Found vs. true positives per month (Go). TP-rate fit: **54%** → **97%**

< 5%

false positives

< 2.5%

missed vs. runtime analysis

43 / 43

past outages flagged

ADOPTION

Developers: real risks, quick fixes

58%

said it could have caused a major outage (L4/L5)

72%

fixed it within five days of work

1.4K

violations remediated by owners

58%, 72%: survey of 155 engineers who resolved tickets. 1,359: ticket records.

IMPACT

Fail-close outages at Uber

2020–2023

≈10

per year (43 in total)

2025

0

cross-tier fail-close incidents

IMPACT

RPCShield's assurance powers tiered SLAs at Uber

Before
~4M cores
~20% utilized

Tiered SLAs
~3M cores
~31% utilized

On failover, low tiers are preempted for critical ones.

Uber's Failover Architecture — Bansal et al., NSDI '26

~1M

of ~4M CPU cores returned

Preemption is safe **only if no T0–T2 service is fail-close on it.**

RPCShield assures this on every diff.

Contributions

THE PROBLEM Fail-close dependencies have long caused cascading outages

THE IDEA A clean, counterfactual definition of fail-close, mapped to static analysis

10/year → 0/year

top-severity fail-close outages: 2020–23 average → none attributed in 2025

~3.8K

fail-close violations caught across ~4K Go & Java services

1M of 4M

CPU cores freed by tiered SLAs, made safe by RPCShield

WHAT'S NEXT Cross-service analysis of retry storms, data flows, storage, as AI writes code few developers read

github.com/uber-research/rpcshield

Thank you! Questions?

Milind Chabbi

Uber