# RPCShield: Defending Microservices Against Cascading Failures

Milind Chabbi
Uber Technologies
Sunnyvale, CA, USA

Sonal Mahajan
Uber Technologies
Sunnyvale, CA, USA

Ivan Beschastnikh*
University of British Columbia
Vancouver, BC, Canada

René Just*
University of Washington
Seattle, WA, USA

Yuxin Wang
Uber Technologies
New York, NY, USA

Yufan Xu
Uber Technologies
Sunnyvale, CA, USA

Elton Pinto*
Georgia Institute of Technology
Atlanta, GA, USA

Seemanta Saha
Uber Technologies
Seattle, WA, USA

Manu Sridharan*
University of California, Riverside
Riverside, CA, USA

Abhishek Jha
Uber Technologies
Seattle, WA, USA

Sandeep Koushik Sheshadri
Uber Technologies
Seattle, WA, USA

Mayank Bansal
Uber Technologies
Sunnyvale, CA, USA

## Abstract

Microservice-based systems are tightly interdependent. A failure in one service can propagate to a dependent service, potentially bringing down critical, user-facing functionality. We have developed and deployed RPCShield, a suite of program analyses for Go and Java that detects such cascading-failure risks. RPCShield uses static program analysis to track error and exception propagation through service code. Two insights make this practical. First, error propagation needs to be only checked per service, so the analysis is intra-service and avoids inter-service analysis entirely. Second, deciding the property counterfactually, by asking whether an endpoint could have succeeded had a given call succeeded, attributes the failure to the specific responsible dependency rather than to every dependency whose error reaches the caller. We have deployed RPCShield at Uber, across afl eet of over 6K microservices. Since deployment, it has detected over 3,800 cascading failure risks, of which service owners have already remediated more than 1,300. Manual review indicates that RPCShield has a false positive rate of around 5%.

*Work performed while at Uber Technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on thefi rst page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SOSP '26, Prague, Czech Republic*
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843878

***CCS Concepts:*** • **Software and its engineering** → *Software maintenance tools*; **Automated static analysis**; **Software reliability**.

***Keywords:*** Fail-close violations, counterfactual causation, static analysis, remote procedure calls.

## 1 Introduction

Large-scale microservice platforms accumulate thousands of inter-service dependencies over time [9, 24, 38, 54]. Many of these dependencies are benign: if a downstream service is slow or unavailable, the caller service degrades gracefully and continues serving its own clients. But some dependencies are *fail-close*: a failure in the downstream service causes the caller service to fail as well, propagating the error to its own callers and potentially triggering a cascading outage. Fail-close dependencies are latent and a form of *gray failure* [20]. These failures remain invisible during normal operations and reveal themselves only when a downstream service actually fails.

Cross-system interaction failures, including unhandled error propagation across service boundaries, are a major cause of production incidents at major cloud providers [40]. The consequences of fail-close dependencies surface in several operational scenarios: a *higher-criticality* service that propagates errors from a *lower-criticality* callee service will fail if that callee is preempted during a failover; a service calling an unreliable third-party API without error handling inherits that API's failure rate; and even during routine partial outages or rolling deployments, an unhandled RPC error in one service can cascade through its callers. A single overlooked fail-close dependency can turn a contained failure into a widespread outage.

Detecting fail-close dependencies in such systems is difficult. Thefi rst challenge is that we need a definition that captures the essence of fail-close dependency and can be used

to build practical detection analyses. Second, with thousands of services, tens of thousands of endpoints, and hundreds of millions of lines of code, we need an automated and scalable detection approach. This approach must detect fail-close dependencies *in code*, *before deployment*, and *at scale*, with enough diagnostic detail for developers to address them.

We address these challenges by making two key observations. First, fail-close can be formally defined as a counterfactual code-level property. Our formulation (§3) is language-agnostic, handles both synchronous and asynchronous RPCs, and works in the context of multiple RPCs to different endpoints. Second, fail-close is a property of error propagation *within a single service's code*. That is, a service's inbound RPC endpoint has a fail-close dependency on an outbound RPC to another service if an error can propagate from the outbound call site through the service's internal function calls and error handling logic to an error exit of the inbound RPC. Other services are only relevant to the fail-close property through the outcomes they deliver to the caller service. At scale, fail-close dependency safety, therefore, decomposes into independent per-service analyses, each an inter-procedural problem solvable in isolation and in parallel.

Not every fail-close dependency is a defect. In fact, most fail-close dependencies in a largefl eet are benign. A service that faithfully reports a downstream failure to its own caller may behave exactly as expected, but a fail-close dependency becomes a liability if it violates a reliability policy. So, we distinguish the mechanism from the policy: a fail-close *dependency* is a code-level property of how a service propagates errors, while a fail-close *violation* is a fail-close dependency that the deployment's reliability policy forbids (§2.2).

Based on these insights, we designed and developed RPCShield, a program analysis pipeline implemented separately for Go and Java. RPCShield takes a single service's source code and determines, for each inbound RPC endpoint, whether any reachable outbound RPC's outcome can propagate an error back to the caller. It does so by identifying endpoints, constructing intra-service call graphs, and analyzing error propagation along each path. At Uber, where services are classified into criticality tiers, RPCShield cross-references the results with tiering metadata tofl ag cross-tier violations. That is, RPCShield analyzes code tofi nd fail-close dependencies and then reports those dependencies that are cross-tier violations.

Making RPCShield effective required us to address several implementation challenges: a service-level analysis is a whole-program static analysis with complex call graphs, subject to scalability and imprecision challenges; real-world services use varied RPC protocols, which must be recognized to identify RPC program points; and real services use asynchrony, framework-mediated dependency injection, and varied error-handling idioms, all requiring specialized support to trace error propagation. RPCShield handles these challenges and is deployed across Uber's microservice platform with over 6K services. Since deployment, RPCShield has detected over 3,800 fail-close violations. Service owners have remediated 1,359 of them (421 by changing the code and 938 by re-tiering the callee). RPCShield's false positive rate is around 5%.

**Table 1.** Service tiers and scale. Endpoint counts are unique endpoints observed as of August 2026.

| Tier | Description | Services | Endpoints |
|---|---|---|---|
| T0 | Infrastructure and critical applications | 151 | 1.5K |
| T1 | Critical user-facing requestfl ow | 1,490 | 17K |
| T2 | Business-critical applications | 965 | 12K |
| T3 | Internal tools critical to applications | 1,787 | 15K |
| T4 | Internal tools used by employees | 491 | 1.6K |
| T5 | Test and auxiliary workloads | 1,311 | 5K |

The main contributions of this paper are:

- A formalization of fail-close as a code-level property, using counterfactual reasoning and failure isolation (§3).
- An operationalization of the formal model using an error-propagation reachability analysis to make per-service analysis tractable and scalable for production services containing tens of millions of lines of code (§4).
- A multi-phase static analysis pipeline for detecting fail-close RPC dependencies, instantiated for Go and Java with language-specific solutions for concurrency patterns, dependency injection, and lambda expressions (§5).
- An intrinsic evaluation that reports on production service characteristics (fleet coverage and endpoint detection) and analysis characteristics (call path statistics, path exploration, and performance) (§6).
- A production deployment across 6K microservices, integrated into continuous integration (CI) and continuous deployment (CD); and an extrinsic evaluation of its use in failover safety and a report on developer adoption (§5.3, §6).

RPCShield runs in production at Uber across itsfl eet of Go and Java services. It has been key to Uber's transition to a differentiated fail-over reliability architecture [7], which has returned about 1M out of 4M CPU cores. RPCShield is open sourced at https://github.com/uber-research/rpcshield/.

## 2 Background and Motivation

We describe Uber's microservice architecture, explain why fail-close dependencies are a concern, and identify the gaps in existing detection approaches.

### 2.1 Platform Architecture

Uber operates a microservice architecture with over 6K deployed services across different environments (production, staging, canary, etc.). Table 1 shows how these services are organized into business-criticality tiers from T0 (most critical) through T5 (least critical). The overall system handles tens of trillions of RPC requests per week.

Each service exposes zero or more *inbound* RPC endpoints and may make *outbound* RPCs to other services. The resulting

dependency graph is dense, and a cross-tier traffic analysis showed that roughly 50% of the total weekly RPCs are made from a more critical tier to a less critical one. T1 services alone account for nearly 66% of all cross-tier traffic to lower-tier services. Each of these cross-tier RPC sites is a potential location for a fail-close violation.

### 2.2 Why Fail-Close Dependencies Matter

In a tiered architecture, not all services are equally important. A T1 service handling a critical user-facing requestfl ow must stay available during a regional failure, while a T5 internal reporting tool can tolerate minutes or hours of downtime. Fail-close dependencies undermine this tiering in several ways:

**Capacity oversubscription and failover.** The tiering asymmetry enables a design in which, during failures, lower-tier services are preempted to free capacity for critical services. But if a T1 service calls a T4 service and propagates the error when that call fails, preempting the T4 service takes the T1 service down with it. Uber's differentiated failover architecture [7] treats the elimination of cross-tier fail-close dependencies as a precondition for preempting a less critical tier service; RPCShield is the static analysis it uses tofi nd violations.

**Graceful degradation.** Any outage, misconfiguration, or deployment error in a lower-tier service can cascade upward to the critical services that depend on it [40, 52].

**General service resilience.** A fail-close dependency on an external service is particularly dangerous: a caller inherits the external service's failure risk without any ability to control it, and hence must expect and handle failures.

**From analysis to policy.** Uber's platform has experienced incidents in all three categories above. It has adopted a policy requiring that critical services (T0–T2) not have fail-close dependencies on lower-tier services.

RPCShield enforces this policy and turns a fail-close dependency into a violation in our deployment. A *fail-close violation* is a fail-close dependency of a caller service (e.g., T1) on a callee service with a strictly lower criticality tier (e.g., T5). Other fail-close dependencies are not violations and are not reported by default. The analysis itself is policy-agnostic and classifies dependencies as fail-close or fail-open. The tier comparison is applied afterward, when outbound call sites are resolved to services (§4.6). An alternative deployment can customize which dependencies it considers unacceptable without modifying the underlying analysis.

The tier rule is not the only policy worth enforcing. For example, a team may consider a fail-close dependency on a third-party API a violation tofi nd code that would benefit from fallback logic, caching, or circuit-breaker patterns [31].

### 2.3 Why Existing Approaches Are Insufficient

**Manual code review** does not scale. With hundreds of engineering teams deploying independently and thousands of endpoints per tier, code auditing across all call paths is impractical.

**Listing 1.** A handler that falls back to `rpcB` when `rpcA` fails. Errors from both calls reach the same return statement, but the handler is fail-close only on `rpcB`.

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

**End-to-end testing and fault injection** validates complete userfl ows under simulated failures [19, 29]. Unlike a static approach, however, tools in this space require a concrete input to drive an execution towards a failure. As well, localizing the responsible dependency within a failed multi-service call chain may require manual investigation.

**Runtime dependency analysis** can instrument production RPC traffic andfl ag dependencies as fail-close based on caller-callee failure correlations. This approach has four limitations: it requires the code path to be exercised in production with sufficient samples for robust estimations; it misses rarely-executed or not-yet-deployed paths; it operates post-deployment and discovers regressions only after reaching production; and the observed correlations may not be causal.

**Whole-fleet static analysis** is the natural static formulation of the problem: trace an error from its origin in one service, across RPC boundaries, to a user-visible failure in another. At Uber's scale this is impractical, as it presupposes a consistent source snapshot of every service, a call graph that crosses RPC and therefore language boundaries, and re-analysis of thefl eet whenever any one service changes. RPCShield avoids these concerns by checking a property that can be decided independently for each service (§3.3), and the remediation it points to is a change local to each service.

**The gap** is a tool that can detect fail-close violations *in code*, *before deployment*, *at scale*, *on each code change* and with enough diagnostic detail for developers tofi x the issue without extensive debugging. RPCShield fills this gap.

## 3 Formal Semantics of Fail-Close Behavior

"Fail-close" is a colloquial term commonly used to refer to cascaded failures, but it lacks a precise definition. We formalize fail-close over execution traces using counterfactual reasoning, and §4 shows how our static analyses approximate it.

Determining whether an inbound endpoint has a fail-close dependency on an outbound RPC requires careful reasoning about that endpoint's failure-handling behavior. As an example, consider Listing 1, which shows a Go-based inbound RPC handler from our corpus. The handler calls an outbound `rpcA` and falls back to invoking an outbound `rpcB` if `rpcA`

fails. Since both RPCs must fail in order to reach line 6, an analysis based on error reachability (e.g., taint analysis) would blame both. But intuitively, since the call to `rpcB` serves as the fallback for a failure of `rpcA`, only `rpcB` should be reported. The key goal of our formalism is to capture this intuition in a rigorous manner and guide our static analysis designs.

### 3.1 Programs, Traces, and Responses

We model the execution of a service's inbound endpoint $c$ as a deterministic program whose state evolves as it processes a request $r$. The program interacts with external dependencies through outbound RPCs and consumes their outcomes.

***Program state.*** An abstract *program state* $\sigma \in \Sigma$ captures the full execution context for an endpoint at any point during request processing. Concretely, this may include local variable values, heap values, the call stack, the current program counter (which captures RPC sites), etc.

***Oracle.*** External dependencies are non-deterministic; an *oracle* $O$ resolves that non-determinism,fi xing the outcome of each outbound RPC the request makes to either success or failure. A choice of oracle is thus a choice of failure scenario.

Let $\mathcal{D}$ be the set of dependency endpoints that $c$ may call. Let $j$ be a global call counter that tracks the order in which RPC outcomes are consumed by the program, incremented after each event. When the program wants to consume the result of its $j$-th outbound RPC in state $\sigma$, invoking dependency $d \in \mathcal{D}$, the oracle returns $O(j,\sigma,d) \in \{success, failure\}$ — the ***RPC outcome***.

***Execution trace.*** An endpoint execution produces a trace $\tau = \langle e_1, e_2, \ldots, e_n, \rho \rangle$, a sequence of outbound ***RPC events*** $e_i$, together with an ***inbound response*** $\rho \in \{Ok, Error\}$. Intuitively, the inbound response is the response a client of an inbound endpoint would receive in a given failure scenario.

Each RPC event $e_i = (\sigma_i, d_i, o_i)$ records:

- $\sigma_i$, the full program state at RPC outcome consumption,
- $d_i$, the target dependency,
- $o_i$, the RPC outcome.

Because $c$'s controlfl ow may branch on RPC outcomes, changing the oracle can change subsequent call events, and hence the inbound response. We write $exec(c, r, O) = \tau$ when executing endpoint $c$ for request $r$ with oracle $O$ yields $\tau$, and $resp(\tau) = \rho$ as the inbound response of trace $\tau$.

***Synchronous and asynchronous execution.*** Our model supports both synchronous and asynchronous RPCs.

For a synchronous RPC, the program blocks until the oracle returns the outcome, then continues in the updated state. Since the RPC result is consumed at the RPC invocation state, $\sigma_i$ in the execution trace corresponds to both the RPC invocation and outcome consumption. The relevant error handling code surrounds the blocking RPC invocation.

An asynchronous RPC, on the other hand, is issued at one program point but its outcome is consumed at another—the *wait point* at which the spawning computation collects the outcome. In these cases, the state $\sigma_i$ is taken at the wait point, when the outcomefi rst becomes observable to the enclosing computation. The relevant error handling code surrounds the wait point. An outcome that is never collected cannot be propagated, so the endpoint is correctly fail-open on that call.

### 3.2 Counterfactual Causation

Fail-close is defined via *causal* reasoning [33]: we say that a specific outbound RPC returning *failure* "*caused*" the inbound endpoint to return *Error*. We use counterfactual ("but-for") reasoning applied to individual RPC events within a trace.

***Intervention.*** Given a trace $\tau = exec(c, r, O)$ and a position $i$, $Interv(O, i)$ returns a new oracle $O'$ where:

$$O'(j,\sigma,d) = \begin{cases} O(j,\sigma,d) & \text{if } j < i, \\ success & \text{if } j \geq i \end{cases}$$

For $j < i$, $O'$ agrees with $O$, ensuring the executions share the same prefix up to position $i-1$. For $j \geq i$, $O'$ returns *success* for all RPCs, regardless of which dependency $d$ is invoked.

The intervened execution $exec(c, r, O')$ may follow a different control path than the original trace $\tau$. For example, if position $i$ in $\tau$ was a failing primary RPC that triggered a fallback at position $j > i$, then in the intervened execution, position $i$ succeeds, and the fallback may not be invoked at all. As a result, the sequence of RPC events after $i$ may differ from $\tau$, but all RPC outcomes from position $i$ onward are *success* regardless.

***Definition D1 (Counterfactual causation).*** Let $O' = Interv(O, i)$. The $i$-th RPC event in a trace is a *cause* of that trace's *Error* response, written as $causes(i, \tau)$, if:

(D1.1) $o_i = failure$
(D1.2) $resp(exec(c, r, O)) = Error$;
(D1.3) $resp(exec(c, r, O')) = Ok$.

Conditions D1.1 and D1.2 capture that the $i$-th outbound RPC must have failed and that the inbound endpoint $c$ must have returned Error for request $r$. Condition D1.3 captures the counterfactual: if the $i$-th RPC (and all subsequent RPCs) had succeeded, would $c$ have returned *Ok*? The outer existential quantification over scenarios ($r$ and $O$) appears in Definition D2 below.

### 3.3 Fail-Close as a Behavioral Property

We define fail-close as a behavioral property on RPC events.

***Definition D2 (Fail-close dependency).*** Endpoint $c$ has a fail-close dependency on RPC $e_i$ in trace $\tau$ iff $\exists$ a request $r$ and oracle $O$ such that in $\tau = exec(c, r, O)$:

(D2.1) $causes(i, \tau)$ (intervening at $i$ makes it return *Ok*), and
(D2.2) $\forall j > i : o_j = success$ (all later RPCs in $\tau$ succeeded).

This definition requires the existence of a trace $\tau$ where the counterfactual holds (D2.1) and the $i$-th RPC event in $\tau$ is the last failing one (D2.2). It aims to balance modelfidelity and complexity to support reasoning about observed code patterns. It is intentionally biased toward fallback patterns (e.g., Listing 1), which we empirically determined to be most prevalent. While the model attributes a failure only to the last RPC in cases that require multiple RPCs to fail simultaneously, it supports iterative refinement:fi xing an attributed fail-close dependency can expose other fail-close dependencies.

***Applying the definition to the fallback example.*** Recall the fallback handler of Listing 1. An RPC event's outcome is *failure* when its call returns a non-nil error. Take an oracle $O$ under which both RPCs fail (i.e., the handler takes the fallback branch and returns at line 6); the resulting trace is $\tau = \langle e_1, e_2, Error \rangle$ with $d_1 =$ `rpcA`, $d_2 =$ `rpcB`, and $o_1 = o_2 = failure$.

- $e_2$ *is* fail-close. Condition D2.1 holds: under $Interv(O,2)$ the fallback succeeds and the handler returns `r2` with $\rho = Ok$, so $causes(2,\tau)$ holds. Condition D2.2 holds vacuously: no RPC event follows $e_2$.
- $e_1$ is *not* fail-close. Condition D2.1 holds: intervening at $e_1$ doesfl ip the inbound response to $Ok$, so $causes(1,\tau)$ holds. However, condition D2.2 fails: event $e_2$ also has outcome *failure*.

There exists no trace that witnesses a failing `rpcA` event not followed by an `rpcB` event, and whenever `rpcA` fails the handler issues `rpcB`. If the latter succeeds the handler returns *Ok*. The definition therefore blames only the fallback `rpcB`, which is the intuition of §3 made precise.

***Applying the definition to a production example.*** Listing 2, abridged from a service in our corpus, is more nuanced. The `ORG_FAMILY` arm assigns the error of `InSameFamily` to `err`, declared *outside* the loop; the `ORG_BUSINESS` arm only logs its error, and whatever is left in `err` at the exit becomes the endpoint's error (line 14). Consider a request with both org types that results in both RPC calls; the resulting traces are $\tau = \langle e_1, e_2, \rho \rangle$ with $d_1 =$ `InSameFamily` and $d_2 =$ `GetDelegations`.

- $e_1$ (`InSameFamily`) *is* fail-close. Take an oracle under which thefi rst call fails and the second succeeds ($o_1 = failure$, $o_2 = success$), giving $\rho = Error$, since `err` crosses the iteration boundary to line 14. Condition D2.1 holds: under $Interv(O,1)$ the handler returns $Ok$ at line 8, so $causes(1,\tau)$ holds. Condition D2.2 holds: the later event $e_2$ succeeds.
- $e_2$ (`GetDelegations`) is *not* fail-close. Take an oracle under which both calls fail ($o_1 = o_2 = failure$), giving $\rho = Error$ again, carried from $e_1$. Condition D2.2 holds vacuously: no RPC event follows $e_2$. Condition D2.1 fails: $Interv(O,2)$ leaves $e_1$ failed, so `err` is still non-nil at line 14, the response is still *Error*, and condition D1.3 is violated. No other trace does better: no branch reads the error that `GetDelegations` returns.

**Listing 2.** A production handler (abridged). Both RPCs sit in the same loop and reach the same exit, yet the endpoint is fail-close only on `InSameFamily`.

```
1  func IsChildEntity(req *Req) (*Resp, error) {
2    resp := &Resp{}
3    var err error   // declared outside the loop
4    for _, orgType := range req.OrgTypes {
5      switch orgType {
6      case ORG_FAMILY:
7        resp, err = InSameFamily(req) // RPC
8        if err == nil { return resp, nil }
9      case ORG_BUSINESS:
10       _, dmErr := GetDelegations(req) // RPC
11       if dmErr != nil { log.Warn(dmErr)}
12     }
13   }
14   if err != nil { return nil, internalErr(err) }
15   return resp, nil
16 }
```

### 3.4 Aggregation to service-level dependencies.

For reporting, we aggregate fail-close RPC events:

- Endpoint $c$ has a fail-close dependency on endpoint $d$ if $\exists$ RPC event $e_i$ where $e_i$ targets dependency $d$ and $c$ is fail-close on $e_i$ (per Definition D2).
- Service $S$ has a fail-close dependency on service $T$ if some inbound endpoint of $S$ has a fail-close dependency on some endpoint of $T$.

*Fail-close is a local property.* An important model feature is that the oracle abstracts every outbound call to an outcome. That is, $resp(exec(c, r, O))$ depends on $O$ only through the outcomes at $c$'s own RPC events. Therefore, whether a dependency $d$ is itself fail-close on some further downatream service cannot influence whether $c$ is fail-close on $d$. As a result, fail-close paths can be identified independently and in parallel, without a cross-service whole-program step.

Appendix A works the definition through further error-propagation shapes, including production Go and Java code (Appendices B, C) and patterns that fall outside it.

## 4 RPCShield Design

We now present RPCShield's language-agnostic design. §4.1 bridges the formal model to static analysis, §4.2 outlines our analysis pipeline, and §4.3–§4.6 describe the four phases of the analysis pipeline; §5 gives the Go and Java instantiations.

### 4.1 Instantiating the Model as Static Analysis

The behavioral model (Definitions D1 and D2) defines fail-close in terms of execution traces, oracles, and counterfactuals; static analysis has none of these. We show how program analysis techniques stand in for each: calling contexts for program states, and control- and data-flow analyses for the oracle.

We use the term *RPC site* to refer to the program point where the outbound RPC's outcome is consumed. It captures both synchronous and asynchronous cases. When a distinction is needed, we use precise terminology: *invocation point*

for where the RPC is issued, and *wait point* for where the outcome is consumed.

***System state ($\sigma_i$) ⇒ calling contexts.*** Static analysis does not have access to the runtime system state $\sigma_i$ at each RPC event $e_i$. However, it can construct all calling contexts (chains of function calls each rooted at the inbound API and ending with an outbound RPC) to over-approximate the system state under which an RPC will occur. The calling context serves two important purposes:

**Future modeling:** how does the activation stack handle success or failure responses returned from the RPC driving thefi nal outcome of the inbound API.

**Attribution:** attribute how the error bubbles up from the RPC through internal functions, making the reporting highly actionable. It helps distinguish between the same outbound RPC that may be gracefully handled along one path and unhandled along another.

A calling context is a coarser abstraction than a system state: all states that share a calling context (e.g., successive loop iterations) are conflated into one. The analysis therefore reasons about whether *some* state reaching that context is fail-close, which matches the existential quantification in Definition D2, and we have found this loss of precision to pose no practical hurdle in operationalizing RPCShield.

To derive calling contexts, we construct a whole-program call graph [6] rooted at inbound endpoints and extract all non-recursive reachable paths to outbound RPC sites (Phase 2, §4.4). Each extracted path $p = [f_0 \xrightarrow{cs_1} f_1 \xrightarrow{cs_2} \cdots \xrightarrow{cs_k} f_k]$, where $cs_i$ denotes the call site in $f_{i-1}$ that invokes $f_i$, represents a concrete calling context. This approach captures the *context-sensitive* nature of fail-closeness.

Asynchrony separates RPC invocation from RPC result consumption. Our static analysis synthesizes a single calling context that captures both. The *wait segment* runs from the inbound endpoint to the wait point in the enclosing function; the *outbound segment* runs from the spawned task's entry to the outbound RPC invocation point. Call graph enrichment (§5.1) splices the two, in effect attaching the outbound segment as a callee of the wait point, and the analysis traverses the result exactly as it would a synchronous call chain.

***Oracle (O) ⇒ control- and data-flow analysis.*** The model uses oracles to explore different failure scenarios: the witness oracle returns *failure* for the RPC under test, while the counterfactual oracle returns *success* for that RPC and all subsequent RPCs (the intervention of Definition D1, condition D1.3). Instead of materializing oracles and running the code, the static analysis performs a *what-if analysis* for each function $f$ on a call path:

1. **Witness scenario:** Assume the RPC site $s$ within $f$ returns `failure`, and all other subsequently reachable RPC sites return `success`. Compute reachable return statements via control-flow analysis.
2. **Check propagation:** Use data/control-flow analyses [22, 48] to determine whether any reachable return statement (or throw statement) produces an `Error` (or exceptional) outcome derived from $s$.
3. **Counterfactual scenario:** Repeat with $s$ returning `success` to verify that $f$ *can* return `Ok` when $s$ succeeds, confirming that $s$ causally controls the outcome.

The per-function outcomes compose via inter-procedural analysis: if every function on the call path meets these constraints, the model's conditions are satisfied. The model is existential: the existence of a path (witnessing trace) in the control-flow graph (CFG) of the function suffices.

***Fault model.*** Our model treats a dependency failure as an outcome observed at the call site and presumes that a failure of an outbound RPC manifests as an error there. This holds in our setting because Go and Java RPC libraries surface transport errors, remote errors, and timeouts as returned errors or thrown exceptions. A deployment whose clients signal failure in another way, for instance by returning a sentinel value that the caller must inspect, would need those call sites recognized as failure points before the analysis applies.

***Soundness and completeness.*** The static analysis over-approximates the true fail-close set: it considers all syntactically reachable call paths (including infeasible ones) and uses *may* analyses. For heap escapes, the system's precision depends on alias analysis, introducing false positives and false negatives. We measured how often this arises: of 4,290 unique RPC sites in the top 100 services, 15 (0.35%) had an error value escape the analysis this way, and for only 3 of them (0.07%) would the escape have changed a fail-open verdict to fail-close. Under standard assumptions (sound call graph, sound error-wrapper treatment, no heap-escaping errors), the reported set is a superset of the true set (Definition D2). False positives are rare (§6.2): manual review indicates a false positive rate of approximately 5%.

### 4.2 Design Overview

RPCShield analyzes each service independently. Given a service's source code and metadata, the analysis proceeds in four phases, shown in Figure 1: endpoint detection, call graph construction and path collection, errorfl ow analysis, and outbound resolution with reporting.

Listing 3 gives the core algorithm in pseudocode, which is path-oriented: it extracts call paths from inbound endpoints (IBs) to outbound RPC sites (OBs) and analyzes each path independently. Error summaries are computed per function and per call site, then cached for reuse. Since many paths share intermediate functions, a summary computed for one path is reused on all other paths that traverse the same function.

### 4.3 Phase 1: Endpoint Detection

In Phase 1, RPCShield analyzes a service's source code to identify (1) *inbound endpoints* (IBs), methods that implement

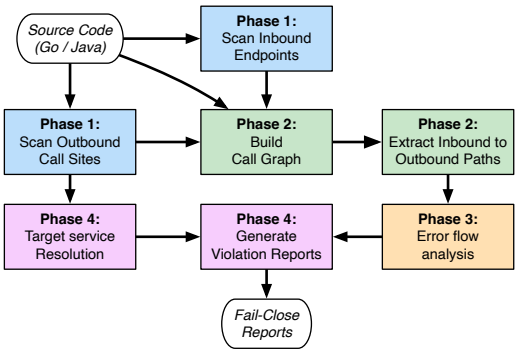



**Figure 1.** The RPCSHIELD analysis pipeline.

**Listing 3.** The RPCSHIELD analysis pseudocode.

```
1  IB, OB = detectEndpoints(service)        # Phase 1
2  CG = buildCallGraph(IB, OB)              # Phase 2
3  paths = extractPaths(CG, IB, OB)         # Phase 2
4  cache = {}
5  for each path p in paths:                # Phase 3
6    for each (f_i, callsite) in p:
7      if (f_i, callsite) not in cache:
8        cache[(f_i, callsite)] = errorSummary(f_i, callsite)
9    result = composeAlongPath(p, cache)
10   if result == FAIL_CLOSE:               # Phase 4
11     dest = resolveOutbound(p.OB)
12     report(p, dest)
```

server-side RPC handlers; and, (2) *outbound RPC sites* (OBs), invocations of client-side RPC stubs.

For IDL-based protocols (gRPC, Thrift) [13, 17], the IDL compiler generates typed server and client interfaces, so detection searches the type system for concrete types that satisfy these interfaces. For protocols without compile-time contracts (e.g., HTTP), the analysis falls back to AST or SSA [8] traversal to locate handler registrations and client method calls.

### 4.4 Phase 2: Call Graph Construction, Path Collection

Given the endpoints from Phase 1, during Phase 2, RPCSHIELD constructs a call graph rooted at inbound endpoints and extracts paths to reachable outbound RPCs.

***Base algorithm.*** The call graph is constructed using Rapid Type Analysis (RTA) [6, 34], which resolves virtual and interface calls based on types that are actually instantiated on at least one reachable path. RTA provides a good balance between precision and scalability over alternatives like CHA [10] or points-to analysis [5].

***Extensions.*** Standard RTA is insufficient for production codebases. We extended RTA in three ways:

- *Asynchronous controlfl ow.* Errors propagated through async constructs (futures, goroutine groups, task frameworks) do not follow the synchronous call stack and are invisible to standard call graph algorithms. The call graph must be enriched with edges that make the outbound call sites inside spawned tasks reachable, and that route each task's outcome to the *wait point* (§3.1) at which it is collected.
- *Dependency injection.* RPC stubs may be instantiated by a dependency injection framework rather than by application code, making them invisible to RTA's instantiation tracking. The analysis must recover these bindings from framework-specific configuration.
- *Lambda expressions and closures.* Higher-order functions that accept lambdas or closures create indirect call edges that RTA does not resolve natively.

Our Go and Java implementations address these patterns differently (§5).

***Pruning and path collection.*** The call graph is scoped to service code, treating calls to configurable, designated third-party libraries as terminals. After pruning, all paths from each inbound endpoint to each reachable outbound call site are collected. Path enumeration can occasionally lead to combinatorial explosion, which RPCSHIELD contains with two techniques. First, itfi ngerprints each discovered call path independently of the order of its functions, by xoring the function pointers along the path; paths sharing afingerprint are examined for permutations of the same function set, and afi xed ordering is imposed over the permuting functions so that only one arrangement is explored. Second, a limit on the branching factor bounds the enumeration.

### 4.5 Phase 3: Error Flow Analysis

For each call path extracted in Phase 2, in Phase 3 RPCSHIELD determines whether an error from the outbound RPC site can propagate to the inbound endpoint (lines 5–9 of Listing 3).

Our analysis boundary is the inbound RPC handler function. We treat the generated server-side entry point as the root of a call path, assuming that the RPC framework delivers errors and exceptions without custom transformation (i.e., no application-specific middleware alters error semantics).

***Per-function error summaries.*** For each function $f$ on a call path and each call site $s$ within $f$, the analysis computes a summary: if $s$ fails (D1.1), can an error from $s$ propagate to an error exit of $f$? The summary operationalizes the counterfactual definition (§3.2) by checking four properties (of which 1–3 correspond to D1.2 and 4 corresponds to D1.3):

1. *Reachability:* at least one return statement is reachable when $s$'s error occurs.
2. *Non-suppression (Witness):* on that return path, the returned error has not been forced to a success value.
3. *Attributable:* the returned error is traceable to $s$, either through datafl ow (the error value is derived from $s$) or control dependence [12] (the return is reachable because a condition on $s$'s error took the error branch).
4. *Causation:* when $s$ succeeds, a non-error return is reachable. This confirms $s$ controls the error outcome; without it, the function would fail regardless.

If all four hold, $f$ is fail-close with respect to $s$; otherwise it is fail-open. Summaries are keyed by (function, call site) and cached (Listing 3, lines 7–8). The concrete analysis technique differs by language (§5).

***Inter-procedural composition.*** Given per-function summaries for every function on a call path $p = [f_0, f_1, ..., f_k]$, the analysis composes them sequentially (line 9 of Listing 3). Starting from the outbound RPC site in $f_k$, it propagates the error through each function toward the inbound endpoint $f_0$. If the error reaches $f_0$'s error exit, the path is classified as fail-close. If it is handled at any intermediate function, the path is fail-open. Recursion is handled by ignoring repeated occurrences of a function along a path, so composition terminates at thefirst cycle. This is sound for the summaries we observed, which are invariant to calling depth: whether an error propagates through a function from a given call site does not depend on how many activations of that function are already on the stack.

### 4.6 Phase 4: Outbound Resolution and Reporting

In thefi nal phase, RPCShield maps outbound RPC sites to their target services and tiers and generates violation reports.

***Outbound-to-service mapping.*** In modern microservice codebases, the binding between an RPC stub and its destination has migrated out of source code and into configuration: YAMLfi les, dependency-injection wiring, environment-specific overrides, and build-system metadata. Resolving an RPC site to a deployed service name requires reasoning about code and configuration jointly.

Our primary strategy exploits type information in IDL-generated code. For gRPC and Thrift, the IDL compiler generates client stubs with canonical service interface names. We build a mapping chain: stub type → service interface → build target → deployed service. This mapping is derived from types rather than source-code patterns for stability across dependencies and code changes. When type-based resolution is ambiguous or unavailable (e.g., HTTP), we fall back to secondary resolvers: a configuration-based resolver that parses YAML service definitions, and a pattern-based resolver that matches class names and string constants.

***Report generation.*** RPCShield generates one report per inbound-to-outbound dependency showing all fail-close paths asfl amegraphs [16], with per-function classifications and source locations. Cross-tier violations arefl agged for remediation, aggregated per service for tracking and ticketing.

## 5 Implementation and Deployment

We implemented RPCShield for Go and Java, the two most widely used microservice languages at Uber. While the overall approach is identical, implementation details differ based on language features and error-handling idioms.

Go's idiomatic error handling returns an `error` object as the last return value from client RPCs, propagates it through internal function calls, and returns it from the inbound RPC handler: `error=nil` signals success, `error != nil` signals failure. This explicit return-value mechanism makes error propagation visible in the code. Java, in contrast, uses exceptions that propagate implicitly through the call stack until caught. Both error-handling models are amenable to static analysis; Go tracks error values through datafl ow, while Java tracks exception types through try/catch blocks.

### 5.1 Go Implementation

***Endpoint detection.*** For IDL-based protocols (gRPC, Thrift), the analysis uses Go's structural type system tofind concrete structs that satisfy each generated server interface. For transports that wire interfaces via constructor calls rather than static assignments, the analysis recovers the binding from the constructor's type signature. HTTP endpoints lack compile-time contracts, so the analysis tracks routers from creation to handler-registration sites in SSA, unwrapping middleware decorators as needed.

For outbound detection, calls are classified by their receiver type. Generated client types are distinguished by code-generation naming and package-path conventions. A multi-stage pipeline maps generated names to deployed services via dependency-injection bindings, deployment configuration, and name matching.

***Call graph construction.*** The RTA call graph is built over the Go SSA intermediate representation. Go's structural subtyping makes the standard library RTA [43] unsuitable for services with many types. We optimized RTA's work complexity and parallelized its execution, reducing runtime from ~40 minutes to under 15 seconds for large services [34].

*Handling asynchronous RPCs via graph enrichment.* Go's goroutines introduce asynchronous execution that complicates error-flow analysis: errors propagate from spawned tasks back to callers only at wait points (e.g., `Wait()` calls or channel receives). RPCShield handles three concurrency patterns in our codebase—workflow frameworks (`cff.Flow` [47]), explicit coordination (`errgroup`), and raw `go` statements—through call graph enrichment. The key insight is that asynchronous patterns with explicit wait semantics are equivalent to synchronous calls followed by error consumption. We leverage this by adding synthetic edges that deliver each task's outcome to its wait point—call edges anchored at the wait point where the outcome is a return value, and modeled transfer edges where it travels through a channel or shared variable—so that subsequent analysis phases can traverse async errorfl ow as ordinary call paths. This enrichment uses call graph traversal to discover reachable closures and data-flow analysis to match each closure with its wait point. Appendix D details the graph enrichment for each pattern.

*Path construction.* From the pruned graph, the analysis builds a per-inbound *Call Context Tree* (CCT) [4], a compact representation of all unique paths. The analysis restricts

path discovery to error-returning functions. Extended Sparse Conditional Constant Propagation (SCCP) [48], which includes type propagation, is applied along each candidate path to determine basic-block reachability, and outbound RPCs in provably unreachable blocks are discarded. Paths are partitioned into synchronous and goroutine paths based on go-statement edges.

***Error flow analysis.*** For each function and call site, the Go analysis checks the four properties from §4.5 using SCCP extended with assertion propagation [37]. A forward pass over SSA tracks nilness (nil, non-nil, or unknown), error sources, and per-block assertion facts. The analysis maps nil errors to success and non-nil errors to failure, matching the model's *Ok* and *Error* responses. To implement the counterfactual condition (§3.2), all other reachable RPC sites are assumed to return nil (success). Only reachable CFG blocks participate. Properties (1) and (2) check whether a maybe-non-nil error can reach a function exit point, and property (3) checks that the error arriving there is traceable to the call site; property (4) requires a second SCCP pass with the call site's error forced to nil, verifying that a maybe-nil error (success) can reach the exit. For freshly created errors (not data-derived from the RPC), attribution uses control-dependence traversal [12] tofi nd the innermost condition checking an RPC-returned error; that RPC is attributed as the cause.

### 5.2 Java Implementation

***Endpoint detection.*** Endpoint detection runs as a `javac` compiler plugin [1] that hooks into the build process. The plugin implements AST scanners that identify inbound endpoints by matching method declarations against gRPC service interface implementations, and outbound RPC sites by matching invocations against gRPC stub patterns. Running as a compiler plugin requires no separate build step, and the analysis sees exactly the code the compiler processes.

***Call graph construction.*** The Java call graph is built on ASM, a Java bytecode analysis framework [32]. The custom RTA call graph algorithm includes three key components:

*Lambda expressions and functional interfaces.* This extension scans each reachable method body for lambda expressions and invocations. It identifies generated lambda methods and adds them to the call graph worklist. For calls through functional interfaces, a mapping from interface call site to type-compatible lambdas is maintained. A post-processing step adds edges by connecting interface call sites to type-compatible and reachable lambda definitions.

*Asynchronous patterns.* APIs like `CompletableFuture` (CF) compose asynchronous operations into a DAG whose error flow is invisible to standard call graphs. Our extension identifies calls to CF factory methods (`supplyAsync`, `thenApply`, etc.), computes error summaries for these methods, and resolves continuations to their lambda implementations. Next, the analysis constructs a CF dependency graph, which is a directed graph of CF operations within a method. It uses this graph to track how exceptionsfl ow through the async DAG. In model terms (§3.1), this graph identifies the wait point at which each asynchronous outcome is consumed, playing the same role as call graph enrichment in Go.

*Dependency injection (DI).* RPC stubs may be instantiated by Spring's DI framework via `@Bean` methods in `@Configuration` classes, making them invisible to RTA. Our extension scans for configuration classes, resolves bean methods, and adds their return types to the set of instantiated classes. It also resolves `@Qualifier` annotations and constructor parameter dependencies.

***Error flow analysis.*** The Java analysis tracks exception propagation. For each function and call site, it computes a summary that maps exception typesfl owing in from the call site to exception typesfl owing out of the method. If the outgoing set is empty, the function is fail-open. The summary records how try/catch blocks transform exceptions: a catch of type `E` that re-throws type `F` is recorded as a transformation from `E` to `F`; a catch block that suppresses an exception removes it from the outgoing set.

For `CompletableFuture` DAGs, the analysis uses the CF dependency graph from Phase 2 to determine whether exceptions reach the DAG's terminal value. Exception handler methods, like `exceptionally`, that suppress exceptions induce further analysis of their lambda parameters.

### 5.3 Deployment

RPCShield runs as a recurring analysis pipeline across the Go and Java monorepos, targeting all T0–T2 services. Fail-close dependencies on lower-tier services arefi led as tickets assigned to the owning team of the inbound (calling) service. Each ticket includes the inbound and outbound services, their tiers, and the complete call path with per-function fail-close/fail-open annotations and links to the relevant source lines. Developers have reported that this call-path detail is the most useful part of the report.

We developed RPCShield in stages, and coverage and accuracy improved as we added additional features. During these steps, we engaged experts to evaluate the violations found. For example, we used the false positives identified during this process to prioritize new features.

## 6 Evaluation

We evaluate RPCShield along three dimensions. First, we measure the tool's intrinsic efficacy: how accurately and efficiently it detects fail-close paths (§6.2). Second, we assess its extrinsic impact on the failover safety use case (§6.3). Third, we present usability and adoption data (§6.4).

The Go version of RPCShield was developedfi rst, since the majority of our services are in Go. It has been in production for about two years. The Java version is relatively recent and

**Table 2.** Corpus statistics for 2,393 Go and 1,551 Java services.

| | Go | | Java | |
|---|---|---|---|---|
| | P50 | P95 | P50 | P95 |
| LOC | 2.8M | 35.4M | 514.6K | 875.5K |
| Packages | 1.0K | 3.8K | 964 | 1.6K |
| Files | 3.8K | 12.9K | 4.2K | 7.4K |
| Functions | 133.1K | 1.6M | 26.5K | 46.4K |

**Table 3.** Inbound / outbound endpoint statistics (by call site).

| | | Go | | | Java | | |
|---|---|---|---|---|---|---|---|
| Endpoints | Protocol | P50 | P95 | Total | P50 | P95 | Total |
| Inbound | gRPC | 4 | 50 | 28.2K | 16 | 50 | 39.2K |
| | HTTP | 0 | 0 | 493 | 2 | 6 | 8.5K |
| | Thrift | 0 | 39 | 16.4K | 0 | 35 | 14.0K |
| Outbound | gRPC | 32 | 187 | 203.4K | 179 | 830 | 426.2K |
| | HTTP | 40 | 58 | 132.4K | 33 | 50 | 65.1K |
| | Thrift | 31 | 128 | 137.6K | 143 | 1.9K | 801.8K |

has been in production for about six months. For that reason, most of our evaluation focuses on data from Go services.

### 6.1 Methodology and Experimental Setup

Uber's monorepos collectively host over 6K services (4,618 Go and 1,577 Java). We exclude stateful, storage services since they follow a different and stronger reliability policy and exempt from service preemption during failover; our analysis applies unchanged if the policy is extended to cover them. We also exclude services without an analyzable build target. This leaves a corpus of 3,944 services (2,393 Go and 1,551 Java), which we use for the intrinsic evaluation (§6.2). For the extrinsic evaluation (§6.3), we further restrict this corpus to services in tiers T0–T2—the tiers subject to the cross-tier fail-close policy (§2.2). This benchmark comprises 1,399 services (1,123 Go and 276 Java). We ran all evaluations on a shared cluster of AMD EPYC 7B13 (Milan) machines, where a CI job is restricted to 80 of the 128 SMT cores [45]. Each machine has 312 GB of DRAM and a 6 TB SSD, and runs Linux 6.1.87. Evaluations use Go-1.26.1 and Java 17.

### 6.2 Intrinsic Evaluation: Analysis Efficacy

This subsection evaluates the tool itself, independent of any particular tier-based fail-close policy.

**Corpus statistics.** Table 2 characterizes the analyzed Go and Java corpora. Go services are substantially larger (median 2.8M LOC, reflecting monorepo-wide transitive dependencies in the build). Call graph sizes differ by orders of magnitude as a result: the P95 Go service has 1.2M nodes and 209.7M edges, against 31.9K nodes and 355.1K edges for Java.

**Endpoint detection.** Table 3 breaks down the inbounds and outbounds detected in the Go and Java corpora by protocol. Go spreads its calls across all three protocols, with gRPC and Thrift roughly equally dominant and HTTP close behind on the outbound side; Java concentrates on Thrift for outbound calls and gRPC for inbound handlers. HTTP is the smallest share in both languages, but supporting it is necessary for full coverage. In absolute terms, Java services carry the larger endpoint counts—1.29M outbound call sites against Go's 473.4K—even though the Go corpus is far larger in code size.

**Call path statistics.** Figure 2(a,b) shows statistics for call paths in Go and Java. The Go paths (left plot) include sync and async paths, while the Java paths (right plot) only show sync paths[1]. Sync fail-close paths are shorter than fail-open ones in both languages: a median of 13 vs. 15 calls in Go, and 7 vs. 10 in Java.

Every fail-open path must handle the error somewhere, and thefi gure also plots the *depth* at which it is resolved or swallowed (green lines). Handling clusters within a few call hops from RPC call site site: half of all sync fail-open paths resolve within 3 calls in Go and 4 in Java. Thus, developers typically handle RPC errors close to where they arise.

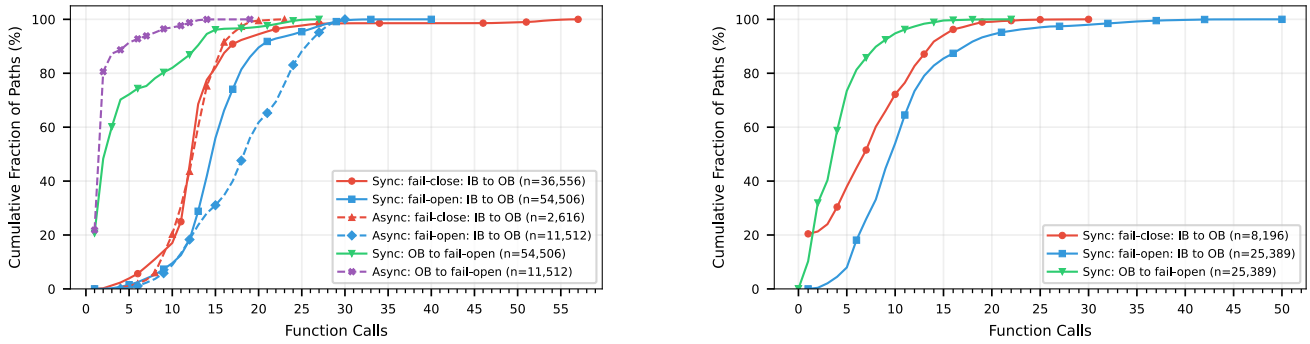
In Go, async fail-open paths are longer than sync ones (median 19 vs. 15), whereas async fail-close paths match their sync counterparts (both median 13). Despite the longer paths, async errors are suppressed at a similar or slightly shallower depth (median 2 vs. 3).

Across all Go services there are 66,018 fail-open and 39,172 fail-close paths. Grouped by unique (inbound, outbound) tuple, 45.3% of tuples carry *only* fail-close paths, 45.7% carry *only* fail-open paths, and the remaining 9.0% carry both. That last group is why the verdict must be per path: the same pair of endpoints can be fail-close on one path through the code and fail-open on another. Appendix E takes the Go intra-procedural analysis apart condition by condition, reporting how many paths each check in Definition D2 retains.

**Path exploration effectiveness.** Path enumeration (§4.4) employs permutation detection to prevent combinatorial explosion. Across the analyzed corpus, only 0.83% of inbound-to-outbound dependencies hit the permutation threshold. Rare as it is, the mechanism is necessary: without it, sporadic path explosions exhaust memory and kill the analysis. Among candidate inbound-to-outbound paths, our extended inter-procedural constant propagation prunes 88% as infeasible (2.33M of 2.65M). Pruning is concentrated in the services that carry infeasible conditional branches—roughly a third of the corpus—where it eliminates a median of 99% of candidate paths, enabling scalable analysis.

**Performance.** Figures 3a and 3b plot RPCShield's execution time against service size for Go and Java. For Go, service building and call path collection are the two dominant phases, and which one leads depends on the size of the service. Up to the

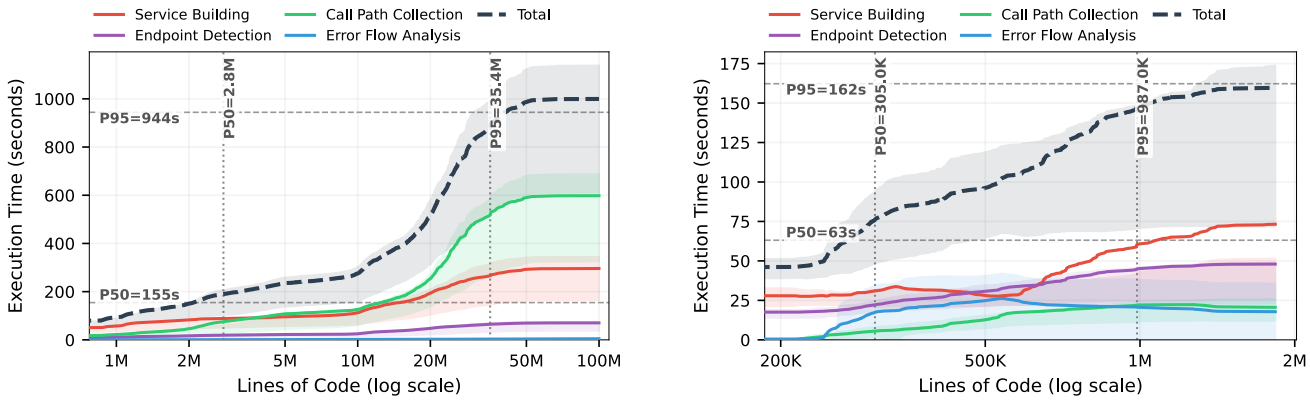
[1]Async Java path analysis has not been fully productionized.




**(a)** Go call path lengths

**(b)** Java call path lengths

**Figure 2.** CDF of call path lengths – inbound (IB) to outbound (OB) – and distance at which fail-open is handled from OB for Go and Java services. Solid lines denote synchronous paths; dashed lines denote asynchronous paths.




**(a)** Go analysis execution time.

**(b)** Java analysis execution time.

**Figure 3.** Per-phase execution time vs. service size for the corpus of Go and Java services. Lines show rolling means; shaded bands show IQR (P25–P75). Dashed lines mark P50/P95 for service size (vertical) and total time (horizontal).

median size of 2.8M lines, building is more expensive; at about 20M lines (a small fraction of services), call path collection becomes noticeably more expensive as the number of paths to explore grows non-linearly. Constructing the call graph is not a bottleneck: our parallelized RTA (§5.1) builds it in seconds even for the largest services, and the time in this phase goes to enumerating paths over it. Java services are far smaller than Go services and analyses have a different performance profile: service building and endpoint detection are the two prominent costs, while call path collection stays well contained. Overall, analysis time per service at p95 is about 16 minutes on the Go side and about three minutes on the Java side; at p50, Go completes in about 2.5 minutes and Java in about one minute.

### 6.3 Extrinsic Evaluation: Failover Safety

This subsection evaluates RPCShield with the cross-tier fail-close policy from §2.2: critical services (T0–T2) must not have fail-close dependencies on lower-tier services.

**Violation volume and resolution.** Table 4 shows the 3,842 fail-close violations that RPCShield found during a 20-month deployment period, broken down by caller and callee tier (as defined in Table 1). Most violations (2,545, 66%) originate in T1 callers, and the single largest group is T1 services depending on T2 services (1,701 violations, 44% of the total).

Service owners carried 2,540 (66%) of these to a terminal ticket state by one of four strategies (Table 5): *fix in code*, adding error handling or a fallback so the caller degrades gracefully; *re-tier the callee*, promoting a genuinely critical callee so the call is no longer cross-tier; *defer*, acknowledging the violation but scheduling thefi x for a later sprint (this is a routine triage offi nite engineering effort against competing roadmap priorities, not a dismissed report); and *close without documented resolution*, where no per-ticketfix can be attributed. We count only the 1,359 violations in the first two categories asfi xed. The remaining reports are still open, blocked on upstream dependencies, or classified as

**Table 4.** Cross-tier fail-close violations detected by RPCShield (January 2025 – August 2026). Cells marked "–" denote invalid pairs, as violations are only reported when the callee tier is strictly lower than the caller's.

| | Callee Tier | | | | | |
|---|---|---|---|---|---|---|
| **Caller** | T1 | T2 | T3 | T4 | T5 | **Total** |
| T0 | 0 | 9 | 35 | 1 | 23 | **68** |
| T1 | – | 1,701 | 623 | 13 | 208 | **2,545** |
| T2 | – | – | 1,122 | 5 | 102 | **1,229** |
| **Total** | | | | | | **3,842** |

**Table 5.** Remediation of reported fail-close violations.

| **Strategy** | **Violations** | **%** |
|---|---|---|
| Fixed in code | 421 | 16.57 |
| Fixed by re-tiering callee service | 938 | 36.93 |
| Deferred resolution† | 811 | 31.93 |
| Closed without clearfi x attribution‡ | 370 | 14.57 |
| **Total acknowledged or remedied** | **2,540** | |

† Acknowledged as genuine policy violations, but theirfi x is deferred. Common reasons include: the violating endpoint is non-critical (though hosted in a critical service), is slated for deprecation, or belongs to a service due to be rearchitected or migrated.

‡ Ambiguous attribution includes single code changes that resolved multiple tickets and resolutions via re-tiering that could not be conclusively attributed due to time delays and/or de-centralized teams.

acceptable risk by the service team. Median time to ticket resolution from ticket creation is 60 days.

**False positives.** Because our analysis team manually reviews every violation beforefi ling a ticket (§5.3), we have a direct measure of the false positive rate. Figure 4 plots this rate over time for the Go implementation. It illustrates how the Go implementation has improved its true positive rate over time, as new features were developed and rolled out.

Manual review of the last 200 detected Go violations indicates a false positive rate of less than 5%. For the true positivesfi led as tickets, no tickets were rejected as false positives by the owning service team. The primary sources of false positives are: (1) error handling in library code that the static analysis does not analyze; (2) paths through dead code that cannot execute in practice; and (3) imprecision in virtual call resolution that includes infeasible call targets.

**False negatives.** Manual review bounds precision, but not recall. To approximate recall, we compare against the runtime dependency analysis (§2.3), which infers fail-close dependencies from production RPC traffic and is independent of the static pipeline. Taking its output as the reference set, RPCShield fails to report 2.5% of the fail-close dependencies that the runtime analysis reports. This is an estimate, not a bound

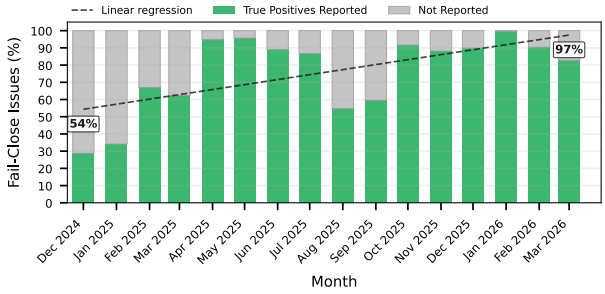



**Figure 4.** Monthly breakdown of fail-close issues detected by RPCShield for Go services (Dec 2024 – Mar 2026), showing the proportion of true positives reported. The dashed line shows a linear regression of the true positive rate. True positives increase from 54% to 97%, reflecting iterative improvements in RPCShield's analysis precision.

in either direction: the runtime method correlates caller errors with callee failures rather than establishing causation and cannot attribute a violation to a specific call site, so its non-causal reports inflate our apparent miss rate, while paths that production traffic never exercises hide real misses. As a second check, RPCShield would havefl agged the responsible code, as it stood at the time, for all 43 historical fail-close incidents of §7.

**Commit-time analysis.** RPCShield's success prompted a further shift left: when a developer submits a code change for review, RPCShield analyzes the affected services and reports any new violation as a warning. It surfaces about 10 violations a week, and of the 449 reported at commit time so far, 101 (22%) werefi xed before the change merged. These are violations that never reached deployment, and never needed a ticket.

### 6.4 Usability and Adoption

**Methodology.** To assess developers' experience with RPCShield, we surveyed engineers who had resolved fail-close tickets. We collected the 1,528 fail-close tickets created and resolved in 2024–2025. An automated feedback system sampled 380 unique ticket assignees and sent each a survey; 155 responded (41% response rate). Respondents had substantial experience with fail-close issues: 46% had addressed 2–5 violations since 2024, 28% had addressed exactly one, 9% had addressed 6–10, and 17% had addressed more than ten.

**Perceived severity.** We asked respondents to estimate the worst-case incident severity if the fail-close issue they fixed had not been addressed and the dependency had failed in production. 14% estimated the highest severity level (L5, a major customer-facing outage) and 44% estimated L4 (significant service degradation with direct customer impact)—58% in all. The remainder estimated L3 (26%, major internal impact) or L2 and below (16%, minor internal impact). The violations RPCShield detects are therefore not

theoretical: the engineers closest to the code judge a majority of them to be operationally dangerous.

**How developersfi x violations.** We asked respondents how they resolved the reported fail-close issue. The question allowed multiple selections, so thefi gures below are shares of respondents rather than a partition of them. 62% of respondents applied a codefi x; of those, 63% made the dependency call fail-open (e.g., adding error handling with a fallback), 19% deleted the offending code path entirely, and 15% re-architected the calling code. Among all respondents, 34% re-tiered the callee service to a higher priority so that it would no longer be preempted and 20% requested a policy exception, the two non-code resolutions.

These are respondent-levelfi gures with multiple selections allowed, so they neither partition nor line up with the violation-level counts of Table 5: an engineer who resolved several violations counts once here, the survey covers only tickets closed in 2024–2025, and it reached only engineers who had already resolved one—leaving Table 5's deferred and unattributed categories with no counterpart. Counted per violation rather than per engineer, re-tiering is the more common outcome.

Free-text responses revealed additional resolution patterns. Several respondents reportedfi xes that went beyond the immediate error-handling code: one team *"split services to not mix tier-2 and tier-3 functionality,"* addressing the root cause rather than patching the symptom. Another noted that *"other than making it fail-open we had to add a timeout to make sure the call didn't consume deadline budget making other subsequent calls to fail,"* illustrating how a RPCShield report can surface multiple latent issues. One respondentfl agged that the violation arose from a configuration error (*"staging metrics were wrongly emitted with production tags"*) rather than a code defect. Another pointed out a granularity mismatch: *"we fail-close on a request level but fail-open on the user flow level,"* indicating that the per-request model (§3) does not capture all the ways operators reason about failure impact.

**Time tofix.** We asked respondents how long it took them tofi x a fail-close violation, from root-causing to deployment. 72% reportedfi ve days or fewer (21% under one day, 26% in 1–2 days, 30% in 2–5 days), while 17% took 5–15 days and 13% took more than 15 days. Activefi x effort is thus far shorter than the 60-day median ticket lifetime of §6.3: roughly 55 of those days are backlog wait, with the ticket open until a team schedules it behind competing priorities. Thefi x is rarely the bottleneck; the queue is.

## 7 Discussion and Lessons

We highlight several observations from building and deploying RPCShield.

**Formalizing the property.** The formalization of fail-close took us three attempts. An early version quantified over service pairs and could not name a call site, and a later one blamed every failing RPC. Counterfactual reasoning is what made the third attempt work. Formalizing the property helped usfi nalize the tool's requirements, and it replaced two ad-hoc suppression rules in the Go analyzer. The model remains an idealization, assuming deterministic execution and errors that stay in returned values, and the implementation departs from it where the codebase demands (§4.1).

**Where the model's ordering rule does not apply.** Condition D2.2 of Definition D2 requires the RPC under test to be the *last* failure in the trace. That requirement is what lets the model blame the fallback rather than the primary (§3.3), but it presumes the failures are causally ordered, and one pattern violates that presumption. Consider a conjunctive check, in which the endpoint issues two RPCs, then conjoins their error checks and fails only if both fail. Our model would blame only the last of the failures but miss the other one, whose failure is necessary for thefi nal outcome. Because neither failure can be isolated as *the* cause, the definition falls back on program order to settle the blame. Appendix A works through this example. The implementation departs from the definition here by design: when the returned error is data-dependent on several RPC errors it attributes every contributor (§4.5), and it applies the ordering rule only in the fallback shape, where the return is control-dependent on a single RPC's error check.

**Per-servicefi ndings are not chained.** fail-close paths compose in principle: if $A$ is fail-close on an endpoint of $B$ that is itself fail-close on $C$, then a failure of $C$ can surface at $A$. Establishing such a chain accurately requires more than simply stitching the two reports, since each hop is witnessed by a particular request and code path, and the per-service analyses do not guarantee that the request $A$ issues to $B$ drives $B$ down its witnessing path. RPCShield therefore reports each hop separately. In practice, chaining is less of a problem than it may appear because propagation along a chain requires *every* hop to be fail-close, so resolving fail-close on any one hop breaks the chain.

**Actionable output determines adoption.** The most precise analysis is useless if developers cannot act on its reports. Early versions of the tool produced reports that identified a fail-close violation but did not include the full call path. Adoption was low. Once we added per-function, per-call-site annotations to the reports (including links to the relevant source lines in the code browser), developer engagement increased and ticket resolution times dropped.

**Incident reduction.** A retrospective of production incidents from 2020 to 2023 attributes 43 highest-severity, customer-facing outages to fail-close dependencies — roughly ten a year. In 2025, after more than a year of RPCShield deployment, no production incident was attributed to a cross-tier fail-close violation. This is correlation, not causation: the tiering policy and the re-tiering it prompted landed over the same period, alongside other reliability work. Still, a category that had recurred annually, at measurable

monetary and user-experience cost, has stopped — consistent with the latent dependencies having been removed.

**Other forms of criticality inversion.** A fail-close dependency is only one way in which a critical service can be harmed by a less critical one. A callee that stays available but misses its latency target can also push a caller past its SLO, and pressure can travel in the opposite direction, with a low-tier caller exhausting a high-tier callee's capacity or rate-limit quota. Both are criticality inversions that are invisible in the error-propagation structure RPCShield analyzes; detecting them would require reasoning about latency and request volume. The classification itself does transfer, however: a separate effort at Uber now uses RPCShield's fail-close/fail-open edge labels to enforce retry policies.

## 8 Related Work

**Cascading failure detection and prevention.** Tang et al. [40] study cross-system interaction failures at Google, Azure, and AWS,fi nding that unhandled error propagation across component boundaries is a leading cause of production incidents. This is precisely the class of bug RPCShield targets. Huang et al. [20] characterize gray failures that escape detection during normal operation, a property shared by latent fail-close dependencies. Zhai et al. [52] prevent correlated failures triggered by service updates, a complementary concern to the dependency-safety violations RPCShield detects in code. LDFI [3] uses lineage-driven fault injection tofi nd minimal failure scenarios in distributed systems; Jepsen [23] tests databases under partition and crash faults; CrashTuner [28] exposes crash-recovery bugs. These tools inject faults and observe behavior at the system level, while RPCShield analyzes source code statically. The approaches are complementary: fault injection validates behavior under real failures; static analysis identifies risks without requiring execution.

**Microservice dependency analysis.** Seer [15] and Sage [14] use machine learning over distributed traces to diagnose performance anomalies in microservice deployments. CRISP [54] performs critical-path analysis across microservice architectures to attribute latency. Lee et al. [24] present thefi rst large-scale empirical study of error propagation patterns in microservices, characterizing how errorsfl ow across service boundaries. RPCShield differs from this body of work in targeting fail-close violations statically, before deployment.

**Distributed-system checking and debugging.** Numerous tools analyze distributed systems [27, 41, 50, 51], checking system-wide behavior against a model of the deployment or against the deployment itself. Micans [53] proposes pointer analysis for microservices, but does not target fail-close violations. RPCShield decides its property from a single service's source, which is what allows it to scale to thousands of services.

**Error and exception handling analysis.** A line of static analysis work tracks error propagation in C programs [18, 36, 49], checking that error codes returned from functions are not incorrectly overwritten or ignored. In contrast, RPCShield flags errors that propagate from an RPC outbound. Other systems focus on inferring andfi xing error-handling bugs in C programs [11, 21, 26, 44]. Robillard and Murphy [35] and Sinha and Harrold [39] analyze exceptionfl ow in Java programs, and RPCShield's search for exceptions that escape a microservice's handler is a close relative. The difference is the question being decided: these analyses ask whether an error can reach a program point, whereas RPCShield asks whether the endpoint's failure is *caused* by the RPC's failure (§3.3).

**Hyperscale reliability.** CAPA [25] enforces a low-dependency requirement for Google's software-defined network. Bansal et al. [7] describe Uber's differentiated failover architecture that is built on RPCShield: eliminating cross-tier fail-close dependencies is what makes preemption of non-critical services safe, and RPCShield is the static analysis layer of the pipeline thatfi nds them. Defcon [30] and Phoenix [2] address runtime overload via feature degradation and container shedding, rather than the pre-deployment analysis RPCShield provides.

## 9 Conclusions

Error propagation across service boundaries causes cascading failures in microservice architectures. We presented a counterfactual formalization of the fail-close property that captures when a dependency failure causes a service to fail its own callers, and admits a static over-approximation through per-service error-flow analysis. We implemented this analysis in RPCShield for Go and Java, addressing the engineering challenges of async controlfl ow, dependency injection, lambda expressions, and multiple RPC protocols across two large monorepos. Deployed across over 6K microservices, RPCShield has detected over 3,800 fail-close violations with a 5% false positive rate. Service owners have remediated more than 1,300 of these violations, and a majority of surveyed developers believe theirfi x prevented a significant production incident.

In 2025, no production incident was attributed to a cross-tier fail-close violation—a class of failure that had recurred roughly ten times a year before RPCShield. The resulting code-level dependency safety assurance let Uber reclaim over 1M CPU cores by oversubscribing low-criticality services onto spare capacity, without sacrificing the availability of critical services [7]. Future work will extend RPCShield to other inter-service reliability and efficiency problems, including the latency and capacity inversions that its error-propagation view does not capture (§7).

## Acknowledgements

We would like to thank the following individuals who made important contributions to this work: Nimish Sheth, Kenneth Boegh, Srikanth Prodduturi, and Pouya Samadi.

## References


[1] 2014. Java Compiler Plugin API. https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/com/sun/source/util/Plugin.html.

[2] Kapil Agrawal and Sangeetha Abdu Jyothi. 2025. Cooperative Graceful Degradation in Containerized Clouds. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3669940.3707244

[3] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *ACM SIGMOD*. 331–346. doi:10.1145/2723372.2723711

[4] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *ACM PLDI*. 85–96. doi:10.1145/258915.258924

[5] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph. D. Dissertation. DIKU, University of Copenhagen. DIKU report 94/19.

[6] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *ACM OOPSLA*. 324–341. doi:10.1145/236337.236371

[7] Mayank Bansal, Milind Chabbi, Kenneth Bøgh, Srikanth Prodduturi, Kevin Xu, Amit Kumar, David Bell, Ranjib Dey, Yufei Ren, Sachin Sharma, Juan Marcano, Shriniket Kale, Subhav Pradhan, Ivan Beschastnikh, Miguel Covarrubias, Chien-Chih Liao, Sandeep Koushik Sheshadri, Wen Luo, Kai Song, Ashish Samant, Sahil Rihan, Nimish Sheth, Albert Greenberg, and Uday Kiran Medisetty. 2026. Uber's Failover Architecture: Reconciling Reliability and Efficiency in Hyperscale Microservice Infrastructure. In *USENIX NSDI*.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320

[9] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80. doi:10.1145/2408776.2408794

[10] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 77–101. doi:10.1007/3-540-49538-X_5

[11] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. doi:10.1145/3338906.3338960

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349. doi:10.1145/24039.24041

[13] Apache Software Foundation. 2026. Apache Thrift. https://thrift.apache.org/.

[14] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3445814.3446700

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3297858.3304004

[16] Brendan Gregg. 2017. Visualizing Performance with Flame Graphs. USENIX Association, Santa Clara, CA.

[17] gRPC Authors. 2026. gRPC: A High Performance, Open Source Universal RPC Framework. https://grpc.io/.

[18] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *USENIX Conference on File and Storage Technologies (FAST)*.

[19] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *International Conference on Distributed Computing Systems (ICDCS)*. doi:10.1109/ICDCS.2016.11

[20] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *ACM HotOS*. doi:10.1145/3102980.3103005

[21] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *USENIX Security Symposium*.

[22] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *ACM POPL*. 194–206. doi:10.1145/512927.512945

[23] Kyle Kingsbury. 2023. Jepsen: Distributed Systems Safety Analysis. https://jepsen.io/.

[24] I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. 2024. The Tale of Errors in Microservices. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3 (2024). doi:10.1145/3700436

[25] Bingzhe Liu, Colin Scott, Mukarram Tariq, Andrew Ferguson, Phillipa Gill, Richard Alimi, Omid Alipourfard, Deepak Arulkannan, Virginia Jean Beauregard, Patrick Conner, P. Brighten Godfrey, Xander Lin, Joon Ong, Mayur Patel, Amr Sabaa, Arjun Singh, Alex Smirnov, Manish Verma, Prerepa V Viswanadham, and Amin Vahdat. 2024. CAPA: An Architecture For Operating Cluster Networks With High Availability. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[26] Haoran Liu, Shanshan Li, Zhouyang Jia, Yuanliang Zhang, Linxiao Bai, Si Zheng, Xiaoguang Mao, and Xiangke Liao. 2025. Error Delayed Is Not Error Handled: Understanding and Fixing Propagated Error-Handling Bugs. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE114 (June 2025), 23 pages. doi:10.1145/3729384

[27] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems. In *USENIX NSDI*.

[28] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *ACM SOSP 2019*. doi:10.1145/3341301.3359645

[29] Juan Marcano, Ashish Samant, Kai Song, Lingchao Chen, Kaelan Mikowicz, Tim Smyth, Mengdie Zhang, Ali Zamani, Arturo Bravo Rovirosa, Sowjanya Puligadda, Srikanth Prodduturi, and Mayank Bansal. 2026. Scaling Mobile Chaos Testing with AI-Driven Test Execution. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2026)*. doi:10.1145/3786583.3786900

[30] Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In *USENIX OSDI*. 607–622.

[31] Michael T. Nygard. 2007. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

[32] OW2 Consortium. 2026. ASM: Java bytecode manipulation and analysis framework. https://asm.ow2.io/.

[33] Judea Pearl. 2009. *Causality: Models, Reasoning, and Inference* (2nd ed.). Cambridge University Press. doi:10.1017/CBO9780511803161

[34] Elton Pinto and Milind Chabbi. 2026. Beyond Nominality: Faster Rapid Type Analysis in the Presence of Structural Subtyping. In *Proceedings of the ACM on Programming Languages (OOPSLA)*. Association for Computing Machinery, New York, NY, USA.

[35] Martin P. Robillard and Gail C. Murphy. 2000. Designing Robust Java Programs with Exceptions. *ACM SIGSOFT Software Engineering Notes* 25, 6 (2000), 2–10. doi:10.1145/357474.355046

[36] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis forfi le systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1542476.1542506

[37] Alexander Sakharov. 1994. Propagation of Constants and Assertions. *ACM SIGPLAN Notices* 29, 3 (1994), 28–32. doi:10.1145/181587.903973

[38] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *ACM SOSP*. doi:10.1145/3600006.3613156

[39] Saurabh Sinha and Mary Jean Harrold. 2000. Analysis and Testing of Programs with Exception Handling Constructs. *IEEE Transactions on Software Engineering* 26, 9 (2000), 849–871. doi:10.1109/32.877846

[40] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. 2023. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *ACM EuroSys*. 433–451. doi:10.1145/3552326.3587448

[41] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *European Conference on Computer Systems (EuroSys)*. doi:10.1145/3627703.3650077

[42] The Go Authors. 2016. Package errgroup. https://pkg.go.dev/golang.org/x/sync/errgroup.

[43] The Go Authors. 2026. Package rta: Rapid Type Analysis for Go. https://pkg.go.dev/golang.org/x/tools/go/callgraph/rta. Part of `golang.org/x/tools`, v0.42.0.

[44] Yuchi Tian and Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *ACM ESEC/FSE*. doi:10.1145/3106237.3106300

[45] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. 392–403. doi:10.1145/223982.224449

[46] Uber Technologies. 2017. Package multierr: Combine Go Errors. https://pkg.go.dev/go.uber.org/multierr.

[47] Uber Technologies. 2021. CFF: Concurrency Flow Framework for Go. https://github.com/uber-go/cff.

[48] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM TOPLAS* 13, 2 (1991), 181–210. doi:10.1145/103135.103136

[49] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-Backed Program Analysis for Scalable Error Propagation. In *International Conference on Software Engineering (ICSE)*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). doi:10.1109/ICSE.2015.75

[50] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *USENIX NSDI*.

[51] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *USENIX NSDI*.

[52] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *USENIX NSDI*. 575–589.

[53] Teng Zhang, Yufei Liang, Ganlin Li, Tian Tan, Chang Xu, and Yue Li. 2025. Bridge the Islands: Pointer Analysis for Microservice Systems. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA023 (June 2025), 23 pages. doi:10.1145/3728896

[54] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *USENIX ATC*. 655–672.

***Note: appendices have not been peer-reviewed***

# A Worked Fail-Close Judgments

This appendix connects the behavioral definition of fail-close to the error-propagation patterns encountered in production services. Wefi rst apply the definition to three canonical causal patterns, then explain how RPCShield turns behavioral causation into actionable source-level attribution. Appendices B and C ground that reasoning in production Go and Java code. Throughout, an RPC event has outcome *failure* exactly when the call returns a non-nil error (Go) or throws an exception (Java). Note that appendices are not peer-reviewed.

## A.1 Canonical Patterns

**A.1.1 Reading the Judgments.** Each judgment follows the same four steps. We choose a *witness* execution in which the RPC of interest fails; identify the resulting endpoint response; apply the intervention of Definition D1, which makes that RPC and all subsequent RPCs succeed; and compare the two responses to obtain the *verdict*. The witness must also satisfy the last-failure condition of Definition D2: every RPC after the selected event succeeds.

**A.1.2 Simple Propagation.** Listing 4 shows a handler that makes one RPC and returns its error directly.

**Listing 4.** Direct propagation of an RPC error to the handler's return.

```
1 func handler(req *Req) (*Resp, error) {
2   resp, err := downstreamRPC(req)  // event i
3   if err != nil {
4     return nil, err
5   }
6   return resp, nil
7 }
```

**Witness.** Let `downstreamRPC` fail. The handler returns its non-nil error, producing $\tau = \langle e_i, Error \rangle$.

**Intervention.** Making $e_i$ succeed falsifies the guard and the handler returns *Ok*; no later event exists, so the last-failure condition holds vacuously.

**Verdict.** The endpoint is fail-close on `downstreamRPC`.

**A.1.3 Disjunctive Checks.** Listing 5 shows a handler that returns an error when *either* RPC fails.

**Listing 5.** A disjunctive check that returns an error when either RPC fails.

```
1 func handler(req *Req) (*Resp, error) {
2   r1, err1 := rpcA(req)  // event i
3   r2, err2 := rpcB(req)  // event j > i
4   if err1 != nil || err2 != nil {
5     return nil, ErrFailed
6   }
7   return merge(r1, r2), nil
8 }
```

Both calls are issued unconditionally, so every trace contains events $e_i$ and $e_j$.

**Case 1: `rpcA`.**

**Witness.** Choose $o_i = failure$ and $o_j = success$. Thefirst disjunct is true, so the handler returns *Error*; because the only later event succeeds, the last-failure condition also holds.

**Intervention.** Intervening at $i$ makes both $e_i$ and every subsequent event successful. Both errors are therefore nil, the disjunction is false, and the handler returns *Ok*.

**Verdict.** The endpoint is fail-close on `rpcA`.

**Case 2: `rpcB`.**

**Witness.** Choose $o_i = success$ and $o_j = failure$. The second disjunct is true, so the handler returns *Error*; because $e_j$ is thefi nal event, the last-failure condition holds vacuously.

**Intervention.** Intervening at $j$ preserves the successful prefix at $e_i$ and changes $e_j$ to success. Both errors are nil, the disjunction is false, and the handler returns *Ok*.

**Verdict.** The endpoint is fail-close on `rpcB`.

Thus each RPC can independently be the sole failure that changes the endpoint response.

**A.1.4 Conjunctive Checks.** Listing 6 shows a conjunctive check that returns an error only when *both* RPCs fail.

**Listing 6.** A conjunctive check that returns an error only when both RPCs fail.

```
1 func handler(req *Req) (*Resp, error) {
2   r1, err1 := rpcA(req)  // event i
3   r2, err2 := rpcB(req)  // event j > i
4   if err1 != nil && err2 != nil {
5     return nil, ErrBothFailed
6   }
7   return merge(r1, r2), nil
8 }
```

Both calls are unconditional.

**Case 1: `rpcA`.**

**Witness.** No valid witness exists. If $e_i$ and $e_j$ both fail, the endpoint returns *Error* but $e_i$ violates the last-failure condition because the later event $e_j$ also fails. If $e_j$ succeeds, the conjunction is false and the endpoint returns *Ok*.

**Intervention.** Not applicable.

**Verdict.** The endpoint is not fail-close on `rpcA` under Definition D2.

**Case 2: `rpcB`.**

**Witness.** Let both calls fail, producing $\tau = \langle e_i, e_j, Error \rangle$. The conjunction is true, and because $e_j$ is thefi nal event, the last-failure condition holds vacuously.

**Intervention.** Intervening at $j$ changes `err2` to nil while preserving the failed prefix at $e_i$. The conjunction becomes false and the handler returns *Ok*.

**Verdict.** The endpoint is fail-close on `rpcB`.

Although both failures are necessary, their program order is causally arbitrary; §A.2 explains how the static analysis handles and reports such combined failures.

## A.2 Realizing the Model in Static Analysis

Definition D2 is stated over concrete executions and a counterfactual intervention on an oracle. RPCShield realizes these conditions with a static analysis over each (function, call-site) pair. For the witness execution, the analysis assumes that the selected RPC fails and subsequently reachable RPCs

succeed, then checks whether an error exit is reachable. For the intervention, it assumes that the selected RPC succeeds and checks that a successful exit becomes reachable. Per-function summaries compose these local results along an inbound-to-outbound call path (§4.5).

This realization also requires implementation choices that have no direct syntactic counterpart in the model. In particular, the analysis must preserve causation when errors are transformed or combined, and recover causation from control dependence when a function returns a freshly constructed error. The following cases explain those choices.

**A.2.1 Data Dependence for Error Provenance.** When code wraps, converts, or aggregates an RPC error, the returned value is no longer identical to the value produced at the call site. RPCShield models common program constructs with standard data-flow transfer functions [22] that propagate both the abstract error value and its RPC provenance. Wrappers and converters preserve non-nilness and transfer the input provenance to the result. The transfer function for `multierr.Append` [46] is disjunctive: its result is non-nil if *either* input is non-nil, and its provenance is the union of the contributing inputs. Consequently, the analysis preserves every RPC that may contribute to the aggregate. This choice also determines what developers see in a report: although the conjunctive judgment for Listing 6 selects the later call as its formal last-failure witness, RPCShield reports both necessary contributors rather than treating their program order as causal priority (§7).

**A.2.2 Control Dependence for Fresh Errors.** When code constructs a fresh error, no data-dependence path connects that value to the RPC error. RPCShield therefore realizes causation by walking outward through control dependence until itfi nds the nearest RPC-derived guard (§4.5).

**Listing 7.** A fresh error attributed to the nested fallback RPC through control dependence.

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

In Listing 7, the returned `ErrNotFound` is not data-derived from `err1` or `err2`. The return statement is control-dependent on the nested condition `err2 != nil`. Walking the control-dependence chain outward from the return statement, the implementation stops at thefi rst RPC nil-check it meets, `err2 != nil`. Since `err2` originates from `rpcB`, the fresh error is attributed to `rpcB`, not `rpcA`. The same guard is inverted for the intervention analysis: when `rpcB` succeeds, `err2 != nil` becomes false and the fresh-error return is no longer forced.

**A.2.3 Skipping Non-RPC Conditions.** Listing 8 illustrates why attribution must look past intervening conditions that are unrelated to an RPC outcome.

**Listing 8.** Fresh-error attribution through nested RPC and non-RPC guards.

```
1 func handler(mode string) (*S, error) {
2   r, err := rpcA()
3   if err != nil {
4     if mode == "strict"{   // non-RPC condition
5       return nil, ErrStrictMode  // fresh error
6     }
7     return nil, ErrLenient
8   }
9   return r, nil
10 }
```

The `ErrStrictMode` return is control-dependent on two nested conditions: the innermost `mode =="strict"` (a string comparison, not derived from any RPC) and the enclosing `err != nil` (derived from `rpcA`). Stopping at the innermost condition would lose the connection to the RPC. The implementation instead skips the string comparison and continues outward to `err != nil`, attributing `ErrStrictMode` to `rpcA`. Thus only an RPC-derived guard can establish the source-level link required by the witness and intervention analyses.

# B Production Go Fail-Close Patterns

The following production cases are organized by the analysis capability they exercise: preserving provenance through error transformations, tracking deferred propagation through loops, and attributing compound or freshly constructed errors. Together they show how the canonical judgments of Appendix A survive the controlfl ow and error-handling idioms found in deployed Go services. We mark the RPC site of interest with a red right triangle ▶.

## B.1 Error Transformation Across Calls

Wrapping, converting, or re-encoding an error changes its representation but does not suppress the failure it represents. The next three cases exercise increasingly indirect transformations while preserving a data-dependence chain from the marked RPC to the returned error.

**B.1.1 Switch-Directed Wrapping.** In Listing 9, every arm selected by the YARPC status code—and the fall-through taken when no case matches—returns a typed wrapper derived from `err`. The branch structure changes the reported error category, but no branch suppresses the RPC failure. Consequently, a failing `g.billingAccountsClient.Create` reaches the caller as a non-nil error and witnesses fail-close. RPCShield correctly classifies this as fail-close.

```
// Create creates a new billing account
func (g *gateway) Create(ctx context.Context,
    request *CreateRequest) (*CreateResponse, error) {
  ...
▶  response, err := g.billingAccountsClient.Create(
    ctx,
    createRequest,
    tenancy.ApplyCallOptionWithDefault
        (ctx, g.config.DefaultTenancy),
  )
  if err != nil {
    logger.Errorw("Failed to create
        billing account", err, "request", request)

    switch yarpcerrors.FromError(err).Code() {
    case yarpcerrors.CodeAlreadyExists:
      return nil, errors.WrapWithType
          (errors.AlreadyExistsError, err).
        WithField("request", request)
    case yarpcerrors.CodeInvalidArgument:
      return nil, errors.
          WrapWithType(errors.InvalidParamError, err).
        WithField("request", request)
    case yarpcerrors.CodeDeadlineExceeded:
      return nil, errors
          .WrapWithType(errors.TimeoutError, err).
        WithField("request", request)
    }
    return nil,
        errors.WrapWithType(errors.GatewayError, err).
      WithField("request", request)
  }
  ...
}
```

**Listing 9.** Switch-directed wrapping of an RPC error.

**B.1.2 Error Conversion.** Listing 10 checks and returns `errCasted`, not the original RPC result `err`. The conversion through `ToIDLServiceError` nevertheless preserves provenance: on the witnessing path, a non-nil RPC error produces a non-nil IDL error that reaches the return. The endpoint is therefore fail-close despite the intervening representation change. RPCSHIELD correctly classifies this as fail-close.

```
func (handler SomeHandler) Evaluater(
  ctx thrift.Context,
  request *idl.Request,
) (_ *idl.EvaluationResponse, err error) {
  ...
▶ res, movementResults, err := handler.evaluation(ctx
    ,...)
  errCasted := cuErrors.ToIDLServiceError(err)
  if errCasted != nil {
    return nil, errCasted
  }
  ...
  return result, nil
}
```

**Listing 10.** Conversion to an IDL service error before propagation.

**B.1.3 Nested Wrapping.** Listing 11 carries the RPC error through `fmt.Sprintf` and then `status.Error`. Following both transformations connects the original non-nil error to the returned gRPC status, yielding the same fail-close verdict. RPCSHIELD correctly classifies this as fail-close.

```
...
▶ response, err := h.controller.Create(ctx, ...)
if err != nil {
  return nil, status.Error(codes.Internal
      , fmt.Sprintf("failed to create diff: %v", err))
}
...
```

**Listing 11.** Nested conversion through a formatted string and gRPC status.

### B.2 Deferred Propagation Through Loops

Loops separate the iteration that observes an RPC failure from the return that exposes it. The relevant question is whether some feasible execution retains an RPC-derived non-nil error until a function exit.

**B.2.1 First-Error Retention.** For Listing 12, choose an iteration where the result status denotes failure and the RPC-derived `err` is non-nil. If `firstError` is still nil, the guarded assignment stores that error. Later iterations preserve that value because the `firstError == nil` guard prevents replacement, and the function eventually returns it. The retained value supplies a witness even though processing continues after the failing iteration.

```
func (c *
    controller) processAllEntities(ctx context.Context
    , ...) ([]entity.EvaluationResult, bool, error) {

  ...
  for _, entity := range entities {
▶   entityResult, err := c.processEntityBackfill(
    thriftCtx, entity, ...)
    if entityResult
        .Status == constants.EntityStatusFailure {
      anyFailure = true
      if firstError == nil {
        firstError = err
      }
    }
    results = append(results, entityResult)
  }

  return results, anyFailure, firstError
}
```

**Listing 12.** Retention of thefi rst RPC error across loop iterations.

**B.2.2 Conditional Aggregation.** In Listing 13, each failed fetch contributes to `fetchErrors`, but the aggregate is exposed only if no successful result reaches `allRuns`. Each application of `multierr.Append` uses the same disjunctive transfer function, so `fetchErrors` becomes non-nil once any iteration contributes a non-nil error and accumulates the provenance of all such iterations. An all-fail execution leaves `allRuns` empty and `fetchErrors` non-nil, satisfying the return guard and witnessing fail-close. Executions containing a successful fetch may remain fail-open; the verdict records the existence of the cascading path.

```
1
2  func (c *controller) fetchAndGroup(ctx
       context.Context, ...) ([]*entity.Group, error) {
3    ...
4    var allRuns []runsWithURI
5    var fetchErrors error
6    for uri := range uriSet {
7      ...
8  ▶   resp, err := c.gateway.FetchTestRunsWithFilters(
       ctx, req)
9      if err != nil {
10       fetchErrors = multierr.Append
           (fetchErrors, fmt.Errorf("...", uri, err))
11       continue
12     }
13     allRuns = append(allRuns, ...)
14   }
15   ...
16   // Return err if we couldn't fetch any test
17   if len(allRuns) == 0 && fetchErrors != nil {
18     return nil, fetchErrors
19   }
20   return c.groupRunsByURI(allTestRuns), nil
21 }
```

**Listing 13.** Aggregated loop errors returned when every fetch fails.

### B.3 Compound and Control-Dependent Attribution

Thefi nal cases cannot be explained by a single direct value flow. One combines errors from two RPCs; the other returns a fresh error whose provenance exists only in the guard controlling the return.

**B.3.1 Conjunctive Guard with Disjunctive Aggregation.** Listing 14 contains two marked RPCs. Thefirst RPC can propagate independently through the early return, whereas the combining block is reached only when both RPC errors are non-nil.

**Case 1: `GetCurrentState`.**

**Witness.** Let `GetCurrentState` return a non-nil invalid-argument error. Thefi rst guard is true, so the function returns `rolloutErr` before invoking `ListRunningGates`. Because no later RPC event occurs, the last-failure condition holds vacuously.

**Intervention.** Make `GetCurrentState` succeed. The early guard becomes false; with the subsequently invoked `ListRunningGates` also succeeding, execution reaches the final nil-error return.

**Verdict.** The endpoint is fail-close on `GetCurrentState` through the early-return path.

**Case 2: `ListRunningGates`.**

**Witness.** Let `GetCurrentState` return a non-nil error that does not satisfy the early guard, and let `ListRunningGates` fail. Both operands of the conjunctive guard are then non-nil, so the function returns their aggregate. Because `ListRunningGates` is thefi nal RPC event, the last-failure condition holds vacuously.

**Intervention.** Preserve the failed `GetCurrentState` prefix but make `ListRunningGates` succeed. Then `cmiErr` is nil, the conjunctive guard is false, and execution reaches thefi nal nil-error return.

**Verdict.** The endpoint is fail-close on `ListRunningGates`.

Within the combining block, `multierr.Append` is modeled by a disjunctive transfer function: the aggregate is non-nil if either operand is non-nil, and its provenance is the union of both operands. The enclosing conjunctive guard establishes that both operands contribute on this path, so RPCShield retains both RPCs in the diagnostic as explained in §A.2.

```
1  func (m *manager) GetStatus(ctx context.Context
       , ...) (*pb.RolloutStatus, []*pb.Gate, error) {
2  ▶ rollout, rolloutErr := m.GetCurrentState(ctx, ...)
3
4    if rolloutErr
         != nil && (yarpcerrors.IsInvalidArgument
         (rolloutErr) || !rolloutID.IsFullyQualified()) {
5      return nil, nil, rolloutErr
6    }
7
8  ▶ cmiResp, cmiErr := m.Inspector.ListRunningGates(ctx,
       ...)
9    if cmiErr != nil && rolloutErr != nil {
10     multiErr := multierr.Append(rolloutErr, cmiErr)
11     return nil
           , nil, shared.WrapAsYarpcError(multiErr, ...)
12   }
13   ...
14   return rollout, cmiResp, nil
15 }
```

**Listing 14.** A conjunctive guard followed by disjunctive error aggregation.

**B.3.2 Control-Derived Error.** In Listing 15, the returned `NewInternalError` is not data-derived from `err`. Its attribution instead follows control dependence: `err != nil`, derived from the marked RPC, is the nearest RPC-based condition that can force the return. A witness lets `GetInfo` fail; in the corresponding intervention it succeeds and returns a non-nil `storedPm`, falsifying both disjuncts and permitting success. The marked RPC is therefore fail-close even though the returned error has fresh identity.

```
1 ...
2 ▶ storedPm, err := a.engine.GetInfo(ctx, id)
3 if err != nil || storedPm == nil {
4   return nil, errors.NewInternalError("..." )
5 }
6 ...
```

**Listing 15.** A fresh error controlled by an RPC-derived guard.

### B.4 Takeaways

Across these cases, error representation, return location, and object identity all change. The stable property is causal propagation: wrapping and conversion preserve data provenance, loop analysis identifies a feasible retained-error path, aggregation preserves every contributing error, and control dependence attributes fresh errors to the RPC guard that makes them reachable.

## C Production Java Fail-Close Patterns

The following production cases show how the canonical judgments of Appendix A apply to Java's exception semantics. They exercise two capabilities of the Java analysis: propagating an uncaught exception across multiple functions and preserving provenance when a caught exception is wrapped in a different type. As in the Go appendix, we mark each outbound RPC site of interest with a red right triangle ▶.

### C.1 Exception Propagation Across a Call Path

An exception need not be syntactically uncaught at the inbound endpoint to be fail-close. What matters is whether it reaches the endpoint's error response rather than being converted into a successful response.

```
// Inbound gRPC adapter
public void
      get(DocReq req, StreamObserver<DocResp> obs) {
  try {
    obs.onNext(aggregator.fetch(req));
    obs.onCompleted();
  } catch (Throwable e) {
    obs.onError(e);
  }
}

// Aggregator called by the adapter
public DocResp fetch(DocReq req) throws Throwable {
▶ return provider.fetch(req);
}
```

**Listing 16.** Propagation from the marked outbound RPC through the aggregator and inbound gRPC adapter.

Listing 16 shows the complete inbound-to-outbound call path. The aggregator does not catch an exception from `provider.fetch`. The inbound adapter does catch it, but `obs.onError` exposes the exception as an error response; this is propagation, not suppression.

**Witness.** Let the marked `provider.fetch` RPC throw an exception. The exception exits the aggregator, reaches the adapter's catch block, and is passed to `obs.onError`, producing an endpoint error. No later outbound RPC occurs, so the last-failure condition holds vacuously.

**Intervention.** Make the marked RPC succeed. The aggregator returns its response, after which the adapter invokes `obs.onNext` and `obs.onCompleted`; the error path is no longer taken.

**Verdict.** The endpoint is fail-close on `provider.fetch`. Because the caller is Tier 1 and the callee is Tier 3, this dependency also violates the tier policy.

#### C.1.1 Fail-Open Remediation.

Listing 17 shows the developer's remediation after RPCShield reported the violation.

```
public DocResp fetch(DocReq req) throws Throwable {
  try {
▶ return provider.fetch(req);
  } catch (Exception e) {
    return DocResp.getDefaultInstance();
  }
}
```

**Listing 17.** The developer's fail-open remediation using a default response.

For exceptions caught by this handler, the marked RPC can fail while the method returns a normal default value. The RPC failure therefore no longer forces an endpoint error, making the dependency fail-open on that path.

### C.2 Exception Type Transformation

Listing 18 illustrates why the presence of a catch block does not by itself imply suppression.

```
try {
  CacheBlockingStub client = newClient(host);
▶ ProfileResponse resp = client.getProfile(req);
  return resp;
} catch (StatusRuntimeException e) {
  throw new GatewayError(
      e.getMessage(), e, mapStatus(e.getStatus()));
} catch (Exception e) {
  throw new GatewayError(
      e.getMessage(), e, GatewayError.Type.UNKNOWN);
}
```

**Listing 18.** Exception wrapping at an outbound RPC site preserves fail-close behavior.

**Witness.** Let the marked gRPC call throw `StatusRuntimeException`. Thefi rst catch block constructs and throws `GatewayError`, so an exception still exits the function and reaches the endpoint's error path. The generic catch block behaves analogously for other `Exception` subtypes.

**Intervention.** Let client construction succeed and make the marked RPC return normally. Execution then returns `resp` from the try block without entering either catch block.

**Verdict.** The endpoint is fail-close on `getProfile` despite the change in exception type.

The Java analysis realizes this judgment with per-function exception summaries (§5.2). For a caught exception type $E$ that is re-thrown as $F$, the summary records the transformation $E \rightarrow F$ rather than removing $E$ from the outgoing exception set. The wrapping constructor preserves attribution through its cause argument, while the normal return under the intervention establishes causation.

### C.3 Takeaways

Java catch blocks have two distinct meanings for fail-close analysis. A catch that returns a normal fallback suppresses the RPC exception and can establish fail-open behavior; a catch that re-throws, even under a different type, preserves fail-close behavior. Across both Java patterns, RPCShield applies the same causal test used for Go: an RPC failure must reach an endpoint error, and making that RPC succeed must permit a successful response.

## D Go Concurrency and Call Graph Enrichment

This section explains the Go-specific call-graph enrichment that makes errors produced inside goroutines visible to the ordinary error-flow analysis (§4.5). Standard RTA records a goroutine launch as a spawn edge with no return semantics. RPCShield therefore identifies the asynchronous operation's *wait point* (§3.1)—the site where the enclosing function observes its outcome—and reconnects the task's outbound segment there.

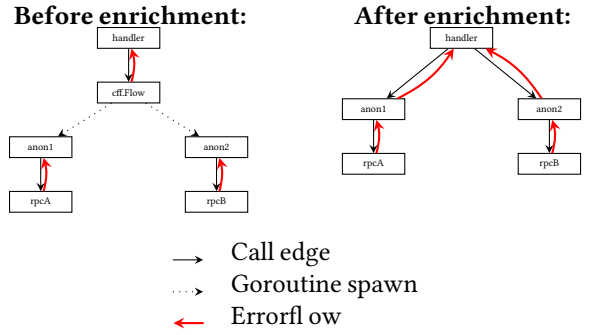



**Figure 5.** `cff.Flow` enrichment. Before enrichment, error flow exists within each closure but cannot cross a dotted spawn edge. Enrichment replaces the CFF scaffolding with call edges from the wait-point site to both closures; red arrows show the error paths that the existing summaries can then compose.

Two mechanisms cover the production idioms below. For `cff.Flow`, `cff.Parallel`, and `errgroup`, the outcome is returned by a framework wait point; enrichment replaces framework and registration plumbing with synthetic call edges from that wait-point site to the task closures. For raw go statements, the spawn edge remains for reachability and a modeled transfer edge carries the error from a channel send or shared-variable store to its corresponding receive or load. In the diagrams, black and dotted edges belong to the call graph, blue edges are modeled data transfers, and red arrows only illustrate errorfl ow that Phase 3 can trace after enrichment. As in the preceding Go appendix, red triangles mark outbound RPC sites in code listings.

### D.1 Return-Value Wait Points

When a framework call returns task errors, its call site can serve as an ordinary interprocedural summary point. The enrichment anchors task closures at that site and removes the intervening asynchronous scaffolding.

**D.1.1 `cff.Flow`: Multiple Tasks.** CFF [47] is a task-parallel library that generates glue code from an embedded Go DSL. Tasks registered through `cff.Task` execute concurrently, while the `cff.Flow` call synchronously returns a non-nil task error, if any.

```
1  func handler(ctx context.Context) (*Response, error) {
2    var a, b *Response
3    err := cff.Flow(ctx,
4      cff.Task(func() (*Response, error) {
5  ▶     return rpcA()
6      }),
7      cff.Task(func() (*Response, error) {
8  ▶     return rpcB()
9      }),
10     cff.Results(&a, &b),
11   )
12   return merge(a, b), err
13 }
```

**Listing 19.** A `cff.Flow` wait point returning an error from either RPC task.

Listing 19 registers two independent RPC tasks and binds both outputs through `cff.Results`. The `cff.Flow` invocation is the wait point because it returns the task error to `handler`. In Figure 5, generated CFF scaffolding is abstracted as the `cff.Flow` node. Enrichment removes that node and anchors direct call edges to both closures at the original `cff.Flow` call site, allowing existing function summaries to compose from each RPC back to the handler.

**Verdict.** RPCShield classifies both `rpcA` and `rpcB` as fail-close. For either RPC, choose a witness in which the selected RPC fails and the other succeeds. The relative execution order is immaterial: the successful task contributes no error, while the selected errorfl ows without suppression from its closure through the `cff.Flow` wait point to the handler's return. Under the intervention, the selected RPC also succeeds, so both tasks succeed, `cff.Flow` returns nil, and the handler can return successfully.

**D.1.2 `cff.Parallel`: Slice Iteration.** `cff.Parallel` is a variant of `cff.Flow` that applies the same lambda function to each element of a slice concurrently. It uses bounded parallelism and, by default, returns thefi rst non-nil invocation error.

```
1 func handler(ctx Context, ids []string) error {
2   return cff.Parallel(ctx,
3     cff.Slice(func(id string) error {
4 ▶     _, err := rpcFetchData(id)
5       return err
6     }, ids),
7   )
8 }
```

**Listing 20.** A `cff.Parallel` wait point returning an error from a slice task.

Listing 20 differs from Listing 19 only in closure multiplicity: one closure is reused across slice elements rather than registering distinct task closures. Figure 6 makes this one-closure rewrite explicit. As in Figure 5, enrichment removes CFF scaffolding and anchors the closure at the framework wait point.

**Verdict.** RPCShield classifies the marked `rpcFetchData` call site as fail-close. A witness lets one slice invocation fail while the remaining invocations succeed; its non-nil error flows from the reused closure through the `cff.Parallel` wait point to the handler. When all invocations succeed under the intervention, `cff.Parallel` returns nil.

**D.1.3 `errgroup`: Separate Registration and Wait.** The `errgroup` [42] pattern coordinates goroutines through explicit registration via `Go()` calls and a blocking `Wait()` call that serves as the wait point. Unlike CFF, where the framework call both spawns tasks and waits for completion, `errgroup` separates registration from the wait point.

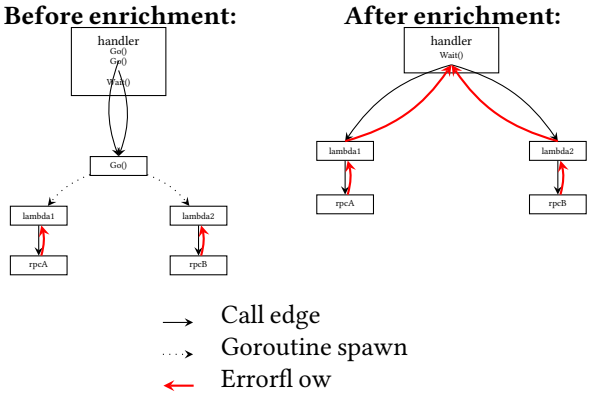



**Figure 7.** `errgroup` enrichment. Before enrichment, `Go` registration reaches the closures only through spawn edges. Enrichment removes registration plumbing and adds call edges from the `Wait` site to both closures, making their error summaries available at the wait point.

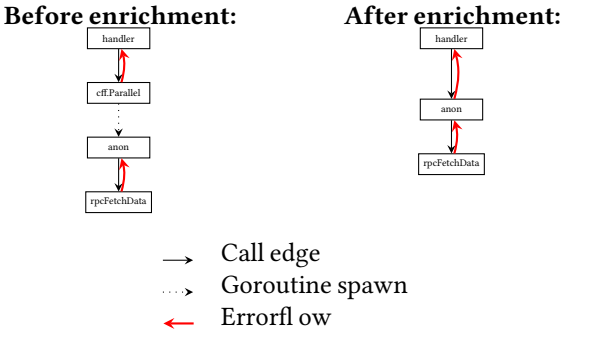



**Figure 6.** `cff.Parallel` enrichment. Before enrichment, repeated invocations share one closure reached through a spawn edge. Enrichment replaces the CFF node with a call edge from the `cff.Parallel` wait-point site to that closure, enabling its error summary to compose back to the handler.

```
1  func handler() (*Response, error) {
2    g := errgroup.Group{}
3    g.Go(func() error {
4  ▶   _, err := rpcA()
5      return err
6    })
7    g.Go(func() error {
8  ▶   _, err := rpcB()
9      return err
10   })
11   err := g.Wait()
12   return &Response{}, err
13 }
```

**Listing 21.** An `errgroup.Wait` point returning thefirst recorded task error.

In Listing 21, `Go` registers two closures, while `Wait` blocks until completion and returns thefi rst recorded non-nil error. Figure 7 shows the pattern-specific rewrite: registration plumbing is removed and both closures are anchored at the `Wait` call site. The analyzer can then compose each closure's summary into the value returned by `Wait`.

**Verdict.** RPCShield classifies both `rpcA` and `rpcB` as fail-close. For either RPC, a witness lets the selected task fail while the other succeeds, so `Wait` returns the selected error and the handler propagates it unchanged. When both tasks succeed under the intervention, `Wait` returns nil and the handler can succeed.

### D.2 Transfer-Edge Wait Points

Raw `go` statements do not expose a framework return value. Instead, the analysis retains the spawn edge and models the operation that transfers an error back to the enclosing function.

**D.2.1 Error Channels.** Raw `go` statements spawn goroutines that communicate errors through channels. The caller blocks on a channel receive operation, which is the wait point where the goroutine's error is collected.

```
1  func handler() (*Response, error) {
2    errChan := make(chan error, 1)
3    go func() {
4  ▶   _, err := rpcFetchData()
5      errChan <- err  // send error to channel
6    }()
7    err := <-errChan  // receive error from channel
8    return &Response{}, err
9  }
```

**Listing 22.** A channel receive serving as the wait point for a raw goroutine.

In Listing 22, the receive from `errChan` is the wait point. RTA reaches the closure through the dotted spawn edge but does not connect the closure's send to the handler's receive. Figure 8 shows the additional modeled transfer edge. Unlike the synthetic call edges used for framework wait points, this blue edge represents data movement from `errChan <- err` to `<-errChan`; the spawn edge remains unchanged.

**Verdict.** RPCShield classifies `rpcFetchData` as fail-close. In the witness, its non-nil errorfl ows from the closure through the channel send, across the modeled transfer edge to the receive wait point, and then to the handler's return without suppression. Under the intervention, the RPC sends nil, the receive yields nil, and the handler can return successfully.

The blue edge summarizes the channel's data transfer rather than adding a call or control-flow relation. It carries the provenance of the `rpcFetchData` error from the send to the wait-point receive, where the ordinary function summary can continue tracking it.

**D.2.2 Shared Error Variables.** Some production goroutines publish an error through a captured variable rather than a channel. The same abstraction applies: the spawn edge provides reachability, while a modeled transfer edge carries error provenance from the task's write to the synchronized read that serves as the wait point.

### D.3 Takeaways

Concurrency enrichment does not introduce a separate definition of fail-close or a separate error analysis. It restores the

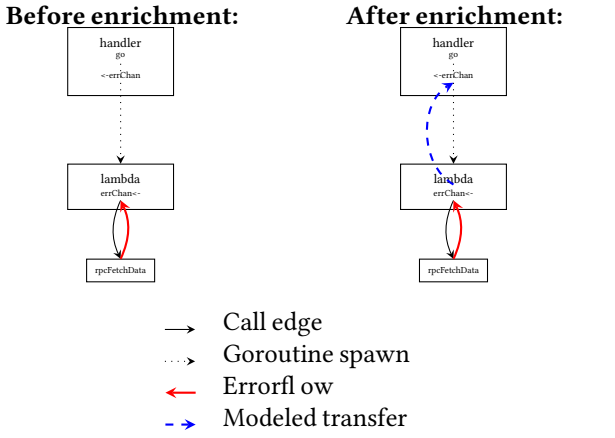



**Figure 8.** Channel enrichment for a raw go spawn. The spawn edge provides reachability in both graphs. Enrichment adds only the blue send-to-receive transfer edge, making the RPC-derived error available at the handler's wait point.

interprocedural path that ordinary call graphs omit: synthetic call edges handle framework-returned outcomes, while transfer edges handle values communicated through channels or shared state. Once an RPC-derived error reaches the wait point, the same witness, attribution, and intervention checks used for synchronous code determine the verdict. Thus, each example is fail-close only when the enclosing function returns the recovered task error without suppressing it.

# E Additional Evaluation Results

This appendix supplements the main evaluation with a condition-by-condition breakdown of the Go intra-procedural analysis sampled over a subset of 250 services.

## E.1 Intra-Procedural Classification Funnel

RPCShield analyzes 13,811 unique (function, call-site) pairs and caches each result for reuse across interprocedural paths. These pairs induce 15,274 intra-procedural paths from the selected call site to a function exit. Each path is tested against the witness, attribution, and intervention requirements; path results are then aggregated into one verdict per call site. Figure 9 summarizes this funnel.

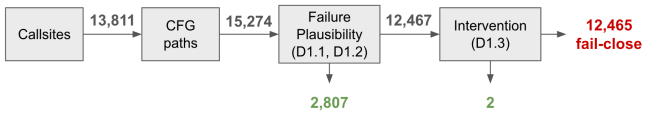



**Figure 9.** Go intra-procedural classification funnel. From 13,811 unique (function, call-site) pairs, RPCShield enumerates 15,274 exit paths. Non-suppression and attribution retain 12,467 candidate witnesses; intervention retains 12,465. Aggregating those paths yields 12,465 fail-close call sites.

**E.1.1 Witness: Non-Suppression.** Thefi rst check asks whether the path can carry a failed RPC outcome to an error exit ($\rho = Error$), as required by condition D1.2 of Definition D1. Of 15,274 control-flow paths analyzed, 13,586 (88.9%) preserved the error, while 1,688 (11.1%) suppressed it by reassigning to nil or wrapping in success responses. Suppressing paths advance no further in the funnel. Because one call site may have several paths, however, rejection of one path does not determine the call-site verdict.

**E.1.2 Attribution: Linking the Error to the RPC.** Returning an error is insufficient unless that error is attributable to the selected outbound call (§4.5). Of the 13,586 paths that preserved an error, 12,467 (91.8%) attributed it to the outbound call and 1,119 (8.2%) did not, the latter returning an error that originated elsewhere. Direct data dependence accounts for 11,810 passing paths. Control dependence accounts for another 657 paths by linking a freshly constructed return error to an RPC-derived guard.

**E.1.3 Counterfactual Intervention.** The inverse check asks whether changing the selected RPC outcome to *success* permits a non-error return, operationalizing condition D1.3 of Definition D1. Of the 12,467 paths that passed the witness and attribution checks, 12,465 passed the inverse check, confirming the outbound call controls the outcome. Only 2 paths failed because their functions returned errors independently of the selected RPC outcome.

**E.1.4 From Paths to Call-Site Verdicts.** A path rejected by any check is not a fail-close witness; it is not, by itself, evidence that the call site is fail-open. A call site is fail-close if at least one constituent path satisfies all conditions in Definition D2. Combining all checks yields 12,465 (90.3%) fail-close call sites and 1,323 (9.6%) fail-open call sites. The remaining 23 call sites receive no verdict. Within the analyzed RPC call sites, the 90.3% fail-close rate shows that unsuppressed error propagation is the dominant behavior.

# F Commit-Time Detection Over Time

The deployment results in §6.3 report aggregate commit-time outcomes. Figure 10 adds the temporal view, showing that new fail-close issues are detected throughout the observation period rather than in a single rollout spike.

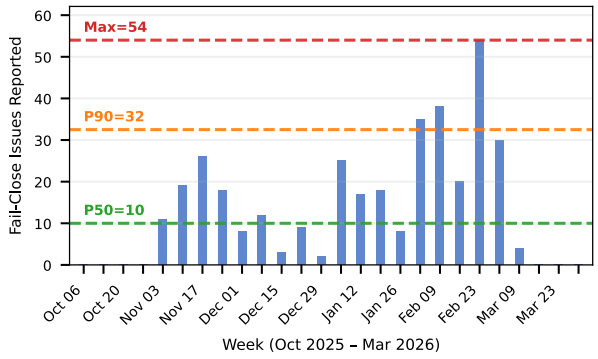



**Figure 10.** Weekly fail-close issues reported by commit-time analysis (Oct 2025 – Mar 2026). Dashed lines mark the median (10 per week), ninetieth percentile (32), and observed maximum (54).

These recurring reports demonstrate that commit-time analysis continuously exposes new cascading-failure risks while developers still have the code change in review.