Intent-Based Verification of Service Mesh Configuration

Ruize Ma
ShanghaiTech University
Shanghai, China
marz2024@shanghaitech.edu.cn

Haoxian Chen*
ShanghaiTech University
Shanghai, China
hxchen@shanghaitech.edu.cn

Abstract

Service mesh configurations are the critical routing layer for modern microservice deployments. Their misconfiguration is common and consequential, yet existing tools provide no semantic guarantees: schema linters miss silent routing errors, and sampling-based test suites cannot prove properties hold for all possible requests. We present a formal verification approach for service mesh traffic management that checks configurations against correctness specifications: both structural properties checked automatically on any configuration (no routing loops, no dependence on unspecified mesh semantics) and operator-specified intent properties (isolation, reachability). We lower the Istio configuration to proxy functions, abstract each service by how it rewrites attributes and forwards calls, and check intent over their composition via SMT, producing concrete counterexamples. A proof-of-concept evaluation catches injected violations of three intent classes (isolation, header propagation, and reachability) that escape `istioctl analyze` and cannot be proven by sampling-based approaches. At 100-host synthetic meshes, median per-query proof time stays under 90 s; all 500 queries return a definite verdict.

CCS Concepts

• **Networks** → **Network manageability**; **Formal specifications**.

Keywords

service mesh, configuration verification, network verification, microservices

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotNets '26, Salt Lake City, UT, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859958

ACM Reference Format:
Ruize Ma and Haoxian Chen. 2026. Intent-Based Verification of Service Mesh Configuration. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16-17, 2026, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3847290.3859958

1 Introduction

Service mesh systems such as Istio [11] and Linkerd [3] have become the communication backbone of modern microservice deployments. Istio alone comprises over 1,000 components and $3{\times}10^5$ lines of code [23], and provides more than ten custom resource types to configure traffic management. Every inter-service request flows through the mesh: routing decisions, load balancing, A/B testing, and access control are all expressed as declarative YAML configurations applied to the mesh data plane.

Misconfiguration is common and consequential. An empirical study of 2,260 Kubernetes configuration scripts found 719 defects; roughly one in three configurations contains a bug [22]. In March 2023, a Kubernetes upgrade misconfiguration caused a five-hour Reddit outage affecting millions of users [17]. Service mesh amplifies these risks: MeshTest [23] found 23 previously unknown bugs in Istio and Linkerd, which arose from *interactions between resources* rather than defects in any single resource.

Existing tools address parts of this problem: config linters (*e.g.,* `istioctl analyze`) catch schema violations, and MeshTest [23] uncovers implementation bugs via live traffic. Neither verifies configurations against operator intent before deployment. Here, *intent* refers to the routing behavior the operator designed the configuration to achieve: which requests should reach which services, which headers must propagate, and which paths should be isolated.

Consider a concrete gap (Figure 1): an operator routes requests under `/pay` to the payment backend, leaving the internal `/payouts` admin endpoint unauthenticated under the assumption that no external path reaches it. But Istio's `uri.prefix` is a raw string-prefix match, so `/payouts` also matches and reaches the admin endpoint directly. The YAML is schema-valid, `istioctl analyze` is silent, and the flaw is invisible until someone crafts that request in production.

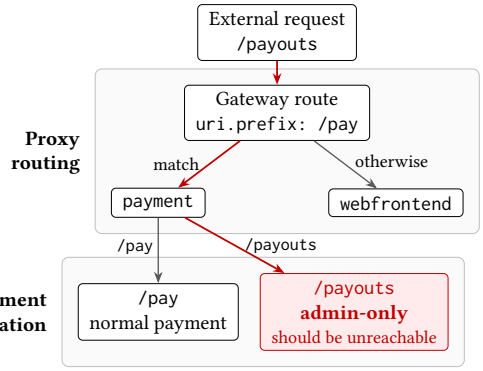


Figure 1: Raw-prefix routing exposes the admin-only `/payouts` endpoint because it begins with `/pay`.

Given the configuration, application summaries, and an intent assertion forbidding external requests from obtaining a response from the internal payout handler, a verifier synthesizes `/payouts` as a counterexample before deployment.

Challenges. Building such a verifier faces three challenges.

(1) Informal compositional semantics. Istio documents each resource in prose and JSON schemas, but not how resources compose: *e.g.,* how match priority resolves across exact, prefix, and wildcard rules, or what cross-stage side effects a `ServiceEntry` exerts on routing. These gaps are where bugs live, as the resource-interaction failures in MeshTest's corpus illustrate [23]. A verifier must therefore construct a faithful model of how resources compose.

(2) Application logic mutates routing attributes. Services sit between proxy hops and can rewrite the very request attributes that downstream routing decisions depend on. A verifier that treats services as black boxes cannot prove end-to-end properties; one that encodes arbitrary application code faithfully in a symbolic model is intractable. The right level of abstraction is a per-service *request transformer*: a summary of which attribute rewrites and calls each service may make, without modeling the code that does so.

(3) Deliberately unspecified semantics. In places the mesh's semantics are not merely informal but *deliberately* unspecified: when multiple VirtualServices bind the same gateway host, Istio documents their merge order as undefined [12]. A verifier cannot adopt one resolution; it must reason over every admissible order and flag configurations whose routing outcome depends on the choice.

Insight. Unlike L3/L4 network verification, where forwarding logic lives entirely in the control-plane rules, a service mesh has a second layer: services that receive requests, modify headers, and make their own routing decisions based on arbitrary application code. Modeling this layer at full fidelity is intractable. The key insight is that end-to-end routing properties depend only on *how services read and write request attributes*, not on their internal logic. This means (1) proxy forwarding logic can be extracted mechanically from configuration; (2) each service can be abstracted as a *request transformer* capturing only its routing-relevant read/write behavior; and (3) remaining non-determinism becomes explicit choices the solver must satisfy universally. The full deployment then composes into a single functional program that an SMT solver can reason over exhaustively.

This paper presents λMesh, a formal verifier for service mesh configurations (Figure 3 illustrates the two-layer model). Our contributions are:

- *Specification for service mesh.* A taxonomy of correctness properties for service mesh traffic management, including generic structural properties and operator-specified intent properties specific to each deployment (§3).
- *A formal verifier λMesh.* It checks configurations against intent using a two-layer model: proxy functions are extracted mechanically from YAML, while each service is modeled as a request transformer summarizing how it rewrites attributes and forwards calls, via an exact summary or a coarse over-approximation (§4).
- *Preliminary evaluation.* On configurations drawn from four widely used microservice examples, λMesh catches six injected, schema-valid misconfigurations that `istioctl analyze` misses, verifies clean baselines correctly, and shows measurably better precision with exact application summaries than with coarse application summaries; a scalability stress test decides every query up to 100 hosts (§5).

We conclude by discussing open research directions, including automated extraction of application summaries from application code and validation of the semantic model against live Istio deployments (§6).

2 Background

Service mesh traffic management. A service mesh (*e.g.,* Istio [11], Linkerd [3]) places a local proxy in front of every service instance, so all traffic between services passes through a proxy; a central controller distributes routing policy to every proxy. Figure 2 shows the two resources that set that policy for the running example of §1: a `Gateway` opens a listener at the network edge, and a `VirtualService` matches a request's URI, headers, and source against an ordered list of conditions and routes each match to one or more targets. Two further resources do not appear in this example: a `DestinationRule` names *subsets*, groups of service instances

```
kind: Gateway
metadata: {name: gw}
---
kind: VirtualService
spec:
  hosts: [shop.example.com]
  gateways: [gw]
  http:
  - match:
    - uri: {prefix: /pay}
    route:
    - destination: {host:
        payment}
  - route:
    - destination: {host:
        webfrontend}
```

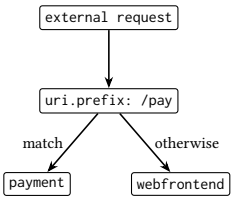


Figure 2: Example config snippet and the traffic flow.

picked out by a label such as a software version, that a routing target may address instead of a whole service; and a `ServiceEntry` admits an external hostname so it routes just like an in-mesh service. We use this same structure directly when translating configurations into a formal model (§4).

Two semantics are easy to miss. URI `prefix` matching is a raw string comparison, so `prefix: /pay` also matches `/payouts`, an internal endpoint the payment service exposes alongside `/pay` (§1). And when multiple VirtualServices bind the same (gateway, host) pair, Istio merges their rules but leaves the merge order *undefined* [12].

Compositional behaviors are hard to reason about. No single resource expresses end-to-end behavior: a ServiceEntry can shadow a VirtualService's host match, a mismatched header contract between two VirtualServices can silently drop a routing-critical header, and mismatched VirtualService and Gateway hosts can drop all traffic, each without error. Such cross-resource bugs are individually invisible: every resource passes schema validation and linting, and the fault emerges only from their composition [23]. The rest of this paper develops specification and verification to reason about them.

3 Specification

Correctness properties. We distinguish two kinds of correctness properties: generic structural properties and deployment-specific intent properties.

(1) Structural properties hold of any correct deployment and need no input from the operator (Table 1). LoopFreedom asks that forwarding a request through the mesh never cycles back to a service it already visited. OrderIndependence asks whether a configuration's routing outcome depends on the VirtualService merge order that Istio leaves undefined (§2). Both are checked automatically on any configuration.

(2) Intent properties say what a specific deployment should do. Take the running example: the operator's intent is that no external request can obtain a response from the internal payout handler. The operator states this as an assertion over the model, the Isolation row of Table 1. Reachability and HeaderPropagation state a reachability guarantee and a header contract the same way. A typical intent assertion can be read in five parts:

```
#assert forall (req : Req) [-> guard] ->
  ENTRY(SOURCE, req)[.OBSERVATION] == EXPECTED
```

- **Entry point:** where the request enters, either a named Gateway such as `gw[h]` or, for an in-mesh call, `mesh[h]`.
- **Source:** the request's last-hop caller at the entry point, written `@name` (*e.g.*, `@attacker` for an external client or `@checkout` for an internal service).
- **Guard (optional):** which requests to check, filtered by method, URI, or header; omitting it covers every request.
- **Observation (optional):** what to inspect on the result, such as a response header, a request header at the terminal workload, or a model-level *tag* (explained below); omitted, the whole response is compared (*e.g.*, to `http404`).
- **Expected value:** what the operator requires: a string, its absence, or a rejection.

In the isolation assertion above, the entry point is `gw["shop.example.com"]`, the source is `@attacker`, there is no guard, the observation is tag `internal`, and the expected value is `none`. Here, a tag is an in-model annotation that exists only in the verifier and is attached to modeled responses like a header. When summarizing `payment`, the operator marks its admin handler (`/payouts`) by setting tag `internal` on its responses, so the assertion can simply state that external requests must never reach internal functionality. The operator must supply this annotation since the verifier cannot infer which operations are sensitive. But the annotation effort is limited to responses from endpoints relevant to the intent that expose sensitive functionality; the operator need not tag every response. The assertion names no URI or matching rule, and the solver finds `/payouts` itself.

Operator inputs. To check a configuration before deployment, the operator supplies: (1) the service mesh configuration, (2) intent assertions, and (3) application summaries of relevant calls and transformations (§4). Assertions state required behavior without naming specific misconfigurations or triggering requests.

Verification results. The verifier composes mesh routing with the application summaries and searches for a request and routing choices that violate an intent assertion. The check has three possible outcomes: (1) the assertion is proved in the model; transferring this guarantee to a deployment requires the encoding and summaries to cover its relevant behavior; (2) a counterexample violates the assertion in the model, but may be unrealizable in the deployment if a

Table 1: Structural checks and intent assertions.

Property	Policy question	Example
Structural properties: checked automatically, no operator input		
LoopFreedom	Can a request be forwarded through the mesh in a cycle, back to a service it already visited?	`server[a]` ⇝ `server[b]` ⇝ `server[a]` (cycle)
OrderIndependence	When several VirtualServices bind the same gateway host, does the routing outcome depend on the order Istio applies them?	`gw[x]` → `VS1, VS2` → `server[a]` `gw[x]` → `VS2, VS1` → `server[b]`
Intent properties: stated by the operator as an assertion		

Isolation	Can an external source obtain a forbidden response attribute through some entry point?	`#assert forall (req : Req) ->`
		`gw["shop.example.com"](@attacker, req)`
		`.tag(‘internal) == none`

Reachability	Does every request in some traffic class reach its intended target?	`#assert forall (req : Req) ->` `mesh[recommendation](@checkout, req) != http404`

HeaderPropagation	Does a header set upstream survive to the terminal workload?	`#assert forall (req : Req) ->`
		`mesh[checkout](@client, req)`
		`.headerReq(‘recVariant) == "v1"`
		`|| mesh[checkout](@client, req)`
		`.headerReq(‘recVariant) == "v2"`

coarse application summary admits extra behavior; the operator must inspect it and may need to refine the summary; or (3) an inconclusive result (unknown or timeout) leaves the assertion unproven.

4 Model and Verification

We check a configuration against operator-specified intent by treating each network hop—through a proxy or a service—as a function call from request to response, compiling the deployment into a small functional program and asking an SMT solver whether every request satisfies the assertion.

As shown in Figure 3, the approach takes three inputs: (i) the mesh configuration; (ii) the operator-specified intent assertions described in the previous section; and (iii) application summaries. It then checks the property against all possible requests and routing behaviors.

Modeling mesh semantics. Figure 3(b) is the model extracted from the YAML in (a). Line 2 is a *proxy*: given a request to `recommendation`, it picks a backend, `v1` or `v2`, matching the 50/50 split in (a) (lines 3–5). Lines 8–16 are *servers*, one per service, saying what each does with a request it receives: `checkout` forwards to `recommendation` unchanged (line 9); whichever backend is picked sets the request header `recVariant` to its own version before forwarding to `catalog` (lines 10–15); `catalog` replies (line 16). Composing these definitions gives one function for the deployment: a multi-hop request is just one call leading to the next, traced in (c).

Application summaries. Mesh semantics are built into λMesh; operators only summarize what each service does with a request, on a spectrum where recording more application logic lets more properties be proved.

(i) Exact summary. When forwarding depends only on the request, the operator writes the service's forwarding logic (lines 8–16 of Fig. 3(b)): which callee and rewrites each request triggers.

(ii) Coarse summary. When the application is opaque, it suffices to know which callees and attribute rewrites each service may trigger, without modeling the code that decides between them: each hop becomes an unconstrained choice (`NSelect`) among those candidate edges and rewrites.

In between, a summary keeps request-dependent branches and makes only stateful ones (*e.g.,* a database lookup) nondeterministic. In Fig. 4, the service routes on a database lookup, so no exact summary exists; keeping only its URI test still proves that non-`/login` requests reach `frontend`, while the coarse summary raises a false alarm. A coarse summary must over-approximate the service's relevant behavior for proofs to transfer to the deployment; a counterexample may still be unrealizable. Every summary must cover the calls and attribute transformations relevant to the assertion, including those used in downstream routing; §5 measures the precision gap.

Verification. Verification proceeds in two steps: first encode non-determinism and discharge structural checks, then ask the solver if the assertion holds under all of it.

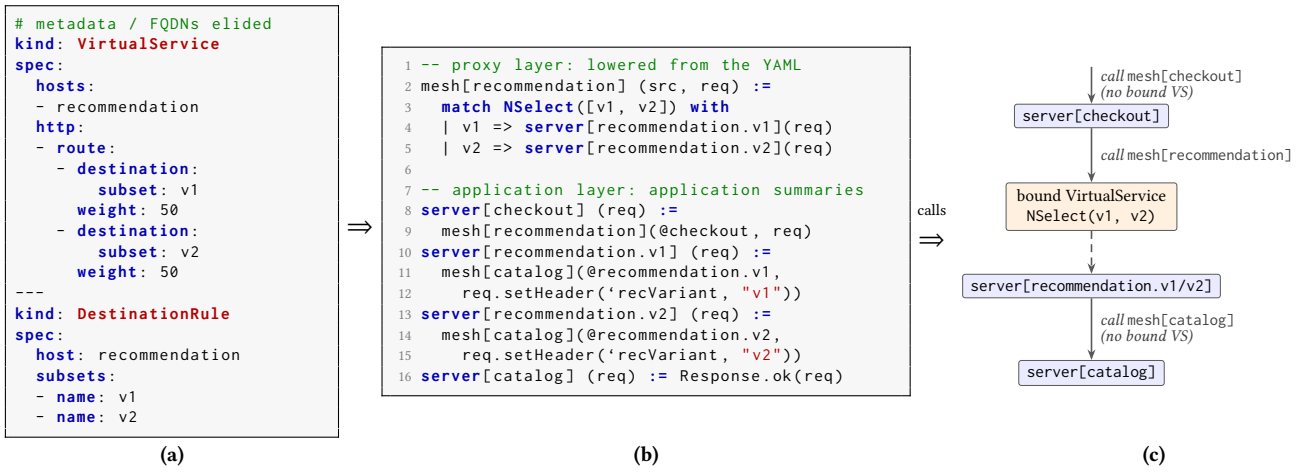


Figure 3: A canary release: (a) YAML, (b) the model with application summaries, and (c) its call path.

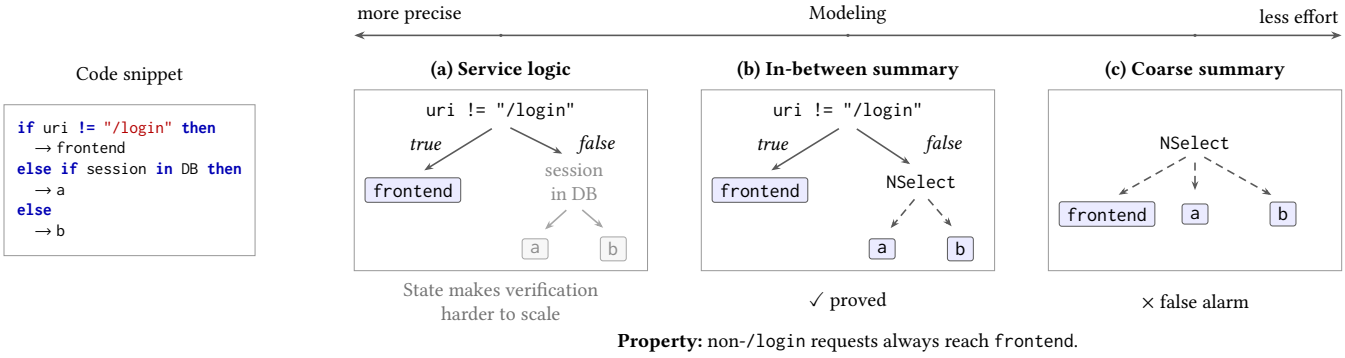


Figure 4: (a) Service logic; (b) in-between summary: abstracts state; (c) coarse summary: abstracts both tests.

Step 1: Non-determinism and structural checks. Before checking intent, the verifier encodes non-deterministic choices and checks the two structural properties (§3). (i) *Weighted routing*: which backend a weighted route picks becomes a free solver variable, so a property must hold for every possible backend selection. (ii) *VirtualService merge order*: when several VirtualServices bind the same gateway host, Istio does not specify which applies first (§2). Rather than assuming one order, the verifier checks whether order can ever affect routing. Let π_0 be the declared order and π_i any other permutation; the solver asks, for each:

$$\exists\,(src, req).\ route(\pi_i, src, req) \neq route(\pi_0, src, req).$$

If no such request exists, order never matters and OrderIndependence holds; otherwise the operator is warned that routing is order-sensitive. (iii) *Routing cycles*: a cycle would forward a request forever; LoopFreedom holds if the forwarding call graph is acyclic.

Step 2: Checking an assertion. The compiler encodes proxy and server definitions as solver functions, preserving calls as function applications. Requests and responses are records of fields such as URI, headers, tags, and method, with string operations encoded in the solver's string theory. For a universally quantified assertion, the solver searches for a counterexample: `unsat` proves the assertion for every modeled request and choice; `sat` identifies a counterexample in the model. An `unknown` result or timeout is inconclusive. Checking the header-propagation assertion of §3 against Figure 3(b) confirms it: every request reaching `catalog` carries `recVariant`, whichever backend `NSelect` picks. Drop the `setHeader` call from line 12, and checking again finds a counterexample: a request taking the `v1` branch reaches `catalog` with no `recVariant` set.

5 Preliminary results

We evaluate λMesh on two questions: (1) *Effectiveness*: can the verifier find intent violations? (2) *Scalability*: how does verification time grow with mesh size?

Table 2: Evaluation deployments.

Deployment	VirtualServices	Assertions
Bookinfo [10]	1–7	3–8
Sock Shop [20]	13–14	12–21
Online Boutique [7]	2	9–21
Kubeflow (slice) [13]	16	16

Table 3: Injected misconfigurations.

Misconfiguration	Property
Configuration errors	
Canary header written `"true"`, matched `"True"`	HeaderPropagation
`exact /pay/` vs. app-issued `/pay`	Reachability
Extra list dash turns AND into OR	Isolation
`prefix /pay` admits `/payouts` (§1)	Isolation
Canary weights mistyped 100/0	Reachability
Documented non-determinism	
Two VirtualServices on one gateway host	OrderIndependence

Implementation. The prototype compiles Istio and Kubernetes YAML together with application summaries (either exact summaries or coarse call-graph summaries) into solver queries (4,143 LOC). We evaluate both summary types in RQ1 and compare their verification costs in RQ2. We use Z3 4.16 and `istioctl` 1.29.2.

Dataset and setup. We evaluate on four public deployments (Table 2). For each, we write 16–153 lines of hand-written exact application summaries capturing request-dependent forwarding. Sock Shop is additionally evaluated with coarse call-graph summaries to measure the precision tradeoff (§4).

RQ1: effectiveness. We injected six misconfigurations (Table 3) into 11 configurations, checked by 29 intent queries; none is drawn from a deployed system. Each passes `istioctl analyze`, yet the verifier catches each before deployment. Separately, we verified 16 application-model variants over 11 correct baseline configurations with 196 intent queries; every verdict matched the intended outcome. No single query exceeds 0.05 s; the full corpus finishes in under 2.1 s, and each misconfiguration is caught within 0.04 s.

Summary precision tradeoff. In Sock Shop, an exact summary records that every `/cart` request from `front-end` calls `carts`, so the verifier proves that all such requests reach `carts`. A coarse summary records only that `front-end` may call several services, including `catalogue` and `carts`; as in Fig. 4(c), the solver may then route a `/cart` request to `catalogue`, a false positive that the exact summary rules out.

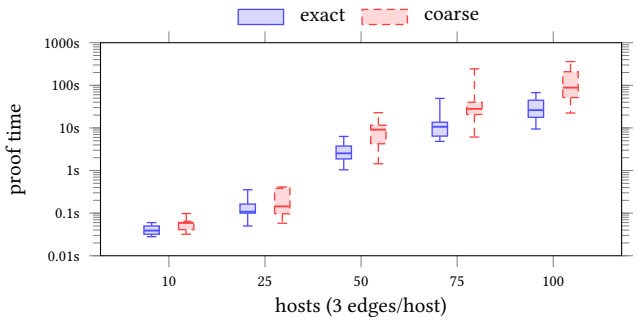


Figure 5: Mean per-query proof time across ten seeds; whiskers span seed-level minima and maxima.

RQ2: scalability. We stress-test both summary types (§4) on synthetic layered meshes with 10–100 hosts, three edges per host (e=3h), five headers, and five to ten layers. For each size, ten random seeds yield paired exact- and coarse-summary models with the same topology. We check five isolated assertions per model, for 100 models and 500 solver queries in total. All 500 queries return a definite verdict.

Figure 5 plots per-seed mean proof time over five queries. At 100 hosts, median proof time is 26.1 s for exact-summary models and 88.3 s for coarse-summary models, so the additional behaviors admitted by coarse summaries increase verification cost but remain practical at this scale.

Scope and generalizability. These results show that the approach is effective and scales to realistic mesh sizes. The RQ2 meshes bound scalability on topology; real deployments at the same scale are typically sparser, so we expect faster verification in practice. Our evaluation targets Istio, but the modeling recipe (proxies lowered from configuration, services as request transformers) is not Istio-specific: supporting Linkerd only requires a front end that compiles its configuration into a similar DSL.

6 Discussion

This work demonstrates that formal verification of service mesh configurations is tractable and finds bugs that standard tooling misses. Several directions build naturally on this foundation and open interesting research problems at the intersection of networking and formal methods.

Automating application summary extraction. Unlike mesh configuration, application routing logic lives in service code. Coarse summaries—candidate callees and header rewrites—could be mined from distributed traces (*e.g.,* OpenTelemetry or Envoy access logs), while static program analysis of common web and RPC frameworks (or LLM-assisted code summarization) could extract request-dependent forwarding rules automatically. A more general direction is to treat the summary as a *verified assumption*: given the properties

an operator wants to prove, synthesize the weakest application summary that makes verification succeed, then validate that assumption against the implementation. This direction directly connects mesh-layer and application-layer verification, enabling end-to-end routing correctness proofs without requiring full application analysis.

Validating the semantic model against Istio. Istio has no formal specification to prove our hand-written encoding against, and the running implementation is only a partial oracle: where behavior is deliberately unspecified, it exhibits one admissible outcome, not the full semantics. We therefore test *containment* rather than equality: a reverse compiler translates DSL models into runnable Kubernetes/Istio YAML and service stubs, replays solver-generated HTTP requests against a live cluster, and checks that the observed routing falls within the outcomes the model admits. Preliminary tests on generated models confirm consistency across core routing semantics; the remaining discrepancies stem from undocumented Istio behaviors, reverse-compiler lowering gaps, and features not yet modeled in the DSL.

Semantic diff for safe rollouts. The same solver infrastructure that checks a configuration can compare two versions of it: ask the solver for a request whose routing outcome differs between them. The result is a precise semantic diff that tells operators exactly which request classes a configuration change affects, far more actionable than a YAML text diff. Integrated into a deployment pipeline, this gives teams a principled way to scope the blast radius of a rollout before it reaches production.

Dynamic mesh behaviors. The current model covers static routing configuration: path matching, header manipulation, and weighted routing. Retries, timeouts, and health checks all introduce runtime state and change what correct routing means under failure. Modeling these behaviors requires capturing stateful semantics, which is substantially harder but would make the verifier applicable to a much broader class of operational correctness questions.

7 Related Work

Service mesh correctness. MeshTest [23] models service flows to a target workload and uses symbolic execution to produce request suites that uncover mesh implementation bugs. Our work is complementary: we check configurations against operator intent before any YAML reaches a cluster, extending the model with per-service request transformations across multiple hops.

Policy languages. Rego [15] is a general-purpose policy language over structured data; Cedar [4] targets authorization policies. λMesh checks end-to-end intent assertions against the composition of application summaries and mesh configurations, detecting intent violations even when individual resources are locally valid.

SafeTree [8] specifies multi-hop constraints over service-call trees and enforces them at runtime using a distributed visibly pushdown automaton implemented atop Istio. It treats the mesh as an enforcement substrate for a new policy language; λMesh instead assigns semantics to existing Istio traffic-management configurations and checks them against operator intent before deployment.

Copper and Wire [18] design a new cross-dataplane policy language and optimize sidecar placement. λMesh complements this by verifying intent against existing configurations, and could serve as a verification layer atop Copper.

Network verification. Minesweeper [2], Batfish [5], NV [6], APKeep [21], and Tiramisu [1] verify L3 routing such as BGP and OSPF. λMesh brings this approach to L7 service mesh, where header mutation and application-induced recursion fall outside BGP's fixed-point model.

Configuration analysis and linting. `istioctl analyze` and Kiali check mesh configurations against schemas and patterns, without semantic reasoning. Kano [14] verifies L3/L4 NetworkPolicy reachability, and SLI-KUBE [16] statically finds security misconfigurations in Kubernetes manifests. These tools operate on individual resources; λMesh verifies compositional behavior across resource interactions.

Microservice reliability. Acto [9] and Sieve [19] test Kubernetes operator correctness via state reconciliation; they target controller behavior, not traffic management semantics. Our work is orthogonal.

8 Conclusion

Service mesh configurations shape how every request moves through a microservice deployment, yet no tool checks them against operator intent before they reach production. We showed that treating each hop as a function call enables a two-layer model: a proxy layer extracted mechanically from configuration and an application layer captured by operator-supplied application summaries of how each service rewrites attributes and forwards calls. Verification reduces to an SMT query over this model. A preliminary evaluation catches injected misconfigurations that existing linters miss and remains tractable at realistic scale. Automating application summary extraction from application code is the primary direction for future work.

Acknowledgments

We thank our shepherd, Akshay Narayan, and the anonymous reviewers for their constructive feedback. This work is supported by the ShanghaiTech Startup Fund.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. doi:10.1145/3098822.3098834

[3] Buoyant, Inc. 2024. Linkerd: The World's Lightest, Fastest Service Mesh. https://linkerd.io.

[4] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (April 2024), 28 pages. doi:10.1145/3649835

[5] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[6] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: an intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 958–973. doi:10.1145/3385412.3386019

[7] Google Cloud Platform. 2024. Online Boutique: Cloud-Native Microservices Demo Application. https://github.com/GoogleCloudPlatform/microservices-demo.

[8] Karuna Grewal, Brighten Godfrey, and Justin Hsu. 2025. SafeTree: Expressive Tree Policies for Microservices. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 349 (Oct. 2025), 27 pages. doi:10.1145/3763127

[9] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112. doi:10.1145/3600006.3613161

[10] Istio Authors. 2024. Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/.

[11] Istio Authors. 2024. Istio: Connect, Secure, Control, and Observe Services. https://istio.io.

[12] Istio Authors. 2025. Traffic Management Best Practices: Split Large Virtual Services and Destination Rules into Multiple Resources. https://istio.io/latest/docs/ops/best-practices/traffic-management/#split-virtual-services "Although the order of evaluation for rules in any given source VirtualService will be retained, the cross-resource order is UNDEFINED.".

[13] Kubeflow Community. 2026. Kubeflow Community Distribution. https://github.com/kubeflow/community-distribution.

[14] Yifan Li, Chengjun Jia, Xiaohe Hu, and Jun Li. 2020. Kano: Efficient Container Network Policy Verification. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. 63–70. doi:10.1109/HOTI51249.2020.00024

[15] Open Policy Agent Authors. 2026. Policy Language (Rego). https://www.openpolicyagent.org/docs/policy-language.

[16] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 99 (May 2023), 36 pages. doi:10.1145/3579639

[17] Reddit Engineering. 2023. You Broke Reddit: The Pi-Day Outage. https://www.reddit.com/r/RedditEng/comments/11xx5o0/you_broke_reddit_the_piday_outage/.

[18] Divyanshu Saxena, William Zhang, Shankara Pailoor, Isil Dillig, and Aditya Akella. 2025. Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 233–248. doi:10.1145/3669940.3707257

[19] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

[20] Weaveworks. 2024. Sock Shop: A Microservices Demo Application. https://github.com/microservices-demo/microservices-demo.

[21] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 241–255. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

[22] Yue Zhang, Uchswas Paul, Marcelo d'Amorim, and Akond Rahman. 2026. Configuration Defects in Kubernetes. *IEEE Transactions on Software Engineering* 52, 3 (2026), 1125–1144. doi:10.1109/TSE.2026.3659785

[23] Naiqian Zheng, Tianshuo Qiao, Xuanzhe Liu, and Xin Jin. 2025. MeshTest: End-to-End Testing for Service Mesh Traffic Management. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 301–316. https://www.usenix.org/conference/nsdi25/presentation/zheng-naiqian