Application Defined Networks

Xiangfeng Zhu

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington
2026

Reading Committee:
Arvind Krishnamurthy, Co-Chair
Ratul Mahajan, Co-Chair
Stephanie Wang

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Application Defined Networks

Xiangfeng Zhu

Co-chairs of the Supervisory Committee:

Professor Arvind Krishnamurthy
Computer Science and Engineering

Professor Ratul Mahajan
Computer Science and Engineering

Cloud applications today are no longer monolithic programs in the classical sense. They are graphs of microservices in which a single user request fans out into tens or hundreds of remote procedure calls (RPCs), and end-to-end latency and operating cost are determined not only by what each service computes but also by how those services communicate. Yet the way they communicate remains rooted in the Internet architecture. A typical RPC traverses a stack of general-purpose protocols—Protobuf, gRPC, HTTP/2, TLS, TCP, IP—while network policies are often enforced by an out-of-process proxy that must terminate that stack, parse it, and re-originate the request. The result is what this dissertation calls the *RPC tax*: a compound penalty in latency, CPU, and engineering complexity paid on every microservice call, in a setting—a single administrative domain—where much of the layered generality being paid for is not actually needed.

This dissertation argues that the RPC tax is not inherent but a design choice, and it proposes a different approach: *Application Defined Networks (ADN)*, a model in which the entire application network stack—serialization, end-to-end delivery, and application network functions—is specified once at a high level and compiled into a flat, application-specific implementation, rather than assembled at runtime from general-purpose layers.

The thesis is organized around three pillars, each with its own complete artifact. First, **MeshInsight** establishes the problem empirically through a compositional cost model that decomposes an application network proxy into its component operations and shows that today's stack inflates application latency by up to 269% and CPU consumption by up to 163%, with protocol parsing as the dominant cost. Second, **AppNet** introduces a simple and analyzable model for application network functions together with a symbolic-equivalence engine that lets the compiler fuse, reorder, and reposition stateful elements without breaking observable behavior. Common application network functions—and production designs such as Meta's ServiceRouter and Google's Prequal—can be expressed in tens of lines of AppNet code, while compiler-driven optimization reduces latency by up to 82% and CPU by up to 75% over unoptimized baselines. Third, **fRPC** eliminates the layered stack itself: from a high-level specification, it synthesizes a policy-aware split-partition wire format, a non-terminating and non-buffering proxy, and shims that automatically resolve interference between delivery semantics and policy semantics, cutting end-to-end application latency by up to 52% and CPU usage by up to 33% versus the common deployment of gRPC + Envoy while preserving the same semantics and security guarantees.

Together, the three artifacts demonstrate a compilation approach and a new way to build the application network substrate of the cloud: not as standardized infrastructure configured after deployment, but as an application-specific artifact generated from a specification.

Acknowledgements

This dissertation would not have been possible without the support, guidance, and encouragement of many people over the course of my Ph.D. First and foremost, I am deeply grateful to my advisors, Arvind Krishnamurthy and Ratul Mahajan, for their mentorship throughout my time at the University of Washington. From our first meeting, they helped me shape my research direction and provided the freedom, guidance, and resources to pursue it.

Arvind's ability to see the bigger picture and to ask the right questions at the right moments has repeatedly steered my work in directions I would not have found on my own, and his feedback—always sharp, always generous—has made every paper and chapter stronger. Ratul has guided me through every aspect of my Ph.D., from formulating high-level research directions to working through design choices and implementation details. He has been there whenever I needed him, while also giving me the freedom to explore and grow as a researcher when I needed that instead. I feel incredibly fortunate to have had two advisors whose complementary strengths and unwavering support have shaped not only this dissertation, but also the kind of researcher I aspire to be.

I am also thankful to the members of my supervisory and reading committees: Stephanie Wang, Danyang Zhuo, and Akshay Gadre. Danyang has mentored me since my first month at UW and helped me with many aspects of every project I worked on. He provided invaluable feedback and guidance on my research, especially in my early days.

I want to thank Peter Alvaro for introducing me to systems research; his passion for research and teaching still inspires me to this day. I also thank Mosharaf Chowdhury, Fan Lai, and Harsha V. Madhyastha for their guidance during my undergraduate studies and research at the University of Michigan, and for their support in applying to graduate school.

I have been fortunate to collaborate with many talented researchers and co-authors. Thank you to Guozhen She and Matthew Lentz for MeshInsight; Banruo Liu, Yongtong Wu, and Jingrong Chen for AppNet; and Yang Zhou, Xiangyu Gao, Sam Kumar, Aruj Bansal, and Zhuo Chen for fRPC. I am especially grateful to Banruo, Yongtong, Zhuo, Aruj, and Jingrong, who helped me tremendously with the design and implementation of AppNet and fRPC. Matthew and Sam contributed enormously to the design and writing.

During my Ph.D., I had the privilege of interning at VMware Research and Uber. I thank my mentors Radhika Niranjan Mysore and Hongqiang Liu for giving me the opportunity to work on real-world problems and to learn from them.

I am grateful to the members of the systems lab, including but not limited to Zezhou Wang, Wei Shen, Shihang Li, Lequn Chen, Tianyi Cui, Xieyang Xu, Jaehong Min, Shengkai Lin, Tapan Chugh, Chenxingyu Zhao, Liangyu Zhao, Dedong Xie, Kan Zhu, Yile Gu, Yingbo Sun, Mathew Jacob, Zihao Ye, Hongtao Zhang, Jialin Li, Katie Lim, Priyal Suneja, and Samantha Miller, for the discussions, reading groups, conference trips, and friendship that made graduate school so rewarding.

To my friends, both at UW and beyond—Yizhong Wang, Haiyan He, Yuhan Wan, Han Zhang, Haotian Jiang, Enhao Zhang, Kaiyu Zhang, Yuhan Liu, Xia Su, Kaiming Cheng, Xinming Tu, and Dong He—thank you for the meals and parties, the weekend adventures, the late-night conversations, and the steady support that kept me sane through graduate school.

I would also like to acknowledge the staff of the Paul G. Allen School of Computer Science & Engineering, especially Joe Eckert, Christopher Nagle, and Elise Dorough, for their tireless help with travel, reimbursements, scheduling, and the countless administrative details that allowed me to focus on research.

Finally, I owe my deepest gratitude to my family. To my parents, thank you for your unwavering love, the sacrifices you made along the way, and your steadfast support of my education from the very beginning. To my grandparents, thank you for the wisdom, warmth, and encouragement that have shaped me throughout my life. To my aunt and uncle, thank you for your kindness and support, especially when I first arrived in the United States to begin my undergraduate studies. To my partner, Anqi Gao, thank you for your patience, partnership, and love through every stage of this journey. This year, we welcomed our daughter, Qinghuan (Riley), into the world; she arrived just as this thesis was nearing its end, and you made both possible in more ways than I can express. I am grateful to you both beyond words. Throughout the highs and lows

of graduate school, you have been my constant source of strength, encouragement, and perspective. You celebrated every success with me, carried me through moments of doubt, and reminded me of what truly matters when the work felt overwhelming. This dissertation may bear my name, but it reflects years of sacrifices and support that we shared together. Welcoming Riley while completing this thesis has made this final chapter especially meaningful, and I am deeply grateful for the life and family we have built together.

DEDICATION

To everyone who believed in me and supported me through this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern cloud applications are no longer monoliths. The software that powers a checkout at Amazon, a watch session on Netflix, or a ride request at Uber is not a single binary running on a single machine; it is a graph of tens to hundreds of independently developed microservices, often owned by different teams within the same administrative domain, each scaled and updated on its own schedule, and each communicating with the others through remote procedure calls (RPCs), with policies like traffic management, security, and observability enforced on top. A single user-facing request typically fans out into dozens or hundreds of downstream RPCs [69, 98, 115]. In this regime, the latency and operating cost of an application are determined by both what each service *computes* and how those services *communicate*.

Yet the way they communicate still follows an older design. Every RPC traverses a stack of general-purpose protocols—Protobuf for serialization, gRPC and HTTP/2 for framing and multiplexing, TLS for confidentiality, TCP for reliable delivery and congestion control, IP for routing [18, 39, 112]—and policies beyond the application itself (e.g., load balancing, rate limiting, retries, observability, and mTLS) are enforced by an out-of-process proxy that must terminate this stack, parse it, run its policies, and then re-originate the stack on the other side. This is the canonical service-mesh architecture [13, 29, 44]: built from protocols designed for interoperability with arbitrary services across the open Internet, but deployed inside data centers in which the calling and the called service are owned by the same organization.

This dissertation calls the resulting overhead—across both the layered protocol stack and the out-of-process policy enforcement proxy—*the RPC tax*: a compound penalty in latency, CPU, and engineering

complexity that today's stack levies on every microservice call. The tax is large—as we show in this thesis, on two representative microservice benchmarks, enabling an industry-standard service mesh can raise end-to-end latency by up to 269% and CPU consumption by up to 163% relative to running without the service mesh. Multiplied across millions to billions of intra-data-center RPCs that real cloud services issue every day, those overheads translate into entire fleets of servers spent on infrastructure that computes nothing the application asked for.

This dissertation argues that the RPC tax is not inherent. It is the cumulative price of layering—the price of forcing each application's communication needs through a stack whose generality is mandated by the Internet architecture but is not actually required inside a single administrative domain [60, 113]. Once we name the tax as a design choice, a different choice becomes available. This dissertation proposes that choice and defends it: *Application Defined Networks (ADN)*, a model in which the entire RPC communication stack—serialization, end-to-end delivery, and application network functions—is specified once at a high level and *compiled* into a flat, application-specific implementation, rather than assembled at runtime from interchangeable general-purpose layers.

1.1 Application Networks Today

To appreciate why the layered and proxy-centric stack used for microservice communication is the wrong starting point, it helps to look at what that stack actually does and at what it was designed to do.

What an application network is. In a modern microservice deployment, an *application network* is everything that sits between the function call inside one service and the function call inside another. Concretely, it is the cooperation of (i) a wire format for marshalling typed objects into bytes (e.g., using Protobuf, FlatBuffers, JSON), (ii) an RPC framework that defines call semantics and frames messages on top of an application-layer protocol (e.g., using gRPC or Thrift), (iii) an application-layer protocol that multiplexes and demultiplexes those framed messages over a single long-lived connection (typically HTTP/2), (iv) a transport and security layer that delivers and protects those bytes (typically TLS over TCP), and (v) a set of *application network functions* (ANFs) implemented by an out-of-process proxy: load balancing, retries, timeouts, rate limiting, circuit breaking, telemetry, request routing, and traffic shaping. The proxy is most

commonly Envoy, deployed as a sidecar process next to every application instance or as a middlebox-style remote proxy managed by control planes such as Istio [13, 29].

How ANFs are deployed today. ANFs can be placed in several deployment modes [27, 80, 90, 119]. In the sidecar model [13, 44], each service instance is paired with a local proxy, which gives fine-grained policy enforcement, per-instance visibility, and language independence, but adds extra local hops, IPC, memory footprint, and operational complexity. In a middlebox-style deployment [80, 119], traffic is steered through a shared remote proxy, which reduces per-instance resource overhead and centralizes upgrades, but can introduce longer paths, coarser failure domains, and less direct visibility into application-local behavior. A third model is proxyless RPC, where clients and servers implement selected ANFs inside the RPC library itself [90]; this removes the out-of-process hop and can reduce data-path overhead, but it ties policy support to a particular RPC framework and language runtime, making upgrades, heterogeneity, and unsupported traffic harder to manage. These modes trade off performance, isolation, operational control, and portability, but all preserve the broader premise that application communication is assembled from general-purpose layers plus a policy mechanism outside the application logic.

Why it looks the way it does. Today's stack is not accidental. It is the result of reusing successful general-purpose abstractions whose operational benefits were worth more, in practice, than the overhead they added to each RPC. Protobuf and gRPC provide language-neutral interfaces, schema evolution, and broadly supported RPC semantics [18, 39]; HTTP/2 provides a standardized substrate for framing, multiplexing, streaming, and flow control. TLS and TCP provide standardized security, reliability, congestion control, and compatibility with existing operating systems and network infrastructure. The out-of-process proxy gives infrastructure teams a place to enforce policy without modifying application code, while preserving language independence, operational consistency, and the ability to upgrade infrastructure independently of application code. Each choice is reasonable in isolation, but together they impose a substantial cost in bytes, parsing, buffering, handshakes, IPC, syscalls, and repeated protocol termination. Compounded across every microservice call, that cost becomes the RPC tax.

Why the setting has changed. The main justification for the tax is that the stack must work for every application: across organizations, across trust domains, across deployment schedules. However, the dominant workload that this stack now serves does not look like that. A modern microservice deployment is usually a graph of services owned by the same organization, deployed from the same pipeline, trusted to the same root of trust, and updatable in lockstep when needed. The CNCF's recent surveys report that microservices and service meshes are now used by the large majority of cloud-native organizations [61]. In other words, the stack provides portability and operational leverage, but it imposes a much higher cost than it should on calls that often run within one administrative domain.

What today's responses leave on the table. We are not the first to recognize that today's stack is not optimal. eBPF and kernel-bypass have shaved syscall and IPC costs off the data path [2, 79, 107]; QUIC has compressed the handshake and re-implemented reliability in user space [40]; new service mesh designs, including eBPF-accelerated proxies, have rearranged where policy is enforced [2, 80]. Each of these helps, but each accepts the basic shape of the stack—a fixed sequence of general-purpose layers, with policies inserted at a proxy that must terminate and re-originate them—and so each can, at best, reduce only a small portion of the RPC tax. Closing the gap requires asking a different question: not "how do we make each layer cheaper?" but "why are these layers fixed in the first place?"

1.2 Thesis Approach

The goal of this dissertation is to make the implementation of an application network a *compiled artifact*. Concretely, the dissertation argues that developers should specify, at a high level, (1) the per-RPC behavior they expect from the network (ANF semantics) and (2) the delivery semantics they require end-to-end (reliability, ordering, encryption boundaries), and that a compiler should synthesize the implementation that realizes those specifications: the wire format, the proxy execution model, the placement of ANFs, the boundaries of encryption, and the shims that reconcile delivery semantics with policy semantics. The same specification can target different runtimes (user-space proxy today, kernel datapath or SmartNIC tomorrow) without changing the program the developer writes.

To realize this vision, the dissertation takes a three-part approach. The first part establishes empirically

that today's layered stack is the problem—and explains, at component granularity, where its cost actually comes from. The second part contributes a high-level programming model for application network functions and a compiler that can generate optimized implementations while preserving observable behavior. The third part removes the layered stack itself, synthesizing a flat, application-specific communication stack from the same kind of high-level specification. Each part is realized as a complete artifact, evaluated end-to-end on production microservice benchmarks, and released as open-source software.

1.2.1 Dissecting the RPC Tax

Before redesigning the application network, the dissertation establishes empirically that today's stack is the problem—and that the question of *why* cannot be answered by black-box benchmarking alone. Existing studies of service mesh performance compare "with-mesh" against "without-mesh" latency on a few benchmarks and report a few overhead numbers [81, 96, 107]. Those numbers are enough to motivate a redesign, but they are not enough to *guide* one: they tell us neither which components of the proxy dominate the cost, nor how that breakdown changes with the deployment, the workload, or the protocol mode (HTTP/gRPC versus TCP). Because sidecars are the dominant production deployment mode for service mesh proxies, this part of the dissertation focuses on the sidecar model.

The first part of the dissertation contributes **MeshInsight**, a *compositional* model for service meshes with sidecar proxies. MeshInsight decomposes a sidecar's work into a small set of components—IPC with the application, system calls, event notifications, protocol parsing, and per-ANF processing—profiles each component once on a given platform, and then composes the resulting micro-measurements to predict the end-to-end overhead of real deployments without re-running full experiments. The model exposes two findings that anchor the rest of the dissertation. First, enabling Istio, a popular service mesh that uses Envoy as its sidecar proxy, increases latency by up to 269% and CPU consumption by up to 163% on representative microservice benchmarks [139]. Second, the cost breakdown is not uniform: in HTTP/gRPC mode the dominant cost is *protocol parsing*, while in TCP mode it is IPC and kernel interaction.

1.2.2 High-Level Programming for Application Network Functions

Building on MeshInsight's cost breakdown, the dissertation next turns to the layer application developers can most directly reshape: application network functions (ANFs). In today's systems, an ANF is usually expressed in one of two unsatisfying forms: declarative configuration for pre-implemented policies (for example, YAML rules interpreted by the control plane [29]) or low-level code (for example, a Rust-based WebAssembly filter [43] compiled into the proxy). Configuration is easy to use but cannot express the custom and complex policies that real deployments need [114, 131]; code is expressive, but it is written against a low-level, side-effect-rich proxy API, which imposes a high cost in both development and maintenance.

The second part of the dissertation contributes **AppNet**, a high-level programming model for ANFs and a compiler that generates optimized ANF implementations while preserving observable behavior. In AppNet, an ANF is a chain of *elements* with explicit state and explicit match-action rules over RPC fields and the element's internal state; this high-level semantic model is what makes optimization possible. Concretely, AppNet contributes three ideas. First, it makes the dataflow and state of every element first-class, so that the compiler can decide which fields each element reads and writes and which state it depends on. Second, it introduces *symbolic equivalence*—using symbolic abstraction and symbolic execution—to decide whether two candidate configurations of stateful ANFs are equivalent for arbitrary streams of RPCs, providing the safety net that lets the compiler fuse, reorder, and relocate elements freely. Third, it exposes *consistency as a knob*: developers can opt into weaker forms of state or observation consistency in exchange for higher performance when their policies can tolerate those weaker guarantees, expanding the optimizer's search space.

In the evaluation, 12 common ANFs require 7–28 lines of AppNet code each—a 5× to 60× reduction over native C++/Rust filters—and the abstraction tax relative to hand-tuned Envoy filters is 1–4%. Compiler-driven reordering, fusion, and relocation of elements reduce latency by up to 82% and CPU by up to 75% relative to the unoptimized baseline. To further validate AppNet's expressiveness, the dissertation also implements industrial designs—Meta's ServiceRouter [114] and Google's Prequal [131]—in fewer than 100 lines of AppNet code each.

1.2.3 Synthesizing Flat RPC Communication Stacks

AppNet reduces the policy overhead of the application network, but it still relies on a layered stack: the proxy still terminates gRPC, the wire format is still general-purpose Protobuf, and reliability still travels through TCP. In a motivating study, protocol termination and packet buffering alone cause a 2.5× latency increase and a 1.8× CPU increase for RPCs that traverse a sidecar proxy, even before any policy runs.

The third part of the dissertation contributes **fRPC**, a compilation pipeline that synthesizes a flat, application-specific communication stack from a single specification. fRPC's input has three pieces: (a) an existing Protobuf-style RPC schema, (b) an AppNet ANF pipeline, and (c) a developer-written *delivery* pipeline that expresses reliability, ordering, encryption, and congestion control as event-driven elements. Its output is a complete RPC stack tailored to the application:

- A *policy-aware, split-partition wire format* in which exactly the RPC fields read or written by in-network ANFs are placed in a constant-time-accessible *public partition*, while application data remains end-to-end encrypted in a *private partition*.
- A *non-terminating, non-buffering proxy execution model*: ANFs run as soon as the public partition arrives whenever possible, and the remaining bytes are forwarded as a stream, with no protocol re-origination and no full-message reassembly.
- *Mutation-friendly packetization*: controlled slack in the public partition lets common header rewrites happen in place without re-packetizing the rest of the message.
- *Interference resolution shims*: fRPC preserves application-visible behavior when delivery logic and network policy interact in subtle ways. The compiler statically infers which delivery modules *may retry* and which ANFs are *duplicate-sensitive* or *drop-capable*, detects interfering pairs (for example, loss-based reliability interacting with a firewall that may drop), and automatically injects deduplication guards, explicit drop notifications, and delay feedback so that the compiled stack is behaviorally equivalent to a layered baseline. To the best of our knowledge, this is the first systematic treatment of an interaction that previous service mesh work has largely left unaddressed.

End-to-end, the synthesized stack reduces application latency by up to 52% and CPU usage by up to 33% compared with state-of-the-art gRPC + Envoy, while preserving the same semantics, security guarantees,

and out-of-process enforcement model.

1.3 Thesis Statement

The arguments and artifacts above are organized around a single claim.

> **Thesis statement.** For microservice applications operated within a single administrative domain, the RPC tax of today's layered, proxy-centric stacks can be substantially reduced by specifying the entire RPC communication stack—serialization, end-to-end delivery, and application network functions—once at a high level and compiling it into a flat, application-specific implementation, while preserving the behavioral, security, and operational guarantees developers rely on.

The three artifacts of the dissertation defend this claim along complementary axes. MeshInsight establishes empirically that the layered stack and the resulting proxy architecture are the sources of the tax and identifies the components of that tax. AppNet shows that a high-level, expressive specification of ANFs admits behavior-preserving optimization of application network functions. fRPC closes the loop by extending the specification-and-compilation approach to serialization and delivery, synthesizing a flat RPC communication stack rather than merely optimizing the proxy layer.

1.4 Thesis Contributions

This dissertation makes the following contributions.

Naming and quantifying the RPC tax.

- A definition of *the RPC tax*: the latency, CPU, and engineering-complexity costs that today's layered communication stack and out-of-process policy enforcement add to every microservice call.
- **MeshInsight**, a cost model for service mesh proxies that breaks proxy work into small components and predicts end-to-end overhead for real service mesh deployments without re-running full experiments.

Application Defined Networks as a model.

- The *Application Defined Networks (ADN)* model: compiling an application network stack from a per-application specification rather than assembling it at runtime from general-purpose layers.
- A design argument that separates what ADN must preserve—language independence, security, observability, and out-of-process policy enforcement—from the general-purpose runtime machinery it can remove inside a single administrative domain.

Compiling application network functions.

- **AppNet**, a high-level programming abstraction and compiler for ANFs in which each function is a chain of stateful elements with explicit match-action rules and state.
- A symbolic-equivalence checker for stateful ANFs that decides whether two candidate configurations produce the same observable behavior on any stream of RPCs.
- A way for developers to choose weaker consistency for state and observations when their policies allow it, letting the compiler use more optimizations safely.

Synthesizing flat communication stacks.

- **fRPC**, a compiler that turns an RPC schema, an ANF pipeline, and a delivery pipeline into a complete application-specific RPC stack.
- Policy-aware serialization and selective encryption: a wire format in which the fields used by in-network policy are placed in a public partition and the application payload remains end-to-end encrypted in a private partition.
- A non-terminating, non-buffering proxy execution model, including packetization support for in-place header rewrites.
- A compiler pass that detects when delivery logic and policy logic can interfere, then inserts deduplication, explicit-drop, and delay-feedback shims to preserve the behavior of the layered baseline.

Open artifacts.

- Open-source releases of MeshInsight, AppNet, and fRPC, available at `github.com/appnet-org`.

1.5 Thesis Organization

The remainder of this dissertation is organized as follows.

- **Chapter 2** (MeshInsight) measures the RPC tax in today's service meshes and presents a cost model that explains where the overhead comes from.
- **Chapter 3** (Application Defined Networks) introduces the ADN paradigm, relates it to Application Layer Framing, network programming, and service meshes, and explains why those ideas do not by themselves solve the RPC tax.
- **Chapter 4** (AppNet) presents a high-level model for ANFs, the symbolic-equivalence check that makes optimization safe, and the compiler that turns AppNet programs into optimized application networks.
- **Chapter 5** (fRPC) presents the compiler that turns an RPC schema, an ANF pipeline, and a delivery pipeline into a flat, application-specific RPC stack, including the wire format, proxy execution model, and interference-resolution shims.
- **Chapter 6** (Future Work) discusses how ADN could extend to kernel datapaths, SmartNICs, and programmable switches, and describes what the framework does and does not solve.

1.6 Bibliographic Notes

This dissertation builds primarily on the following first-author papers.

- Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, Ratul Mahajan. *Dissecting Overheads of Service Mesh Sidecars* (MeshInsight). ACM Symposium on Cloud Computing (SoCC), 2023. *[Chapter 2]*
- Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, Danyang Zhuo. *Application Defined Networks*. ACM Workshop on Hot Topics in Networks (HotNets), 2023. *[Chapter 3]*

- Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, Nikola Bojanic, Jingrong Chen, Gilbert Bernstein, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, Danyang Zhuo. *High-level Programming for Application Networks*. USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2025. *[Chapter 4]*
- Xiangfeng Zhu, Yang Zhou, Yuyao Wang, Xiangyu Gao, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, Danyang Zhuo. *Rethinking RPC Communication for Microservices-Based Applications*. ACM Workshop on Hot Topics in Operating Systems (HotOS), 2025. *[Chapters 3 and 5]*

Chapter 2

Dissecting the RPC Tax: A Compositional Study of Service Meshes

The previous chapter argued that the layered stack used for microservice communication imposes an *RPC tax* on every call, and that this tax is a design choice rather than an inherent cost. Before redesigning the application network, this dissertation must establish that claim empirically. The empirical foundation must do two things at once. First, it must show that today's stack is expensive enough to justify reconsidering a substrate that the cloud industry has only recently standardized. Second, and more importantly, it must explain *why*: which components of the stack pay the tax, how their costs depend on workload and configuration, and which costs would survive in any reasonable redesign. Without that decomposition, redesign is guesswork.

This chapter contributes that foundation through **MeshInsight**, a tool that quantifies and decomposes the overhead of service mesh sidecars as a function of service mesh configuration and application workload. Its central claim is that the cost of a service mesh is not a single number, but a *composition* of independent component costs. Once those components are modeled, MeshInsight can both account for the headline overheads observed in practice and predict how they will change as configurations, workloads, and hardware platforms change. The results confirm quantitatively what the introduction asserted qualitatively: a production-grade Istio service mesh using Envoy sidecars increases application latency by up to 269% and consumes up to 163% more virtual CPU cores on representative microservice benchmarks; the dominant cost component de-

pends on the proxy mode (protocol parsing in HTTP/gRPC proxies, IPC and socket writes in TCP proxies); and the cost varies by orders of magnitude across applications even under a fixed configuration. MeshInsight focuses on the sidecar data-plane architecture used by Istio and Envoy, but its component-based approach can be extended to other service mesh proxies and deployment models [80, 90, 119].

MeshInsight is available at `github.com/UWNetworksLab/meshinsight`.

2.1 Why a Tool Like MeshInsight Is Needed

Service meshes [13, 29, 44] have become the de facto communication substrate of modern microservice applications. The Cloud Native Computing Foundation's surveys report that around 90% of organizations running cloud-native workloads either already use a service mesh in production or are actively evaluating one [61]. Their popularity is easy to explain: a service mesh externalizes the common cross-cutting concerns of microservice communication—traffic management, security, observability, and reliability—into a uniform infrastructure layer that an organization can manage centrally and enforce consistently across its microservices.

The central operational concern for adopting a service mesh is the performance cost of introducing this new infrastructure layer. Previous research studies, vendor benchmarks, and industry blog posts routinely describe service meshes adding substantial latency and CPU costs [57, 81, 96, 107]. Yet the only practical way users can learn that cost today is a black-box comparison: run the same application with and without the service mesh, then measure the increase in latency or CPU usage.

Why black-box benchmarking is not enough. This black-box methodology—comparing runs with and without a service mesh—is the de facto standard for the existing literature on service mesh performance [81, 87, 96, 107, 117]. It yields a single number per (application, configuration, environment) triple. Such a number can confirm that overhead exists and quantify its magnitude, but it is fundamentally inadequate as a guide to reducing that overhead, for three reasons.

First, the operational space is combinatorial. A service mesh has at least three independent axes of configuration: the proxy mode, meaning the protocol layer the sidecar terminates and parses (e.g., TCP, HTTP, or gRPC); the set and order of network functions (filters, in Envoy's terminology) configured in

the sidecar (40+ built-in filters in Envoy [23], plus arbitrary Lua [35] and WebAssembly [43] extensions); and the application's workload (which microservices call which, with what call graph, message sizes, and request rates for each pair of communicating microservices). As a result, the configuration space scales combinatorially. Envoy's own FAQ documentation makes the same point: when asked how fast Envoy is, it answers that "performance depends a great deal on which Envoy features are being used and the environment in which Envoy is run," and therefore encourages users to benchmark Envoy in their own environments with production-like configurations [66].

Second, a single overhead number gives no insight into why. If a benchmark reports that the service mesh adds a few milliseconds to latency and consumes one and a half additional CPU cores, the developer learns the numbers but not how that cost relates to the functionality being provided. A service mesh feature may be expensive because it performs work the application genuinely needs, because it is implemented inefficiently, or because it is enabled in a context where its benefits are marginal. Without separating these cases, the developer cannot tell whether the overhead is the necessary price of a useful capability, a cost that can be reduced by changing the service mesh configuration, or a signal that the functionality itself is not worth deploying for that workload.

Third, the same opacity hampers service mesh developers. The community has proposed many ways to reduce service mesh data-plane overhead, including kernel offloads via eBPF [2, 107], per-host proxies [77, 80], and zero-copy socket APIs [36]. Each of these techniques targets a particular part of the data path. Yet the main way to determine an optimization's effect on a service mesh is still to implement it, integrate it into the service mesh data path, and measure the resulting system on realistic application workloads. This build-first approach can require substantial engineering effort before the optimization's end-to-end value is known. It also gives little guidance about whether an optimization that improves one data-path component will meaningfully reduce the overall overhead of a real microservice deployment.

The compositional alternative. MeshInsight replaces a single overhead number with a compositional model of sidecar proxy work. It decomposes the data path into a small set of independent components—inter-process communication with the application, socket reads and writes, event notification, protocol parsing, and per-filter processing—and characterizes each component once on a target platform. It then composes these measurements to predict end-to-end overhead for a given configuration and workload. This

reduces a combinatorial measurement space to a compact set of component measurements, while still attributing overhead to its sources and allowing the model to be reused as kernels, CPUs, and hardware offloads evolve.

The remainder of this chapter develops this idea. Section 2.2 reviews the anatomy of a sidecar to motivate the choice of components. Section 2.3 introduces the compositional model itself. Section 2.4 describes the design of the MeshInsight profiler and predictor. Sections 2.5 and 2.6 use MeshInsight to characterize the RPC tax on real microservice benchmarks and to demonstrate two practical use cases. Section 2.7 discusses limitations and implications for the wider service mesh design space. Section 2.8 reviews related work in the context of performance measurement and modeling, and Section 2.9 ties the chapter back into the dissertation's central argument.

2.2 Anatomy of a Service Mesh Sidecar

To motivate the components of the model, we first trace how a single message moves through a service mesh sidecar.

A modern service mesh places a user-space proxy—most commonly Envoy [13]—next to each microservice instance. A control plane such as Istio [29], Linkerd [44], Consul [76], or Cilium [2] configures these proxies, distributes certificates, and installs policy. Once the proxies are configured, the control plane is off the data path; each microservice-to-microservice RPC is intercepted by the caller's sidecar and again by the callee's sidecar before reaching the destination application.

The data path. The data path that a message traverses is shown in Figure 2.1. During initialization, the service mesh control plane installs iptables rules that redirect inbound and outbound traffic to the local sidecar. When an application sends a message, it executes a write system call; the kernel network stack and loopback device process the bytes and notify the sidecar; the sidecar reads the data, processes it according to its configuration, and writes it back to the kernel. The kernel then transmits the bytes out the NIC to the receiving host, where the receiving sidecar intercepts the message before it is passed to the destination application. A single logical RPC connection is thus broken into three separate transport connections: application-to-sidecar, sidecar-to-sidecar, and sidecar-to-application. In TCP mode the sidecar relays an

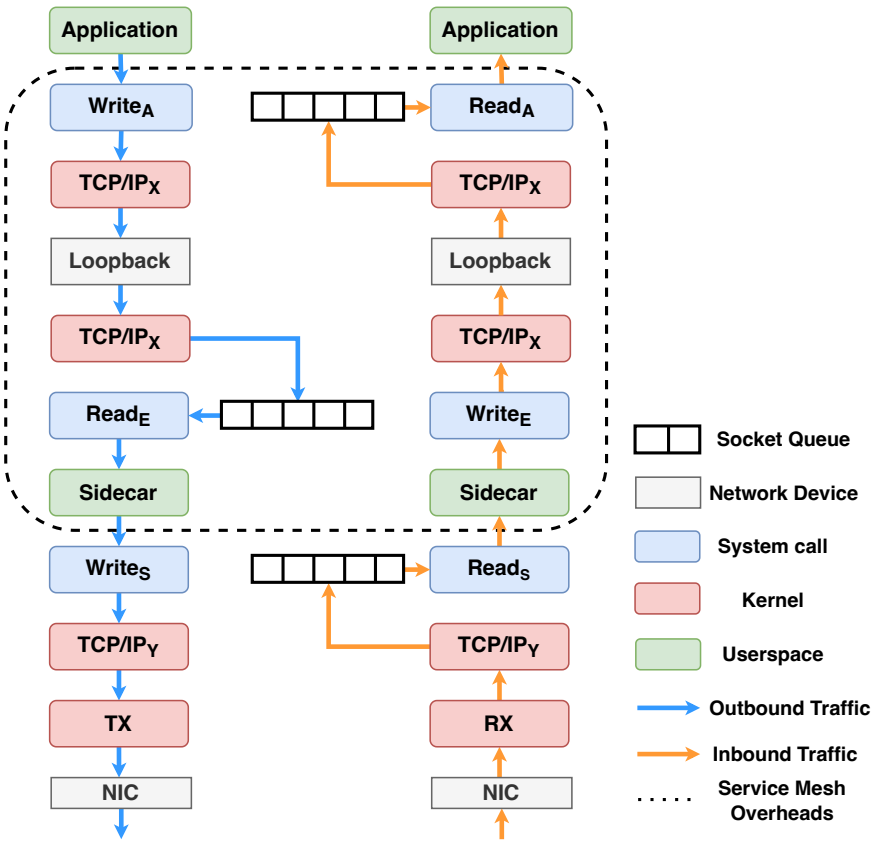


Figure 2.1: Sidecar data path for outbound (left) and inbound (right) messages. Write_A/Read_A and Write_S/Read_S denote the write/read operations of the application and its sidecar, respectively; TCP/IP_X denotes the kernel TCP/IP stack of network namespace X. The extra steps added by the sidecar are highlighted in the dashed box.

opaque byte stream and collects simple L4 telemetry; in HTTP and gRPC modes it additionally parses the protocol stream, applies any configured filters, and serializes the message back into the outbound stream.

Filter processing inside the sidecar. Sidecar processing is itself layered. Envoy is built around a filter-chain model: traffic is first handled according to the configured proxy mode, either as an opaque TCP stream or as parsed HTTP/gRPC messages; it is then handed to a sequence of filters that read or modify it (for example, a router filter that decides where to send it, a rate-limit filter that may throttle it, a tap filter that logs it, or custom filters written in C++, Lua, or WebAssembly). In HTTP and gRPC modes, the sidecar finally encodes and serializes the message back into the outbound stream. Envoy ships with more than 40 built-in filters, and common service mesh deployments enable several of them for specific use cases.

Component	Description
IPC	Inter-process communication between the sidecar and its application (e.g., loopback TCP).
Read	The `read` syscall and the kernel-to-user data copy.
Write	The `write` syscall, the user-to-kernel data copy, and TX-side TCP/IP processing.
Notification	I/O event-notification processing (e.g., `epoll`).
Sidecar Parsing	Protocol parsing inside the sidecar (HTTP/1, HTTP/2, gRPC).
Sidecar Filter	Filter-chain processing inside the sidecar.
Sidecar Other	Baseline sidecar processing not captured by the other components.

Table 2.1: The components in MeshInsight's compositional model. Each component is profiled as a function of message size and request rate on the platform of interest.

2.3 A Compositional Model of Sidecar Overhead

MeshInsight characterizes a sidecar by decomposing its work into a small set of components, measuring each component once on a given platform, and composing the resulting profiles to predict end-to-end overhead for a target configuration and workload. The components are listed in Table 2.1.

Why these components. The component boundaries reflect three constraints. First, each component maps to an identifiable kernel or user-space code path whose overhead can be measured in isolation. The categories in Table 2.1 follow natural sidecar processing cut points: syscalls, kernel/socket processing, protocol parsing, filter-chain processing, and other baseline sidecar work. Second, the components must be independent enough for their overheads to add without double-counting. Third, they must be coarse enough to remain useful predictors when composed; a finer decomposition would require estimating inter-component interactions that MeshInsight does not measure.

Modeling assumptions. The model rests on three explicit assumptions. *(A1)* The total latency overhead and CPU overhead of the sidecar are each the sum of the overheads of the components that the sidecar's configuration exercises. *(A2)* The latency overhead of a component is a linear function of message size s,

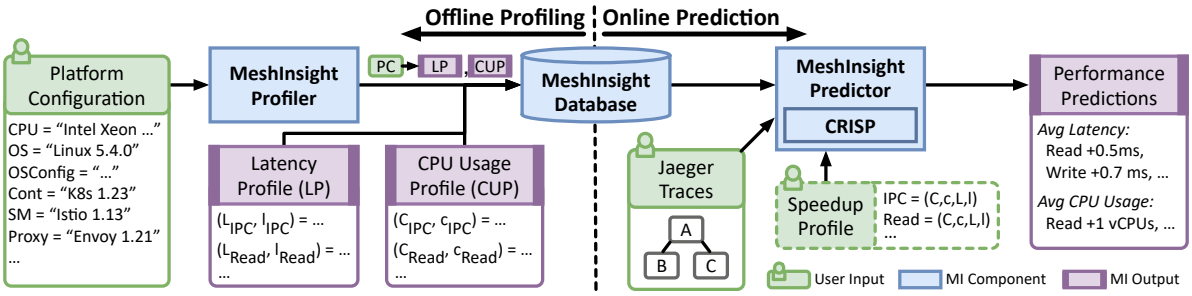


Figure 2.2: MeshInsight workflow. The offline profiler measures component costs as functions of message size and request rate and stores the resulting profiles. The online predictor consumes those profiles, a user-supplied ECG derived from Jaeger traces, and optionally a speedup profile for a hypothetical optimization, then emits predicted end-to-end latency and CPU overheads.

with a base cost L_x and a per-byte cost ℓ_x: $L_x + s \cdot \ell_x$. *(A3)* The CPU overhead of a component over a given interval is the product of the per-message CPU cost and the request rate: $r \cdot (C_x + s \cdot c_x)$, where C_x and c_x are the base and per-byte CPU costs and r is the request rate.

These assumptions are deliberately conservative. By profiling components in isolation, MeshInsight ignores contention for shared resources such as caches, CPU schedulers, and kernel queues under high load. Its predictions are therefore best-case lower bounds. We leave the study of contention-aware modeling under high utilization to future work.

The rest of this chapter shows that these choices are accurate enough to guide both application developers and service mesh designers.

2.4 MeshInsight: Design

MeshInsight operates in two phases. An offline *profiling* phase exercises each component in isolation and records per-component performance profiles for a platform. An online *prediction* phase consumes an extended call graph (ECG) of a microservice deployment and the sidecar configuration of each service, then composes the relevant profiles to estimate end-to-end overhead. The overall workflow is shown in Figure 2.2.

2.4.1 Profiling Components Offline

For each component in Table 2.1, MeshInsight builds a profile tuple (L_x, ℓ_x, C_x, c_x) for latency and CPU by exercising the component under controlled microbenchmarks and fitting the linear models from assumptions

A2 and A3.

Driving the components. The profiler pairs the sidecar of interest with a simple echo server and drives traffic with a traffic generator [45, 46]. To eliminate queueing effects and isolate per-message costs, latency profiling allows at most one outstanding request, while CPU profiling varies request rate explicitly. MeshInsight profiles the sidecar in two configurations: *(i)* a TCP, HTTP, or gRPC proxy with no additional filters, which exercises the base data path of these deployment modes; and *(ii)* the same proxy with one filter of interest, which isolates that filter's incremental cost.

Attributing time to components. All components except *Sidecar Other* correspond either to identifiable kernel/user-space functions or to an incremental filter configuration. MeshInsight measures function latency with a modified version of BCC's `funclatency` [7], an eBPF tool that probes function entry and exit with `kprobe`. CPU usage is measured with standard `perf`-style sampling [34]. *Sidecar Other* is the residual after subtracting the measured components from total sidecar overhead in the same run. MeshInsight maps probes and samples to components using the function symbol, the observed input/output direction, and the process ID of the application or sidecar being measured.

Profiling filters. Filters are often hard to attribute to a single function: Lua, WebAssembly, and built-in C++ filters all execute through runtime or dispatch layers. MeshInsight therefore measures each filter by subtracting an otherwise identical run *without* the filter from one *with* it. Because filter parameters can materially change cost (e.g., local versus global rate limiting), MeshInsight treats distinct configurations as distinct filter components.

Fitting component profiles. For each component, the profiler measures latency and CPU at five message sizes (100 B, 1 KB, 2 KB, 3 KB, 4 KB) and four request rates (25%, 50%, 75%, and 100% of the maximum throughput sustainable by the platform). It then fits the linear models from assumptions A2 and A3. This compact grid keeps profiling time low: in our empirical exploration, denser grids reduced residual fit error only negligibly. The resulting profile, (L_x, ℓ_x, C_x, c_x), is stored in a database keyed by platform (CPU model, OS, kernel version, Envoy version, kernel configuration), allowing later predictions on the same platform to amortize the profiling cost across many deployments.

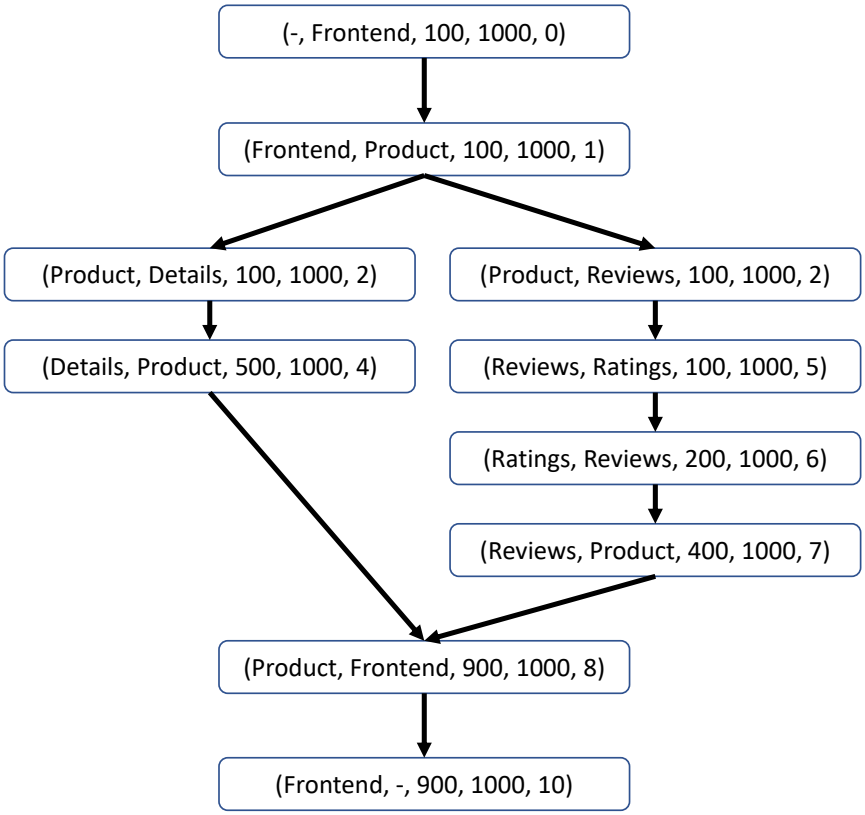


Figure 2.3: An example call graph for the Bookinfo application.

2.4.2 Predicting End-to-End Overhead

A microservice deployment is described to MeshInsight as an *extended call graph* (ECG). Formally, an ECG is a tuple (V, P, S, G) where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of microservice instances; P maps each instance to a platform (so that the right component profiles can be applied to each side of an RPC); S maps each instance to a sidecar configuration (proxy mode, filter chain configuration, IPC mechanism); and G is a directed acyclic graph of *invocations*, where each invocation $k = (u_k, d_k, s_k, r_k, t_k)$ records the calling service u_k, the called service d_k, the message size s_k, the request rate r_k, and the timestamp t_k of the invocation relative to the start of the user's request in a baseline deployment without a service mesh. Figure 2.3 shows an example of G for the Bookinfo application.

ECGs can be derived from distributed tracing infrastructure that most microservice deployments already use, augmented with measurements of message sizes and rates. The MeshInsight implementation

uses Jaeger [30] for trace collection and CRISP [136][1] for critical-path analysis. Jaeger traces capture the call graph and timestamps, while message sizes and rates are collected separately or supplied as workload assumptions. When the call graph varies dynamically across requests (e.g., a caching layer that incurs both cache hits and misses), MeshInsight accepts a set of traces and emits a probability distribution over overheads.

Predicting per-invocation overhead. Given an ECG, MeshInsight first computes the sidecar overhead of each invocation independently. The invocation $(u_k, d_k, s_k, r_k, t_k)$ involves one or two sidecars (depending on whether u_k and d_k have sidecars in the configuration S), each of which exercises a particular subset of the components in Table 2.1. The overhead of each component is computed from its profile and from the invocation's message size and rate, and the per-sidecar overhead is the sum of the exercised components.

Composing latencies and CPU. CPU overhead is additive: the CPU consumed by the service mesh on a given host across an interval is the sum of all the per-sidecar overheads on that host. Because microservice call graphs are typically not chains but DAGs, the end-to-end latency of a request is governed by the *critical path* through the DAG, and a sidecar that lies on a non-critical branch contributes nothing to end-to-end latency. To capture this, MeshInsight runs a critical-path analysis (using the algorithm described in CRISP [136]) over the ECG *after* adding the predicted per-invocation overheads, since adding a sidecar to a path may shift which path is critical. The end-to-end latency overhead is the sum of per-invocation overheads along the new critical path.

Predicting the impact of optimizations. MeshInsight's compositional model also lets service mesh developers evaluate a proposed optimization before investing the effort to implement and deploy it in a production proxy. The developer supplies a *speedup profile*: updated (L_x, ℓ_x, C_x, c_x) tuples for the components affected by the optimization. MeshInsight then re-runs the online prediction with those tuples substituted into the model and reports the resulting change in end-to-end overhead. The speedup profile can be estimated analytically, measured on a research prototype, or obtained by patching MeshInsight's profiler to capture the optimized component. Section 2.6 evaluates this what-if prediction method using two recent optimizations.

[1] CRISP is a critical-path analysis tool for microservices developed by Uber.

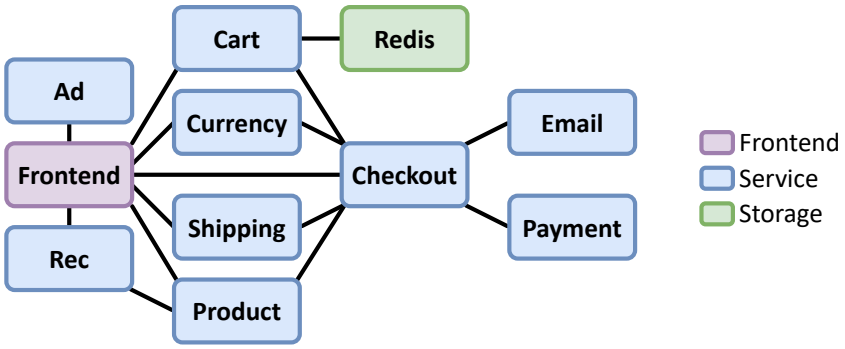


Figure 2.4: Architecture of the Online Boutique application [73].

2.5 Characterizing the RPC Tax in Real Microservice Applications

This section uses MeshInsight to characterize the RPC tax on two microservice benchmarks. It validates prediction accuracy, quantifies the headline overheads used to motivate the dissertation, and attributes those overheads to the components described in earlier sections.

Experimental setup. All experiments use CloudLab [64] hosts with two 16-core Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384 GB of memory, running Ubuntu 20.04 LTS (Linux kernel 5.4.0), Kubernetes 1.12.5, Istio 1.13.0, and Envoy 1.21.0. Turbo Boost, CPU C-states, and dynamic frequency scaling are disabled to reduce measurement variance. Each microservice runs in its own pod, with its sidecar in the same pod.

2.5.1 Two Benchmark Applications

Two open-source benchmarks are used throughout this chapter. *Online Boutique* [73] (Figure 2.4) is Google's reference e-commerce microservices application; it has 11 microservices written in a mix of Python, C#, Java, and Go, communicating over gRPC. *Hotel Reservation* [69] (Figure 2.5) is a benchmark from the DeathStarBench suite; it has 17 Go microservices that implement hotel search, profile lookup, reservation, and recommendation flows. Services in both benchmarks are replicated and deployed across CloudLab nodes to simulate real deployments.

Each benchmark is evaluated under three representative queries. For Online Boutique, the queries are `Index` (home page), `Browse_Product` (product details), and `View_Cart` (shopping cart). For Hotel

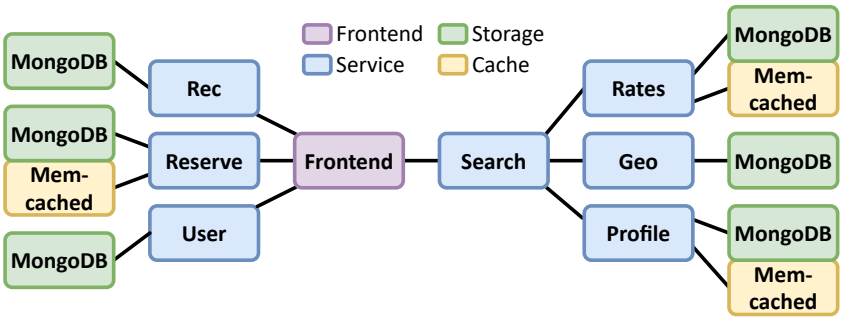


Figure 2.5: Architecture of the Hotel Reservation application [69].

Reservation, they are `User` (login), `Search` (geo-aware hotel search), and `Reservation` (booking). The queries differ in fan-out, in the mix of stateful and stateless services they invoke, and in the message sizes sent between services.

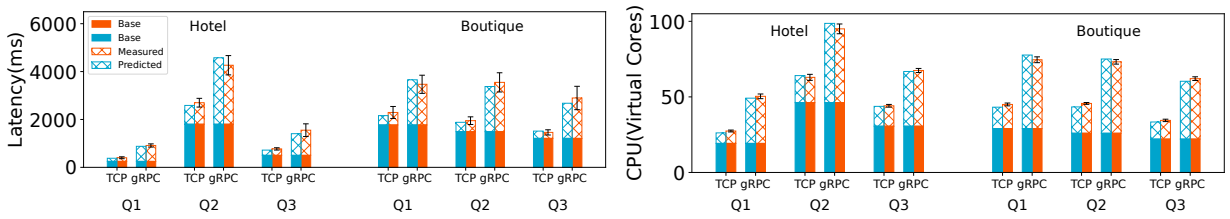
Two proxy modes are studied: *TCP*, in which the sidecars treat the gRPC stream as an opaque byte stream and apply only TCP-level functionality; and *gRPC*, in which the sidecars parse the gRPC stream and collect application-level metrics. HTTP mode is not studied on the application benchmarks because both applications use gRPC end-to-end, but HTTP mode is studied separately on a synthetic echo server in Section 2.5.3.

2.5.2 End-to-End Overhead

We derive an ECG (extended call graph) for each query, forcing a cache miss on every Memcached access so that each query follows a deterministic call graph. Message sizes in the ECG come from the application's observed traffic; most messages are small, on the order of a few hundred bytes. For each query, we set the request rate close to the maximum throughput the deployment can sustain in gRPC mode.

Across the six queries, enabling the service mesh raises end-to-end latency by 27% to 269% and CPU usage by 42% to 163% relative to running the same applications without a service mesh. The full per-query numbers, both measured and predicted by MeshInsight, appear in Figure 2.6.

Three observations matter for the rest of the chapter. First, overhead is large in both proxy modes, but larger in gRPC mode (up to 269% latency, 163% CPU) than in TCP mode (up to 61% latency, 92% CPU). Second, overhead varies by query because deeper critical paths and wider fan-out multiply per-hop sidecar costs. Third, MeshInsight's predictions track measurements closely, although they tend to under-predict



(a) Latency overhead.

(b) CPU overhead.

Figure 2.6: End-to-end latency and CPU overhead of the Istio/Envoy service mesh on the Online Boutique and Hotel Reservation benchmarks. "Base" denotes the application running without a service mesh. Both measured and MeshInsight-predicted overheads are shown; error bars are standard deviations across runs.

measured overhead, especially in gRPC mode.

These large and workload-dependent overheads motivate MeshInsight: developers need a way to understand where mesh functionality is worth its performance cost. For example, many services may only need TCP-mode forwarding, while gRPC-mode parsing can be reserved for services that require finer-grained control or visibility.

2.5.3 Where the Overhead Actually Comes From

To shed light on the sources of overhead, Table 2.2 shows the per-component breakdown of sidecar overhead in the three proxy modes (TCP, HTTP, gRPC), each measured on the same synthetic echo server with 100 B messages at 30,000 requests per second. The echo server provides a controlled workload in which all three modes can be profiled under identical traffic.

The breakdown yields several conclusions that anchor the rest of the dissertation.

HTTP and gRPC are 4–5× more expensive than TCP, primarily because of protocol parsing. Total sidecar-instance latency rises from about 41 μs in TCP mode to 165 μs in HTTP and 192 μs in gRPC; CPU demand rises from 3.35 to 9.70 and 13.36 virtual CPU cores, respectively. Most of this increase comes from *Sidecar Parsing*, which is absent in TCP mode but contributes 74–77% of latency and 63–69% of CPU in HTTP and gRPC modes. The dominant cost of an application-aware service mesh is therefore the per-message parse-process-serialize cycle executed across the proxy boundary.

This parsing cost is also redundant. After the proxy parses a message, the receiving application parses it again, because the proxy forwards only a byte stream over the sidecar-to-application TCP connection. The redundancy is a direct consequence of the sidecar data-plane architecture: the proxy is application-aware,

Component	Latency (μs)			CPU (vCPU)		
	TCP	HTTP	gRPC	TCP	HTTP	gRPC
IPC	12.17 (30%)	12.73 (8%)	13.40 (7%)	0.46 (14%)	0.50 (5%)	0.57 (4%)
Read	8.71 (21%)	9.19 (6%)	9.24 (5%)	0.27 (8%)	0.28 (3%)	0.31 (2%)
Write	14.11 (34%)	13.57 (8%)	14.91 (8%)	0.48 (14%)	0.49 (5%)	0.57 (4%)
Notification	1.27 (3%)	1.32 (1%)	1.28 (1%)	0.26 (8%)	0.27 (3%)	0.26 (2%)
Sidecar Parsing	—	122.07 (74%)	147.03 (77%)	—	6.07 (63%)	9.26 (69%)
Sidecar Other	4.83 (12%)	6.29 (4%)	5.81 (3%)	1.88 (56%)	2.09 (22%)	2.39 (18%)
Total	41.09	165.17	191.67	3.35	9.70	13.36

Table 2.2: Contribution of each component to the overhead of a sidecar instance in TCP, HTTP, and gRPC modes, measured on a synthetic echo server with 100 B messages at 30K requests per second. Numbers include both inbound and outbound paths and exclude per-filter costs. Percentages in parentheses are the share of the sidecar-instance total in that mode.

but it remains outside the application's process and cannot pass along the structured message it has already decoded.

The extra local connection dominates TCP-mode overhead. In TCP mode, where there is no parsing, IPC accounts for 30% of latency and Write accounts for another 34%. Because Write includes TX-side TCP/IP processing on the loopback interface, the two components together capture much of the cost of inserting a local TCP connection between the application and the proxy. This cost remains as long as an out-of-process sidecar communicates through TCP loopback. For the same reason, recent optimizations that reduce IPC overhead have a larger effect in TCP mode than in HTTP or gRPC mode, where protocol parsing dominates the total.

Notification cost is small. `epoll`-based notification consumes only 3% of TCP-mode latency and stays near 1% in HTTP and gRPC modes. Asynchronous event notification is therefore not a major contributor to the RPC tax, so avoiding `epoll` alone is unlikely to justify a substantially more complex proxy design.

Latency and CPU are not interchangeable. Component shares differ substantially between the latency and CPU columns. *Sidecar Other*, for example, contributes only 3–12% of latency but 18–56% of CPU. The dispatcher's per-message work can be hidden behind waiting points such as `epoll` and network I/O, but the CPU cycles are still spent. Optimizations that reduce latency may therefore leave CPU demand unchanged, and CPU-saving optimizations may not move tail latency.

Filter	Latency (μs, % of HTTP base)	CPU (vCPU, % of HTTP base)
Fault Injection	5.74 (3.5%)	0.20 (1.9%)
Rate Limit	8.19 (5.0%)	0.21 (2.0%)
Tap (file)	156.09 (85.0%)	2.95 (30.4%)
Lua	80.59 (43.9%)	3.18 (30.2%)
WebAssembly	26.30 (14.3%)	0.69 (6.6%)

Table 2.3: Latency and CPU overhead of five representative filters atop the HTTP-mode baseline. Percentages are relative to the corresponding HTTP-mode baseline used in the filter experiment.

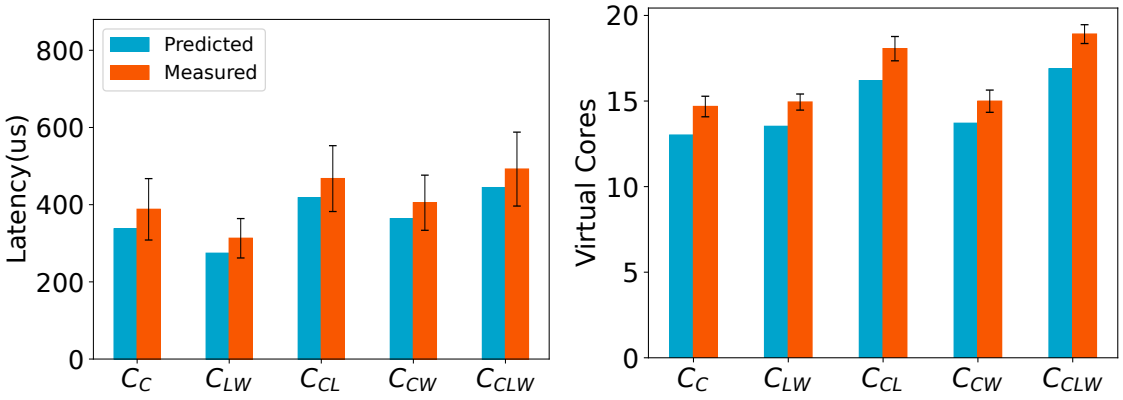
2.5.4 The Cost of Filters

Service meshes are valued not only for the base communication infrastructure but also for the application network functions the proxy can apply to every call. To establish a baseline for these functions, MeshInsight profiles five representative filters that exercise all three filter paths in Envoy: *Fault Injection* (built-in C++), *Rate Limit* (built-in C++), *Tap* configured to log to a file (built-in C++ with file I/O), a no-op *Lua* script, and a no-op *WebAssembly* module. The same synthetic workload as Section 2.5.3 is used (100 B messages, 30K req/s, HTTP mode).

Built-in C++ filters that do not touch external state—Fault Injection and local Rate Limit—cost only a few percent atop the HTTP baseline. Filters that issue I/O (Tap) or are written in higher-level languages (Lua) are much more expensive; WebAssembly lies in between. Even a *no-op* Lua filter substantially increases CPU usage, which is a cautionary result for scripted policies at scale.

The next experiment tests how filter costs combine. MeshInsight predicts that filter costs are additive on top of the base sidecar cost (A1). Direct measurements of five combinations—the three C++ filters; Lua plus WebAssembly; Lua plus the three C++ filters; WebAssembly plus the three C++ filters; and all five together—show that the additive predictions are quite accurate, as shown in Figure 2.7. Operators can therefore estimate the cost of enabling another filter from its standalone profile, rather than re-benchmarking the whole stack.

Chapter 4 returns to filters from a different angle: by treating network functions as high-level programs, AppNet can fuse multiple filters and remove much of the per-filter base cost measured in Table 2.3.



(a) Latency overhead.

(b) CPU overhead.

Figure 2.7: Predicted and measured overheads for different filter combinations on the HTTP-mode synthetic workload. The combinations include the three C++ filters, Lua plus WebAssembly, Lua plus the C++ filters, WebAssembly plus the C++ filters, and all five filters together.

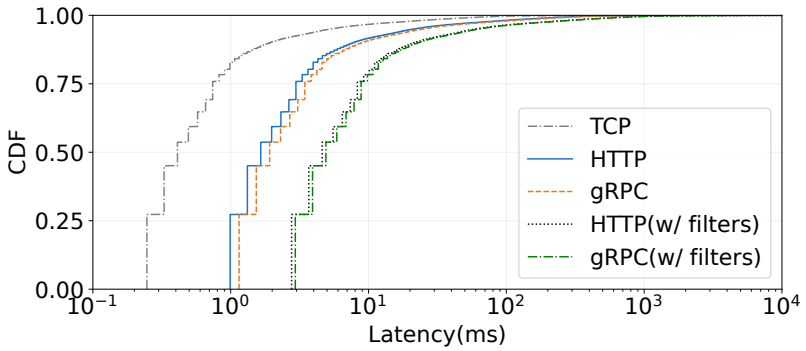
2.6 Using MeshInsight in Practice

Beyond characterizing the RPC tax, MeshInsight supports two practical use cases. Application developers can ask, "How will a given service mesh configuration change my service's end-to-end latency and CPU usage?" Service mesh developers can ask, "How much will my optimization improve performance in production?" This section answers both questions on real workloads.

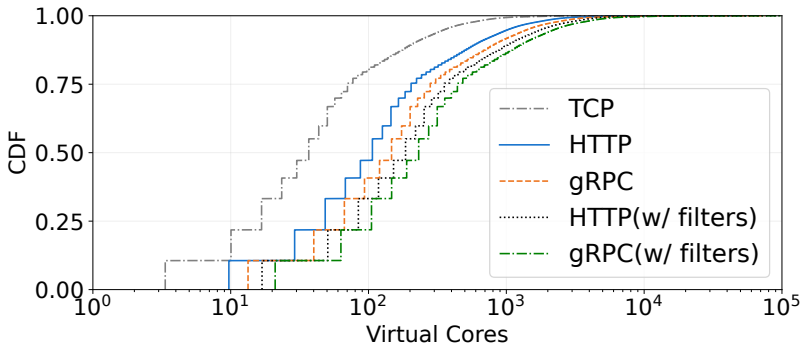
2.6.1 Helping Application Developers Estimate Deployment Impact

A single overhead number hides the fact that the impact of a service mesh configuration depends on the application's call graph, message sizes and rates, and required filters. Given an application's ECG and the proxy mode and filters it plans to deploy, MeshInsight predicts the resulting change in end-to-end latency and CPU usage.

The Alibaba trace study. To demonstrate this at scale, MeshInsight is run over the Alibaba microservice trace dataset [98], which contains more than 20 million call graphs collected over seven days from a production Alibaba cluster. The traces include a long tail of complex call graphs: roughly 10% have more than 40 microservices and the largest ones have thousands. From this dataset, one million call graphs are randomly sampled. Since the Alibaba traces do not record message sizes or rates, conservative defaults are



(a) Absolute latency overhead.



(b) Absolute CPU overhead.

Figure 2.8: MeshInsight's predicted latency and CPU overheads for one million Alibaba call graphs across five service mesh configurations. Within any one configuration, overheads span four orders of magnitude across call graphs; across configurations, the median overhead grows by an order of magnitude as more mesh features are enabled.

used (100 B messages and 30K requests per second, both representative of microservice workloads).

For each call graph, MeshInsight estimates the overhead of five service mesh configurations: TCP without filters, HTTP without filters, gRPC without filters, HTTP with the five filters from Section 2.5.4, and gRPC with those same five filters. The resulting distributions of latency and CPU overhead, shown in Figure 2.8, exhibit two key takeaways.

Within any one configuration, overhead varies by orders of magnitude across call graphs. In TCP mode alone, latency overhead ranges from 0.2 ms to 100 ms, and CPU overhead from 3 to 1000 virtual CPU cores, across the Alibaba call graphs. The variation tracks critical-path depth and fan-out. This wide variation means that a deployment-wide overhead estimate will overstate the impact for some call graphs while understating it for others.

Across configurations, median overhead grows by an order of magnitude. Median latency overhead is 0.2 ms in TCP mode without filters and 2 ms in gRPC mode with the five filters; the 75th percentile rises from 1 ms to 10 ms. CPU follows the same pattern, moving from a median of about 20 virtual CPU cores in TCP mode to over 200 in gRPC-with-filters mode. The same application can therefore see very different service-level objective (SLO) and resource effects depending on the concrete mesh features enabled.

2.6.2 Helping System Developers Triage Optimizations

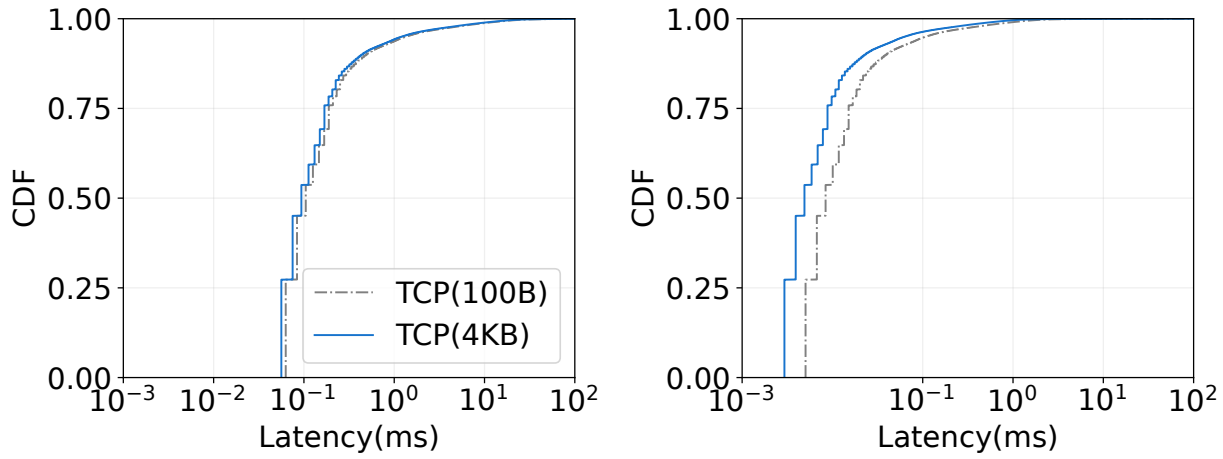
The same compositional model lets service mesh developers triage proposed optimizations before paying the cost of implementing and deploying them in a production proxy. To demonstrate this, MeshInsight evaluates two Linux kernel features proposed as ways to lower service mesh overhead.

Unix domain sockets in place of TCP loopback. The default Envoy/Istio setup uses a TCP loopback connection between an application and its sidecar. As Table 2.2 shows, IPC is 30% of TCP-mode latency and 14% of TCP-mode CPU, and the cost is paid on both sides of an RPC. Unix domain sockets [42] offer the same socket API but avoid the full TCP/IP stack: the kernel copies data into kernel space and places the resulting buffer directly on the receiver's socket queue.

To evaluate each optimization without re-implementing Envoy, we built a 676-line C++ sidecar that mirrors Envoy's data-path architecture, enabled the feature under study, re-ran the profiler against this prototype, obtained new (L_x, ℓ_x, C_x, c_x) tuples for the affected components, and fed those tuples back into MeshInsight as speedup profiles.

Zero-copy socket writes. A second significant source of sidecar overhead is the latency and CPU cost of copying data, which is embedded in the Read and Write components. It is already noticeable with 100 B messages (Table 2.2) and grows with message size. Linux has supported zero-copy TCP writes since kernel 4.14 [36]: for a socket send, applications can pin user-space buffers and receive an asynchronous signal once the kernel has consumed them, eliminating the user-to-kernel copy. Linux does not support zero-copy reads.

MeshInsight evaluates both optimizations on the same Alibaba workloads, using TCP-mode sidecars and message sizes of 100 B and 4 KB. The absolute savings in HTTP or gRPC mode would be similar, but



(a) Unix domain sockets.

(b) Zero-copy writes.

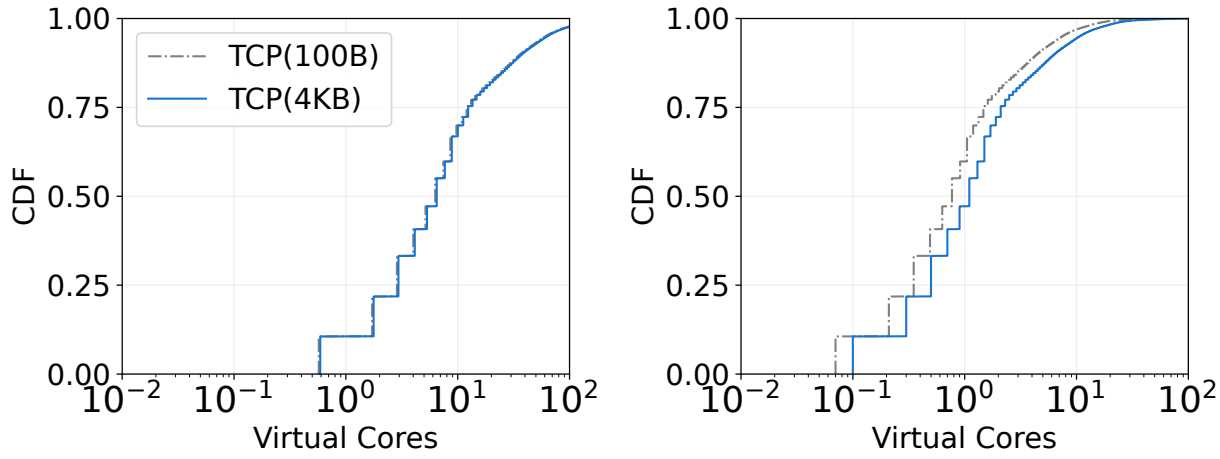
Figure 2.9: Predicted end-to-end latency overheads across the Alibaba call graphs with and without Unix domain sockets (left) and zero-copy writes (right), for TCP-mode sidecars at 100 B and 4 KB message sizes.

the relative advantage in those modes would be smaller because parsing, not IPC or data copy, dominates the total overhead.

Figure 2.9 shows the predicted end-to-end latency overheads with and without each optimization. Unix domain sockets yield a substantial improvement: the average latency reduction is 0.71 ms for 100 B messages and 0.78 ms for 4 KB messages. Because the baseline latency overhead is higher at 4 KB (4.01 ms versus 2.88 ms at 100 B), the relative reduction is smaller for larger messages, which is consistent with IPC contributing less to total overhead as message size grows. In percentage terms, Unix domain sockets reduce average latency overhead by about 27% and average CPU overhead by about 18%. This result helped motivate Unix domain socket support in service mesh deployments; MeshInsight quantified the gain before the engineering cost was paid. Zero-copy writes, by contrast, bring negligible latency improvement. The optimization adds overhead for buffer lifecycle management [36]; for the small messages typical of microservice RPCs, the extra signaling syscalls cancel the savings from avoiding the copy.

Figure 2.10 shows the corresponding CPU results. For 100 B messages, Unix domain sockets reduce average CPU overhead from 86 to 71 virtual CPU cores; for 4 KB messages, the reduction is from 107 to 91 virtual CPU cores. Zero-copy writes barely move CPU overhead, from 86 to 84 virtual CPU cores at 100 B. Zero-copy is useful for bulk transfers, but not for the workload regime that drives microservices.

The methodological lesson is that the compositional model separates optimizations that improve end-to-



(a) Unix domain sockets.

(b) Zero-copy writes.

Figure 2.10: Predicted end-to-end CPU overheads across the Alibaba call graphs with and without Unix domain sockets (left) and zero-copy writes (right), for TCP-mode sidecars at 100 B and 4 KB message sizes.

end performance from those that look attractive in isolation but vanish in real workloads. This separation is difficult with the black-box methodology that MeshInsight replaces.

2.7 Limitations and Discussion

MeshInsight makes deliberate trade-offs to keep its model tractable. Those trade-offs matter both for using the tool and for understanding why later chapters build on top of it.

The model predicts a lower bound. By treating components in isolation, MeshInsight ignores contention effects: cache pressure between simultaneous filters, scheduler interactions between busy sidecars on the same host, and queue buildup under saturating load [91, 94]. These effects raise actual overhead above prediction, especially under high load. At moderate loads, the lower-bound predictions track measurements to within 10% on the benchmark applications; near saturation, they should be read as a floor rather than a forecast. Tail latency is outside the model's scope.

Coverage of service mesh architectures. MeshInsight as developed here targets the Istio/Envoy sidecar architecture. Adjacent deployments replace the per-pod sidecar with a shared proxy, either as an ambient mesh waypoint [80] or as a middlebox-style remote proxy [119]. This chapter does not evaluate those

deployments directly. The relevant components, however, carry over: applications still communicate with a proxy, the proxy still reads, parses, filters, and writes traffic, and the resulting path still composes per-component costs. What changes is the placement and multiplicity of those components. IPC may become a host-local hop, a node-local tunnel, or a network hop to a remote proxy; Read and Write profiles must be re-measured for that path; and shared proxies introduce contention and batching effects that are outside MeshInsight's single-sidecar lower-bound model.

This distinction matters because moving the proxy is not the same as removing the application-level work. Ambient and middlebox-style meshes can reduce per-pod resource cost and change where overhead appears, but they still perform the parse-filter-serialize cycle when they expose application-level policy. Section 2.5.3 shows that this protocol processing is the dominant cost of an application-aware service mesh. Reducing that cost requires a wire format and proxy execution model with no full parse in the common case. That is the contribution of Chapter 5.

What MeshInsight does not do. MeshInsight is a measurement and modeling tool, not an optimizer: it estimates overhead for existing and hypothetical configurations, but it does not choose configurations on an operator's behalf. Nor does it replace at-scale benchmarking under production load. Its predictions are workload-specific lower bounds, so operators should validate candidate configurations with production-shadow tests before deploying them. Its contribution is narrower but important: MeshInsight turns black-box overhead into component-attributed overhead, making principled redesign possible.

2.8 Related Work

This chapter sits at the intersection of four threads of prior work.

Microservices and service meshes. Recent work has addressed many microservice challenges: tail-latency management [69, 85, 108, 120], fault tolerance [84], debugging and observability [70, 92], isolation [54], energy efficiency [97], and correctness [104]. Service mesh performance has received less systematic attention. Public studies largely use black-box methodology [81, 87, 96, 107, 117] and report headline numbers without component attribution. MeshInsight appears to be the first compositional study

of service mesh overhead, providing per-component attribution and configuration-aware prediction.

Host network stack performance. Another line of work examines the host network stack. Peter et al. [105] decompose Linux networking overhead and motivate kernel bypass; Neugebauer et al. [103] study PCIe's contribution to end-host performance; Farshin et al. [67] examine Intel's Data Direct I/O technology; and NSight [74] uses Intel Processor Tracing to diagnose nanosecond-scale network latencies. MeshInsight also decomposes host-network cost, but targets the layer-7 sidecar, which traverses the host stack multiple times per RPC and adds user-space protocol processing. Its findings argue that service mesh optimization should target layer-7 processing, not only the layer-3/4 data path.

Performance of online services and proxies. Performance modeling of online services has a long history. Stewart and Shen [121] use linear models to profile online services as black boxes; later work extends such modeling to multi-tier and microservice workloads. MeshInsight differs by breaking the proxy into components and by supporting "what-if" analysis through speedup profiles. Prior proxy studies focus largely on layer-4 [62, 77, 93, 101]; sidecars are layer-7 proxies with a richer cost structure.

IPC performance. Inter-process communication is a longstanding operating-systems topic. Liedtke [95] and the L4 family [65] reduced IPC cost through microkernel design; later studies [78, 124] systematized Linux IPC performance. MeshInsight builds on this line of work and shows that, in sidecar-based service meshes, the IPC mechanism (TCP loopback versus Unix domain sockets) has measurable end-to-end impact on real microservice workloads.

2.9 Summary and Implications for the Rest of the Dissertation

This chapter establishes the empirical foundation for the rest of the dissertation. By replacing black-box measurement with a compositional model of sidecar overhead, MeshInsight produces three findings.

Finding 1: The RPC tax is real and large. On representative microservice benchmarks, a production-grade Istio/Envoy service mesh increases latency by up to 269% and CPU usage by up to 163% relative to the baseline application. On the Alibaba trace, overheads vary by orders of magnitude across applications and configurations.

Finding 2: Protocol parsing dominates in HTTP/gRPC mode. The dominant cost component of HTTP or gRPC sidecars—the modes that expose application-level metadata for enforcing application policies—is per-message protocol parsing and serialization, contributing 74–77% of latency overhead and 63–69% of CPU overhead. The proxy parses messages to apply policy, serializes them because the application expects a wire-format stream, and pays both costs on every hop. Existing optimizations that reduce kernel-side or hardware-side overhead do not remove this cost—an architectural change is needed.

Finding 3: IPC and socket writes dominate TCP-mode latency, and IPC is optimizable. When the service mesh operates as a TCP proxy, the largest latency costs are inter-process communication between the application and its sidecar and the sidecar's socket writes. Replacing TCP-loopback IPC with Unix domain sockets reduces average latency overhead by 27% and CPU overhead by 18%, but this is a constant-factor improvement rather than a change in architecture.

These findings shape the dissertation's response. Chapter 3 draws the architectural conclusion: the tax is not a constant-factor problem to optimize away one component at a time, but an architectural choice to revisit. Chapter 4 attacks the application network function cost from Section 2.5.4 by treating the filter chain as a high-level program that a compiler can fuse, reorder, and place with formal equivalence guarantees. Chapter 5 attacks protocol parsing directly by synthesizing a policy-aware wire format and a non-terminating proxy execution model.

Chapter 3

Application Defined Networks

The previous chapter established empirically that today's general, layered communication stack imposes an *RPC tax* on every microservice call, and that the architecturally important part of that tax comes from protocol termination: every proxy must parse, buffer, decrypt, and deserialize the entire stack before it can apply any application network policy. This chapter takes the architectural step that the measurement chapter sets up. It argues that, in microservice deployments, application networks should not be assembled from interchangeable, general-purpose layers, and proposes a different design principle: **Application Defined Networks (ADN)**. In an ADN, the entire RPC communication stack—serialization, end-to-end RPC delivery, and the application network functions (ANFs) that the policy and observability infrastructure runs on—is *specified once at a high level* and *compiled* into a flat, optimized, application-specific implementation. ADN is not an argument against abstraction or modularity. It is an argument about where abstraction belongs: above the implementation, in a semantic specification the compiler can analyze, rather than in a fixed stack of runtime layers every RPC must traverse.

This chapter is based on two pieces of prior work that establish the model in stages: *Application Defined Networks* [138], published at HotNets '23, which first articulated the case for compiling per-application network functions from a high-level specification; and *Rethinking RPC Communication for Microservices-Based Applications* [141], published at HotOS '25, which extends the argument from application network functions to the entire RPC communication stack, including serialization and delivery. The chapter combines the arguments of both papers in light of the measurements in Chapter 2 and the artifacts contributed by the

rest of the dissertation. It is best read as a design argument: it defines what ADN is, what it commits to, where it sits in the space of network programming, and what an ADN compiler and controller must therefore do. The mechanics of the compiler itself are developed in Chapters 4 and 5.

3.1 The RPC Communication Component

An RPC communication component is the part of a microservice system that turns a typed procedure call in one service into bytes delivered to another service, and then turns the response bytes back into a typed result. In today's deployments, this component is usually assembled from three separately chosen pieces: a serialization format such as Protobuf, an RPC delivery stack such as gRPC over HTTP/2 over TCP, and a collection of application network functions (ANFs) such as load balancing, access control, compression, tracing, and rate limiting. ANFs often need to inspect application fields, while the fields they need are buried inside a serialized payload carried by general-purpose delivery protocols and protected by security boundaries.

Serialization. Serialization defines the application-level shape of an RPC on the wire. Given a schema, it maps typed request and response objects into bytes and maps bytes back into objects at the receiver. In the conventional stack, serialization is treated as an endpoint concern: the application library serializes before handing bytes to the RPC library, and deserializes only after the full RPC has been delivered. Common serialization formats include Protobuf [39], FlatBuffers [16], and Cap'n Proto [5].

RPC delivery. RPC delivery provides the end-to-end communication semantics of the call: addressing, framing, flow control, reliability, ordering, congestion behavior, and connection management. Previous work sometimes calls this component the RPC transport. The common choice in today's microservice deployments is gRPC layered on HTTP/2 and TCP. That stack provides a standardized, interoperable interface, but it also imposes uniform reliable, ordered byte-stream semantics on endpoints whose needs may differ, and it couples all higher-level RPC streams to TCP's behavior. Security is usually layered onto this path through TLS or mTLS, which authenticates peers and protects confidentiality and integrity on each connection.

Application network functions (ANFs). ANFs are the transformations that run on RPCs outside the application logic. They fall into four broad categories: policy enforcement, such as rate limiting or admission control; security, such as authentication and authorization; traffic management, such as load balancing, retries, and circuit breaking; and observability, such as tracing, metrics, and telemetry. Many ANFs require inspection of application data, and some are stateful, maintaining counters, connection state, rate-limit buckets, or other policy state across RPCs. In the layered model, ANFs are implemented within a separate user-space proxy (e.g., Envoy [13]) above a fixed protocol stack, so they can act only after the proxy terminates and parses enough of the protocol.

3.2 The Cost of Generality

To make the argument concrete, it helps to walk through what a developer actually does today when they need a piece of application-specific network behavior. The example is intentionally small, and the point is not that the work in the example is unmanageable, but that the same shape of work repeats on every microservice in a real deployment.

A working example. Consider an application with two microservices, A and B, where A is the client and B is the server. Service B is replicated into instances B_1 and B_2 and exposes two RPC endpoints: a read-mostly `lookup` endpoint that can tolerate occasional loss or reordering because the caller can retry, and an `update` endpoint whose side effects require reliable, ordered delivery. The developer wants the network to do four things: (1) deliver each RPC with the endpoint-specific reliability and ordering semantics it actually needs; (2) load balance RPCs from A across B_1 and B_2 based on the object identifier carried in the request; (3) compress and decompress the RPC payload between A and the chosen B_i; and (4) enforce access control based on a user identifier in the RPC and the operation being requested.

Today, the developer reaches for the layered stack, typically using gRPC [18] or an equivalent RPC library [3, 4]. They modify the application to embed the relevant fields—an object identifier and a user identifier—in HTTP headers. Choosing gRPC also commits them to HTTP/2 over TCP/IP, so both `lookup` and `update` inherit the same reliable, ordered byte-stream delivery even when only one endpoint needs it. HTTP/2 multiplexing can then force an endpoint that would otherwise tolerate relaxed delivery to wait

behind unrelated bytes lost earlier in the TCP stream. They configure a proxy [13, 22, 37] as a sidecar next to each instance, using `iptables` to redirect the application's outbound traffic through the proxy. The proxy terminates the TCP connection, parses the HTTP headers, applies the application policy, possibly modifies or drops the message, and re-creates a fresh TCP connection on the other side. The same architecture is repeated on every hop in the call graph.

The cost of generality. This architecture is easy to adopt and accelerates development: teams get a common proxy, a common control plane, and a library of reusable policies without rebuilding the communication stack for each application. Its main weakness is the same property that makes it convenient. Because the stack is general-purpose, every RPC must pass through layers, protocol boundaries, and proxies that were designed for broad interoperability rather than for the specific semantics of a particular application. That generality shows up in four ways.

High overhead. Each message is encoded and decoded multiple times by multiple components on multiple hosts, and the dominant cost—protocol parsing in HTTP/gRPC mode—is structural, as Chapter 2 established. Field reports, previous work, and the measurements in this dissertation consistently agree on the order of magnitude [81, 96, 107, 115, 139].

Coarse RPC delivery. The end-host stack provides delivery through a small number of standard transports. Usually that choice is too coarse for microservices: a read endpoint may prefer lower latency and tolerate occasional loss, while a write endpoint may require reliable, ordered delivery. Today's stack forces both through the same gRPC/HTTP/2/TCP machinery.

Non-portability. Policies written against a fixed protocol stack are difficult to move. Programmable kernels [2, 79], SmartNICs [62, 93], and programmable switches [53, 101] could execute parts of the application network, but the fields they need are often buried behind layers they cannot parse or are too deep in the packet to reach [135]. The proxy is also responsible for too many tasks at once: protocol termination, parsing, policy execution, state management, and re-origination are bundled together, making selective offload difficult.

High development burden. Even when the right execution substrates are available, today's stack leaves the hard mapping decisions to developers. They must decide which available in-network processors run which ANFs; in what order those ANFs execute; which fields must be exposed at each point; and how state

is partitioned, replicated, and kept consistent across those placements. This burden becomes most visible when the desired behavior goes beyond the simple cases the stack already anticipates—load balancing, authentication, rate limiting, and telemetry. Policies that depend on application-specific fields, custom state, or coordination across proxies are much harder to express cleanly. Service meshes provide plug-in frameworks such as Lua and WebAssembly [139], but those frameworks add extra overhead and still force new behavior through the proxy's fixed execution model rather than deriving placement, ordering, and state management from the application's semantics.

The case for specialization. The four pain points have the same root: a general-purpose stack is serving a workload that is, by construction, application-specific. In many microservice deployments, the services are owned by the same organization, deployed through the same pipeline, and operated within a shared trust boundary. This means the application graph, RPC schemas, delivery requirements, and policies can be known ahead of time and be tightly coupled by design. Yet today's stack treats them as if they must pass through fixed protocols, generic proxies, and generic filter APIs. It therefore carries mechanisms many endpoints do not need, while forcing application-specific behavior through interfaces not designed to express it. The architectural argument of this dissertation is that this generality is unnecessary for many microservice deployments, and that the right response is to stop paying for it on every RPC.

3.3 Application Defined Networks

We define an *Application Defined Network* as the artifact produced when an application's developer specifies the network's behavior at a high level—which application network functions it must apply to which RPCs, with what state and what consistency guarantees, and which end-to-end delivery semantics it must enforce—and a compiler synthesizes the implementation: the wire format, the proxy execution model, the encryption boundaries, and the placement of each ANF across available in-network processors. The substrate under the application, by default, provides only basic layer-2 connectivity of the kind already provided by cloud virtual networks (e.g., AWS VPCs [49]) and by overlay technologies such as VXLAN [126], between endpoints identified by flat names. Everything beyond that flat connectivity is *compiled* from the high-level network specification.

The ADN model rests on three principles. First, **separate specification from implementation:** the developer says *what* the network must do, not *how*; the compiler chooses *how*. Second, **co-design across the stack:** the compiler is responsible for the entire stack at once—serialization, delivery, ANFs—because the costs paid by today's architecture (Section 3.2) cross cleanly defined layer boundaries, and any compiler restricted to one layer would have to treat the other layers as fixed constraints. Third, **behavioral equivalence as the safety net:** the compiler is free to make optimizations (reorder and fuse ANFs, synthesize wire formats, move encryption boundaries, insert shims) only because it can verify that the compiled artifact is behaviorally equivalent to a baseline artifact with respect to the developer's specification.

These three principles distinguish ADN both from the configure-and-deploy posture of today's service meshes (which fixes the implementation and lets the operator pick parameters) and from hand-rolled, application-specific designs (which fix the implementation by hand for a single application and do not generalize). They also explain why simply widening the interfaces between existing layers is insufficient. A richer HTTP header, a larger TCP option space, or a new cross-layer hint API can expose more context, but such changes leave the runtime stack intact and can optimize only within the boundaries that stack already chose.

Two pillars. Realizing the ADN model is a substantial systems undertaking. The rest of this dissertation organizes the undertaking into two pillars, each a complete artifact, each defending one half of the architectural argument made in this chapter.

The first pillar, developed in Chapter 4, is a programming model and compiler for application network functions. **AppNet** [140] takes the cost-of-generality argument and answers it for the *policy* layer: it makes the developer's ANF specification high-level and analyzable, so that the compiler can fuse, reorder, and relocate stateful elements with formal equivalence guarantees, and so that the same specification can run on very different substrates. AppNet shows that, even when using today's layered stack, lifting the ANF layer to a compiled abstraction recovers up to 82% of the latency and 75% of the CPU that hand-written native filters consume. It does not, however, remove the fixed RPC delivery stack underneath those ANFs.

The second pillar, developed in Chapter 5, attacks the layered communication stack. **fRPC** extends the compile-from-specification idea from ANFs to the entire RPC communication path: serialization, per-endpoint delivery, and the boundary between encrypted and policy-readable bytes. The compiler emits three

artifacts: (1) a policy-aware split-partition wire format in which exactly the RPC fields that in-network ANFs read or write live in a public partition while the application payload remains end-to-end encrypted in a private partition; (2) a non-terminating proxy execution model that runs ANFs the instant the public partition is present and forwards the rest as a byte stream; and (3) shims that resolve interference between delivery semantics (e.g., loss-based reliability) and policy semantics (e.g., a firewall that may drop). Overall, the synthesized stack reduces latency by up to 52% and CPU by up to 33% compared with gRPC + Envoy, while preserving the same semantics, security guarantees, and out-of-process enforcement model that make service meshes operationally attractive in the first place.

3.4 What Goes Into the Specification, What the Compiler Decides

The central design judgment in ADN is the partition of responsibility between developer and compiler. The principle is simple: *the developer specifies what the network must do and what end-to-end behavior the application requires; the compiler decides how that behavior is implemented, where each piece runs, and how data is laid out on the wire.*

Concretely, an ADN specification names the RPC schema, the ANFs that must run on each RPC, the delivery properties each endpoint requires, and any state, consistency, placement, or trust constraints that are semantically meaningful. These are the requirements that belong to the application: what fields a policy depends on, which state must be shared, which RPCs can tolerate relaxed delivery, and which enforcement points must remain outside the application process.

Everything else is deliberately left to the compiler. The compiler may choose the wire layout, the delivery mechanism, the placement of ANFs across available processors, the execution model used by proxies or libraries, and any equivalence-preserving optimizations needed to make the processing fast. This separation is what makes ADN portable: when the target changes from sidecars to eBPF, SmartNICs, programmable switches, or some combination of them, the application's intent should not have to be re-derived by hand.

The rest of the dissertation turns this principle into mechanism. Chapter 4 makes ANF specifications analyzable enough for reordering, fusion, placement, and consistency-aware optimization. Chapter 5 extends the same compile-from-specification idea to serialization, delivery, encryption boundaries, and proxy execution. This chapter keeps the abstraction boundary in view; the later chapters supply the machinery

behind it.

3.5 A Reference Architecture

The ADN model admits more than one realization. The reference architecture used throughout this dissertation follows the familiar separation between a logically centralized control plane and a distributed data plane.

Control plane. The control plane consists primarily of a compiler and a controller. The compiler consumes an ADN specification and a description of the deployment environment: available processors, trust boundaries, and capacity limits. The compiler emits the artifacts those processors must run, while a runtime controller handles events that invalidate compile-time assumptions, such as replica changes, failures, or the arrival of a new target.

Data plane. In the simplest design, the data plane begins inside the application's RPC library. The compiler generates the endpoint-specific serialization and delivery code that the application uses to issue and receive RPCs, and it also inlines the ANFs that are safe to execute in the application's address space. This is the model used by systems such as Proxyless gRPC [90] and Meta's ServiceRouter [114]: the RPC stack remains responsible for delivery, but some of the network-function logic that would otherwise live in a proxy can run directly at the endpoint.

The same deployment can also include a small number of in-network processors:

- A per-host sidecar process, into which the compiler places ANFs that should remain out-of-process for trust, isolation, or operational reasons;
- A kernel offload, into which the compiler places simple per-packet ANFs that benefit from avoiding user-space crossings;
- A hardware offload, such as a SmartNIC or programmable switch, into which the compiler places ANFs that need line-rate execution or should run close to the network;
- A middlebox or other network processor, into which the compiler places ANFs that are naturally shared across many endpoints or should run away from application hosts.

The point of this architecture is that the placement decision is no longer embedded in the application or frozen into a generic proxy. The compiler chooses which ANFs run in the RPC library, which run in sidecars, kernels, hardware offloads, or middleboxes, which delivery code each endpoint needs, and how the wire format should expose the data those processors require.

The mechanisms that let a compiler choose among these placements, and prove that the choices preserve the specified behavior, are developed in Chapters 4 and 5.

3.6 Scoping Decisions and Boundary Conditions

A model becomes useful only when its boundary is drawn precisely. ADN's design space is the single administrative domain: the applications, deployment platform, and communication requirements are owned by one organization or by tightly integrated teams. Those are exactly the assumptions that the open Internet cannot make, and they are what make it possible to compile away standard layers inside the application network.

ADN therefore does not replace Internet protocols or erase the need for interoperability. External clients or third-party services still communicate using standard protocols through ingress and egress gateways [25].

Nor does ADN require discarding the operational value of service meshes. Tracing, policy verification, certificate management, and telemetry remain necessary; they simply attach to compiled artifacts through the controller instead of to a fixed proxy data path. Similarly, adoption need not require business-logic changes: applications already call an RPC library, and an ADN deployment can replace that dependency and introduce gateways along individual communication edges.

Finally, ADN is not just a single wire protocol, a service mesh implementation, or an RPC library implementation. It is a model for synthesizing those artifacts from one high-level semantic specification. AppNet and fRPC are the concrete instantiations built in this dissertation.

3.7 Summary and the Path Ahead

This chapter has argued that the RPC tax measured in Chapter 2 is not an implementation accident. It is the structural cost of using a general-purpose, layered, proxy-centric stack for microservice deployments within

the same organization, where schemas, policies, delivery needs, and trust boundaries are often known ahead of time.

ADN is the architectural response: specify the RPC communication component once, at the level of application behavior, and compile it into an application-specific stack. The next two chapters put mechanism behind that claim. Chapter 4 starts with the programming and compilation model for application network functions. Chapter 5 then extends the argument to the entire RPC communication stack.

Chapter 4

AppNet: High-Level Programming for Application Networks

The previous chapter argued that microservice communication should be *specified once at a high level and compiled into a flat, application-specific implementation*. It laid out three design principles—separate specification from implementation, co-design across the stack, and behavioral equivalence as the safety net—and divided the realization of the paradigm into two technical pillars. The first pillar makes the policy layer programmable and optimizable; the second replaces the layered RPC stack beneath it. This chapter develops the first pillar.

Concretely, this chapter contributes **AppNet**: a programming model and optimizing compiler for the application network functions (ANFs) that implement routing, load balancing, access control, observability, and other policies. AppNet answers, for this policy layer, the question: *what does a high-level ANF specification look like, and what does a compiler for it have to do?* The answer is that an ANF should be specified as a chain of small, stateful *elements* written as match-action rules over RPC fields and state, and that an optimizing compiler should be allowed to fuse, reorder, and relocate those elements as long as a symbolic equivalence procedure proves that the transformed chain preserves the developer's intended observations. The chapter shows that this level of specification is enough to make ANFs expressible in tens of lines rather than hundreds; to let the compiler reduce median RPC-processing overhead by 74–83% under weak consistency; and to do so without touching the layered RPC stack itself, which Chapter 5 attacks next.

The rest of the chapter proceeds from problem to artifact to evidence. Section 4.1 shows why operators currently make placement, ordering, and platform choices by hand. Section 4.2 introduces the element and chain abstractions that make ANFs analyzable. Section 4.3 describes the equivalence checker, optimizer, backends, and update protocol. Section 4.4 describes the prototype implementation. Section 4.5 evaluates expressiveness, overhead reduction, end-to-end performance, and the remaining abstraction tax.

4.1 The Developer's Predicament

Today's application networks start with hand-written data-plane artifacts rather than high-level policy specifications. A developer who wants a new ANF usually writes it for a particular runtime—in C++ as a native Envoy filter [23], in Rust compiled to WebAssembly for Envoy's WASM module API [43], or in Go as an in-process gRPC interceptor [20]. Before the system can run that ANF, the developer must also choose a concrete configuration for it: where it appears in the chain of ANFs on an RPC edge, whether it executes on the client side or the server side, and which execution platform hosts it. These choices are implementation details in the sense that they do not change the policy the developer wants to express, but they are exposed to the developer because today's controllers treat ANFs as black boxes. A controller can load an ANF and steer traffic through it; it cannot infer whether a different order, side, or platform would preserve the intended behavior. In other words, ANFs today are black boxes to the controller.

The platform choice. The same logical ANF can run at several points along the RPC path, as Figure 4.1 shows. Running inside the RPC library avoids proxy crossings but shares fate and trust with the application. Running in a sidecar enforces policies outside the process but adds proxy overhead. Running in a remote proxy consolidates processing but adds a network hop. These choices differ in latency, CPU cost, upgradeability, failure isolation, and trust, so placement is a correctness and performance decision, not just a deployment preference.

The running example. One edge from the Hotel Reservation benchmark [69] makes the coupling concrete. Search, a replicated service, sends RPCs to Geo, another replicated service that partitions geographic objects across instances. Suppose the developer wants four ANFs on this Search–Geo edge: *Logging*, which

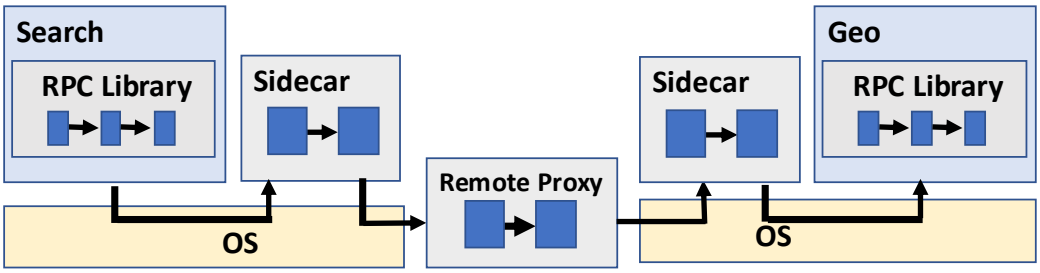
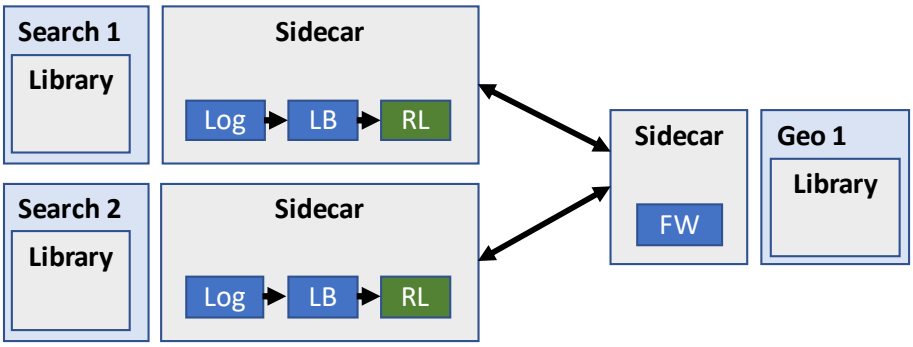


Figure 4.1: RPC processing platforms between the Search client and Geo server in the Hotel Reservation benchmark. Each colored band represents one execution platform an ANF can use: the application's RPC library (in-process), a sidecar proxy (per-host, out-of-process), or a remote proxy (shared, off-host). Each platform has a different performance, trust, and upgradeability profile.

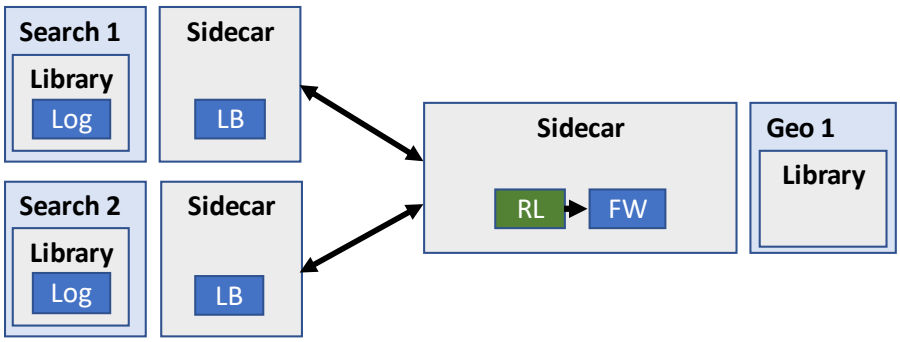
records each RPC and response in a per-edge log; *Load Balancing*, which chooses a Geo replica; *Rate Limiting*, which applies a token bucket to RPCs reaching each Geo replica; and an *Application Firewall*, which drops RPCs whose user field appears in a denylist. The developer can express this desire as a chain, but today's systems also require them to decide the order, side, platform, and state coordination scheme for every element in that chain.

Why the decision is hard. The choices appear local, but they interact through the semantics of the ANFs and through the cost of the platforms.

- *The order in which the ANFs run.* Logging-then-Firewall and Firewall-then-Logging produce different logs (since the latter never logs dropped RPCs). Load-Balancing-then-Rate-Limit and Rate-Limit-then-Load-Balancing produce different state-update patterns (since the rate limiter's state may depend on the destination chosen by the load balancer, or vice versa).
- *The side—client or server—on which each ANF runs.* A rate limiter whose state is indexed by server replica is naturally placed on the server, where a single element instance reads and writes the state; placing it on the client side requires synchronization across client replicas. A firewall whose decision causes a drop is naturally placed upstream—on the client—to spare the network and the downstream ANFs; but if the firewall must be enforced outside the application, it cannot live in the client's RPC library.
- *The platform on which each ANF runs.* The RPC library is the cheapest option, but it lives inside



(a) Developer-supplied configuration.



(b) Compiler-produced, semantics-preserving configuration.

Figure 4.2: Two configurations for the Search–Geo edge in the Hotel Reservation benchmark. The developer-supplied configuration (top) fixes an order, side, and platform for each element. The compiler-produced configuration (bottom) moves Logging into the in-process RPC library and places Rate Limiting where its server-replica state is cheapest to maintain, while preserving the stateful semantics of the original chain. The two configurations differ in performance but are observationally equivalent under the developer's specification.

the application's address space and so cannot be used for ANFs that need to be enforced against an untrusted application. A native Envoy plugin is fast but cannot be hot-upgraded; a WebAssembly plugin is upgradeable but pays a per-invocation overhead. A remote proxy spares each host its sidecar but adds an extra network hop.

- *How shared state is synchronized.* The rate limiter's token bucket may be shared across all client instances (in which case it must be coordinated through external storage) or partitioned per server replica (in which case it does not). The latter is faster; which one is correct depends on the original specification.

Figure 4.2 visualizes this example. The top configuration is a plausible developer-supplied one: each

element has a fixed order, side, and platform. However, once the platform costs and ANF semantics are considered together, a more efficient equivalent configuration exists. Logging can move into the in-process RPC library to avoid proxy overhead, and Rate Limiting can move to the server side, where token-bucket state indexed by Geo replica does not require synchronization across Search replicas. Other performance-improving transformations are unsafe: moving the Application Firewall ahead of Rate Limiting could save work by dropping denied RPCs earlier, but it would also change whether those RPCs consume rate-limit tokens. The problem is therefore not merely to find a better configuration; it is to find one that is faster and semantics-preserving.

What an answer must do. Today's controllers cannot optimize these choices because ANFs are black boxes: they can run an ANF, but cannot tell whether a cheaper configuration is equivalent. AppNet follows the prescription of Chapter 3: lift the ANF specification to a level the compiler can analyze. The specification exposes which RPC fields an element reads and writes, which state it touches, whether it may drop or reorder RPCs, and which auxiliary outputs it produces. Once those effects are explicit, a compiler can search for a cheaper order, side, and platform assignment, then validate it against a notion of equivalence.

4.2 The AppNet Abstraction

AppNet's language has two core ideas. First, ANFs are written as chains of *elements* that process individual RPCs while exposing state and side effects in the syntax. Second, chains describe *what* must happen, not *where* or *how*; placement, platform selection, and optimized order are compiler decisions. The language focuses on the data plane, while allowing control plane logic to update shared state. Figure 4.3 summarizes the grammar used below.

4.2.1 Elements

The unit of programming in AppNet is the *element*: a small piece of code that processes a request or response, may read or write RPC fields and state, may drop the RPC, and may emit to auxiliary channels such as logs or metrics. Each side effect appears as syntax, which makes the element analyzable.

$$
\begin{aligned}
\mathit{Chain} ::= & \left[\begin{array}{l}
\texttt{client}\text{: } \mathit{Element}^* \\
\texttt{any}\text{: } \mathit{Element}^* \\
\texttt{server}\text{: } \mathit{Element}^* \\
\texttt{pair}\text{: } (\mathit{Element}, \mathit{Element})^* \\
[\texttt{weak}]
\end{array}\right. \\
\mathit{Element} ::= & \left[\begin{array}{l}
\texttt{state}\text{: } \mathit{Decl}^* \\
\texttt{init}(\mathit{Var}^*)\text{: } \mathit{Assign}^* \\
\texttt{req}(\mathit{Var})\text{: } \mathit{Action}^*\ [\mathit{MatchAction}] \\
\texttt{resp}(\mathit{Var})\text{: } \mathit{Action}^*\ [\mathit{MatchAction}]
\end{array}\right. \\
\mathit{Decl} ::= & \ \mathit{Var}\ [\texttt{shared}\ [\texttt{weak}\ [\texttt{sum}]]] \\
\mathit{MatchAction} ::= & \ \texttt{match}(\mathit{Expr})\ \mathit{Case}^+\ [\texttt{'*'} \texttt{=>}\ \mathit{Action}^+] \\
\mathit{Case} ::= & \ \mathit{Literal} \texttt{=>}\ \mathit{Action}^+ \\
\mathit{Action} ::= & \ \mathit{Assign} \mid \mathit{Send} \mid \mathit{Foreach} \mid \mathit{Return} \\
\mathit{Assign} ::= & \ \mathit{Var} = \mathit{Expr} \mid \texttt{set}(\mathit{Var}, \mathit{Expr}^+, \mathit{Expr}) \\
\mathit{Send} ::= & \ \texttt{send}(\mathit{Message}, \mathit{Channel}) \\
\mathit{Foreach} ::= & \ \texttt{foreach}(\mathit{Var}, \mathit{LambdaFunc}) \\
\mathit{Return} ::= & \ \texttt{return}\ [\mathit{Expr}] \\
\mathit{Message} ::= & \ \mathit{Var} \mid \texttt{'error'} \\
\mathit{Channel} ::= & \ \texttt{down} \mid \texttt{up} \mid \mathit{Var} \\
\mathit{Expr} ::= & \ \mathit{Literal} \mid \mathit{Var} \mid \texttt{get}(\mathit{Var}, \mathit{Expr}^+[, \mathit{LambdaFunc}]) \\
& \mid \text{BuiltinFunc}(\mathit{Expr}^*) \\
\mathit{LambdaFunc} ::= & \ \texttt{lambda}(\mathit{Var}+) \texttt{=>}\ \mathit{Action}^*\ [\mathit{MatchAction}] \\
\mathit{Var} \in & \ (\text{set of variable names}) \\
\mathit{Literal} \in & \ (\text{literal values, e.g. } 0.1, 42, \mathbf{true})
\end{aligned}
$$

Figure 4.3: Core AppNet specification language, omitting type annotations and usability-oriented syntactic sugar.

Structure. An element is organized into four sections:

- `state`: declares the variables the element carries across RPCs. The state variables can be marked as local to an element instance, shared across instances, or shared with a control plane process.
- `init`: initializes those variables, optionally from constructor arguments supplied when the chain is instantiated.
- `req`: defines the request path processing logic. It receives the RPC as a key-value map, applies match-action rules over RPC fields and state, and either forwards a modified RPC downstream, sends a synthetic message (e.g., an error) upstream, or drops the RPC.

- `resp`: defines the response path processing logic, using the same structure as `req`.

State is persistent memory across RPCs. Local state is private to one element instance; shared state is visible to multiple instances and to control plane logic. Element bodies access state through `get` and `set`, usually with an explicit key such as the destination replica in a token bucket. The compiler can therefore tell not only that an element is stateful, but also which state variable it touches and which RPC fields determine the state key. Element bodies use a small expression language with variables, literals, built-ins, and match-action rules that can update state, modify the RPC, or send a message on a named channel.

Why match-action. AppNet uses match-action rules because they strike a useful balance between expressiveness and compiler visibility. Low-level plugins, such as Envoy WebAssembly modules [43], can implement arbitrary logic, but they hide the RPC fields, state variables, and inter-element dependencies that a controller needs in order to combine, reorder, or place functions intelligently. More restrictive abstractions, such as SQL-style dataflow, are easier to analyze but do not capture common RPC processing patterns. AppNet's match-action rules occupy a middle ground: they operate over RPC fields, structured state, and built-in functions rather than raw packets, while preserving the dependencies the compiler needs. The evaluation shows that common ANFs and two production designs fit in compact AppNet elements without giving up this structure.

Example element: rate limiting. Figure 4.4 shows both the running chain and the rate-limiting element from Section 4.1. The element keeps a shared token bucket in `state`. On each request, `req` extracts the RPC destination, uses that destination as the bucket key, and runs a lambda under `get` to test and decrement the corresponding token count atomically. If a token is available, the element forwards the RPC downstream; otherwise, it returns a rate-limit error upstream. Refill logic lives outside this data-plane element: a control plane process can update the same shared state on a timer. The rate limiter is written in 25 lines of AppNet code; the equivalent Envoy-native C++ implementation in our comparison is 463 lines. The compact syntax also preserves compiler visibility: the compiler can see that the element reads the RPC destination, touches shared state keyed by that destination, and may drop the request.

```
client: logging()->load_balancing()
any: rate_limiting()->firewall()
```

(a) An element-chain specification with four elements.

```
state:
    bucket shared

init():

req(rpc):
    key = get(rpc, 'dst')
    has_token = get(bucket, key, lambda(token):
        match token > 0:
            true =>
                token = token - 1
                return true
            false =>
                return false
    )

    match has_token:
        true =>
            send(rpc, down)
        false =>
            send(err('rate limited'), up)

resp(rpc):
    send(rpc, up)
```

(b) The element specification for rate limiting.

Figure 4.4: Example AppNet specifications for the running Search–Geo chain.

4.2.2 Chains, Sub-chains, and Pair Elements

Above the element, AppNet's unit of composition is the *chain*: an ordered sequence of elements that describes the RPC processing between a pair of microservices. A chain is partitioned into four sub-chains, each capturing a placement constraint:

- `client`: elements that must run on the client side. Typically these are elements that need access to client-only state (e.g., per-caller authentication tokens) or that must execute before the RPC leaves the client process for security reasons.
- `any`: elements with no intrinsic side preference. The compiler is free to place these wherever it

produces the lowest-cost plan.

- `server`: elements that must run on the server side. These are typically state-coordination ANFs (e.g., a server-side rate limiter that aggregates requests across all clients) and policies that must be enforced regardless of client cooperation.
- `pair`: pairs of elements that must be used together, with the first running on the client and the second on the server. Compression/decompression and encryption/decryption are the canonical examples. Pairs are syntactic sugar over a pair of `client`/`server` sub-chains, but the pair form lets the compiler reject configurations that place a conflicting element between the two halves (e.g., a logger that would observe the compressed payload).

The order within each sub-chain is significant; the compiler is allowed to reorder elements only when its equivalence procedure (Section 4.3.2) certifies that the reordering preserves behavior. The implicit cross-sub-chain order is `client` → `pair-client` → `any` → `pair-server` → `server`: any RPC traverses the client elements first, then the client half of each pair, then the `any` elements in their specified order, then the server half of each pair, then the server elements. The chain specification in Figure 4.4a should therefore be read as a semantic chain, not as a mandate to run each element in exactly that place.

4.2.3 Shared State and Consistency as a Knob

Chains give the compiler freedom to choose where elements run, but state often determines whether a placement is cheap or expensive. AppNet therefore exposes whether a state variable is local or shared, and which key an element uses when it reads or writes that variable. The compiler recovers these dependencies from `get` and `set`: a token bucket keyed by server replica may become local if the element runs server-side, while truly global state must pay for coordination.

That coordination is the main cost of shared state. Strongly consistent shared state is implemented with a Redis-backed key-value store [41], following Envoy's global rate limiter [14]; every strongly consistent access can therefore add a network round trip. Marking state as shared is both a correctness annotation and an input to the optimizer's cost model.

AppNet also lets the developer relax state consistency when the ANF can tolerate it. Weakly consistent state uses local reads and writes with asynchronous reconciliation, using last-writer-wins by default or a

developer-supplied aggregator in the style of CRDTs [118]. This familiar eventual-consistency trade-off [63, 134] is a language-level knob because it changes the configurations the compiler should consider.

The same idea applies to observations, not just state. Under *strong observation consistency*, two configurations are equivalent only if they produce the same microservice-visible messages, final state, and auxiliary outputs such as logs and metrics. Under *weak observation consistency*, the microservice-visible messages and final state must still agree, but auxiliary outputs may differ. For example, a logger before a load balancer may record a different `dst` field than a logger after it. If the application treats logs as best-effort telemetry, weak observation consistency gives the optimizer more freedom; if logs are part of the application contract, the stronger setting preserves them.

4.2.4 Location, Platform, and Pair Constraints

A second design decision is that the developer should not normally choose the concrete execution platform. The `client`, `server`, `any`, and `pair` sub-chains describe semantic placement constraints: what must be client-side, what must be server-side, what may move, and what must appear as a pair. Within those constraints, the compiler chooses the platform—an in-process gRPC interceptor, a native Envoy sidecar filter, an Envoy WebAssembly filter, or a remote proxy—that gives the lowest-cost equivalent configuration.

Developers can still add hard constraints when the policy requires them. The two common cases are trust and upgradeability. An element that enforces a policy against the application itself, such as a firewall or access-control filter, cannot safely run inside the application's RPC library because the application could bypass it. An element that must be hot-upgraded under a changing policy can run in the gRPC library or EnvoyWasm, both of which support dynamic loading, but not as an EnvoyNative filter compiled into the proxy binary. AppNet treats these annotations as bounds on the compiler's search rather than as a hand-written deployment plan.

4.3 The AppNet Compiler

AppNet's compiler turns the information exposed by the language into an executable deployment. Given a chain specification and the application's RPC schema, it produces platform-specific modules for each communication edge, plus the deployment artifacts that place those modules in the data path. The central

optimization question is simple to state: *among all configurations that preserve the user's semantics, which one has the lowest cost?* Answering it requires three pieces working together: a cost model that ranks candidate configurations, an equivalence checker that rejects unsafe transformations, and a search algorithm that explores the space of orders, locations, and platforms.

4.3.1 Pipeline Overview

The compiler runs in stages. First, AppNet parses the chain and RPC schema, validates field and state references, and emits serialization code from the schema. It then extracts the properties needed by later analyses: field and state dependencies, possible drops or reorders, and auxiliary outputs. The extraction is conservative, treating any field that appears in any branch as a read.

The optimizer searches over element orderings, side assignments, and platform assignments. A candidate is useful only if the equivalence checker certifies that it preserves the input chain's semantics; among certified candidates, the cost model picks the cheapest. The selected configuration is then compiled into Go gRPC interceptors, C++ EnvoyNative filters, or Rust-to-WebAssembly EnvoyWasm modules. Elements placed on the same side and platform are fused, and generated deployment rules bypass unused platforms.

Finally, the AppNet controller, integrated with Kubernetes [32] and Istio [29], installs the generated modules. It watches deployment and policy changes, recompiles when needed, and uses the two-phase versioned rollout from Section 4.3.5 so each RPC is processed by exactly one version of the chain.

The rest of this section focuses on the two pieces that make the optimizer safe and effective: symbolic equivalence and cost-driven search.

4.3.2 Symbolic Equivalence for Stateful ANFs

The optimizer can reorder elements, move them between sides, and change execution platforms only when those changes preserve the semantics of the user's specification. The equivalence procedure defines that boundary: conservative enough to protect behavior, but precise enough to expose optimization opportunities. Its goal is not arbitrary program equivalence, but equivalence of AppNet chains under the restricted effects exposed by the language.

The challenge. Two features make this harder than equivalence for stateless packet-processing rules. First, ANF elements are *stateful*: processing one RPC can update state that affects later RPCs, so equivalence must hold for arbitrary RPC streams rather than for one RPC in isolation. Second, ANFs may *drop* or *reorder* RPCs. Moving a firewall across a load balancer, for example, can change which requests update the load balancer's state even if the same set of requests eventually reaches the server. Classical techniques for stateless packet forwarding (e.g., NetKAT [50]) and instruction reordering [122] do not cover this combination of state, drops, and reordering.

Symbolic abstraction. AppNet handles these cases by replacing each element with symbolic *transfer functions*. The abstraction records what the element may change and what those changes depend on, without modeling the element's arithmetic or application-specific logic directly. It tracks five effects:

- For each field f the element writes, a transfer function $\tilde{r}.f \leftarrow \mathbf{wf}_{(i,f)}(\textit{deps})$, where $\mathbf{wf}_{(i,f)}$ is an opaque function symbol unique to the (element, field) pair and *deps* is the set of symbolic values (RPC fields and state variables) the element reads.
- For each state variable v the element writes, a transfer function $\tilde{s}.v \leftarrow \mathbf{ws}_{(i,v)}(\textit{deps} \cup \textit{drops} \cup \textit{reorders})$, where the set of dependencies additionally includes the running collection of drop and reorder events accumulated so far in the chain. The inclusion is crucial: a stateful element's behavior depends on whether previous RPCs were dropped or reordered.
- If the element may drop, a fresh symbol $\mathbf{dr}_i(\textit{deps})$ is appended to a running set $\tilde{r}.drops$ of drop-event symbols.
- If the element may reorder, a fresh symbol $\mathbf{ord}_i(\textit{deps})$ is appended to $\tilde{r}.reorders$.
- For each auxiliary channel the element emits to, the current RPC dictionary $\tilde{r}$ is appended to the channel's accumulator.

The function symbols are opaque: $\mathbf{wf}_{(i,f)}$ and $\mathbf{ws}_{(i,v)}$ identify a particular element's write to a particular field or state variable, but carry no algebraic meaning beyond their arguments. By over-approximating dependencies and treating internals opaquely, AppNet avoids reasoning about arithmetic, encryption, or arbitrary built-ins while preserving the information needed for sound comparison.

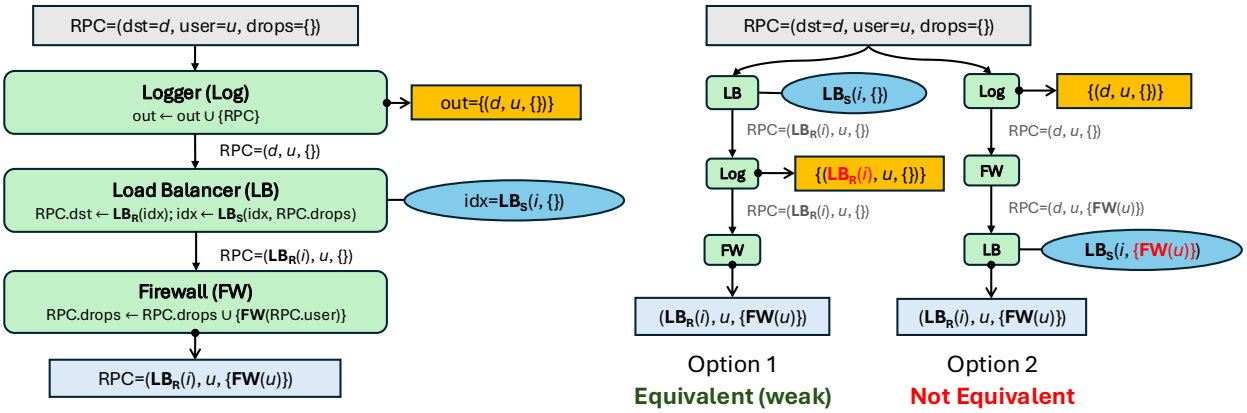


Figure 4.5: Symbolic equivalence checking for a chain of three elements: Logger, Load Balancer, and Firewall. The left column shows the original chain together with the chain-level symbolic transfer functions of the output RPC (blue rectangles), state updates (ovals), and auxiliary channel output (yellow rectangle). The right columns show two candidate reorderings. Option 1 (Load Balancer before Logger) differs from the original only in the logger's auxiliary channel; it is weakly equivalent but not strongly equivalent. Option 2 (Firewall before Load Balancer) differs in the load balancer's state-update symbol (the load balancer no longer sees the dropped RPCs in its update dependencies); it is neither strongly nor weakly equivalent. Symbols marked red are those that differ from the original chain.

Symbolic execution. To analyze a chain, AppNet threads two symbolic dictionaries through the elements in order: $\tilde{r}$ for the RPC and $\tilde{s}$ for state. Each element updates these dictionaries according to its transfer functions and extends the symbolic drop, reorder, and auxiliary-output accumulators as needed. The result is a chain-level transfer function for the output RPC, final state, and auxiliary channels.

Figure 4.5 illustrates the check on a three-element chain: Logger, Load Balancer, and Firewall. The left column is the input chain and its symbolic result. The next two columns are candidate reorderings. Moving Load Balancer before Logger changes only the logger's auxiliary output, so it is allowed under weak observation consistency but rejected under strong observation consistency. Moving Firewall before Load Balancer changes the load balancer's state update, because the load balancer no longer sees requests dropped by the firewall; that candidate is rejected under both consistency settings.

Equivalence checking. Two chains are equivalent when their chain-level transfer functions match structurally. Under strong observation consistency, AppNet compares the output RPC, final state, and all auxiliary channels. Under weak observation consistency, it compares only the output RPC and final state, projecting away auxiliary outputs such as logs. Since each opaque function symbol is unique to the (element, field) or

(element, state variable) pair it denotes, equality reduces to a syntactic comparison: checking that the same symbol is applied to recursively equal arguments.

State updates include accumulated drop and reorder events, so moving a dropping element across a stateful one changes the symbolic expression and is rejected (as in Option 2 of Figure 4.5). Auxiliary outputs are included or ignored according to the developer's observation-consistency choice, which is why Option 1 is weakly equivalent but not strongly equivalent.

Soundness and limits. The procedure is sound: if two chains have equal chain-level transfer functions, then for any concrete RPC stream they produce the same output stream and final state, and under strong observation consistency the same auxiliary outputs. The proof follows from structural induction on transfer-function expressions and induction over the RPC stream.

The procedure is not complete. It can reject transformations that are semantically valid but not syntactically visible through the abstraction. For example, two different load-balancing algorithms might choose the same replicas for a particular workload, or two independent counters might commute because of a specific arithmetic identity. AppNet deliberately rejects such cases unless the symbolic expressions match, preserving soundness at the cost of some missed optimizations.

4.3.3 Cost-Driven Optimization

With the equivalence checker in hand, optimization becomes a constrained search problem. The compiler may consider many orders, side assignments, and platforms, but it can keep only the candidates certified as equivalent to the input chain. Among those candidates, it chooses the one with the lowest estimated cost.

Cost function. The cost model encodes the same preferences an operator would apply by hand, but applies them consistently across the whole chain. It prefers cheaper platforms (gRPC interceptor before EnvoyNative before EnvoyWasm), configurations that bypass unused platforms, placements aligned with state dependencies, co-location with strongly or weakly consistent state, and earlier placement of elements that may drop requests. These preferences capture both per-element cost and whole-path cost: moving one element can make the element itself cheaper, avoid a sidecar crossing, or reduce the number of downstream elements that process a request.

The preferences are combined into a scalar score using hand-set weights. Per-element cost is weighted by the fraction of RPCs expected to reach the element; elements downstream of a dropper receive a smaller weight, and the weighting uses a 5% drop-rate estimate when no telemetry is available. Each sub-chain also pays a fixed platform base cost (S_p) for every platform it uses, and the total chain cost sums the client-side and server-side costs. The results are robust to the exact values as long as the parameter ordering is maintained; runtime profiling of platform overheads remains future work.

This cost model is a heuristic, not an oracle. The optimizer must score many configurations quickly, so the model needs only enough fidelity to rank qualitatively different choices: in-process versus sidecar, local versus Redis-backed state, and one platform crossing versus two.

Search algorithm. For short chains, the optimizer enumerates the configuration space and invokes the equivalence procedure on each candidate. For longer chains, exhaustive search becomes too expensive, so AppNet uses multi-start simulated annealing [125]. Each run starts from a valid configuration and proposes small mutations: moving an element, swapping two elements, or changing a platform. Mutations that fail the equivalence check are rejected; equivalent mutations are accepted according to the annealing schedule and their cost improvement. Restarts help the search escape local minima.

The evaluation in Section 4.5.2 shows that multi-start simulated annealing finds the brute-force-optimal configuration in more than 98% of cases for chains of up to twelve elements, with a worst-case cost gap below 1.5%, and runs 473× faster than brute force for the same chain size. For chains of size 100, simulated annealing remains practical, whereas brute-force search becomes computationally prohibitive.

4.3.4 Code Generation and Platform Backends

Once the optimizer selects a configuration, the compiler turns it into deployable code. AppNet supports three backends that correspond to the main places application-network logic runs today:

- **gRPC interceptors** (Go). Modules are emitted as Go functions that implement gRPC's interceptor interface [20]. Because the interceptor runs inside the application's address space, this backend cannot host elements marked as "must not run in the RPC library"; conversely, it is the cheapest backend by a significant margin.

- **EnvoyNative** (C++). Modules are emitted as C++ Envoy filters that compile into the Envoy binary. EnvoyNative is the fastest of the out-of-process backends but cannot be hot-upgraded; elements that require hot upgrades cannot use it.
- **EnvoyWasm** (Rust $\rightarrow$ WebAssembly). Modules are emitted as Rust source that the compiler then translates, via the Rust compiler and a WASM target, into WebAssembly bytecode loadable through Envoy's proxy-wasm interface [43]. EnvoyWasm is the slowest of the three backends—each invocation crosses the WebAssembly boundary—but it is hot-upgradeable and language-agnostic.

Each backend includes a small runtime that adapts the AppNet element interface to the platform's native API: deserializing RPCs, managing state, routing `send` actions, and connecting strongly consistent state operations to Redis. The generated code is platform-specific, but element semantics are shared across backends.

Module fusion. When multiple elements are placed on the same side and platform, the compiler emits one fused module rather than one module per element. Fusion removes per-module dispatch overhead, especially for EnvoyWasm where each module invocation crosses the WebAssembly boundary, and it lets co-located elements share deserialization and re-serialization work. The fused module preserves the selected element order, so it does not introduce a new equivalence question.

Bypass scripts. The compiler also emits deployment scripts, including Kubernetes custom resources and network policies, that route traffic only through the platforms used by the chosen configuration. If no element runs in a sidecar on one side of an edge, AppNet bypasses that sidecar for the edge. This granularity matters because today's proxies often intercept every RPC whether or not a filter needs to run.

Shared state. Strongly consistent shared state is realized through Redis, following Envoy's global rate limiter [14] and StatelessNF [83]. A `get` or `set` on a strongly consistent variable becomes a synchronous Redis operation. Weakly consistent state uses per-instance local copies with background reconciliation: writes apply locally first and propagate asynchronously, with last-writer-wins or developer-supplied aggregators resolving conflicts. These implementation choices are the costs the optimizer accounts for earlier.

4.3.5 Consistent Updates

An application network is not static. Policies change, replicas scale, and the controller may recompile an edge after either event. Each update creates a consistency problem: if one side of an edge runs the new configuration while another side still runs the old one, an RPC can traverse a chain that no version of the specification described. Service meshes can update individual filters atomically, but they do not by themselves guarantee edge-wide consistency across a chain [1, 82].

AppNet ensures that every RPC traversing an updated edge is processed entirely by the old configuration or entirely by the new one, never a blend. It does so by adapting the two-phase versioned update protocol of Reitblatt et al. [111] to RPC chains.

Mechanism. The client tags each RPC with a version number in a custom `appnet-version` header. Each installed configuration processes only RPCs carrying its version. To move from version v to $v + 1$, the controller first installs the new configuration everywhere without activating it. It then switches the client tagger so new RPCs carry $v + 1$. In-flight RPCs tagged with v continue through the old modules, while new RPCs use the new ones. Once all v-tagged RPCs have drained, the controller unloads the old configuration.

The protocol works because the version tag is set once, at the client, and is not modified in transit. Every RPC therefore selects exactly one configuration end-to-end.

4.4 Implementation

The AppNet prototype consists of a 16K-line Python compiler and a 3.1K-line Go controller. Developers submit AppNet specifications through a custom Kubernetes resource (CRD). The controller watches this resource together with Kubernetes deployment state, such as replica creation or removal, and recompiles affected communication edges when the policy or application topology changes. It then installs the compiler output through Kubernetes and Istio control plane APIs.

The compiler targets the three execution platforms described above. For gRPC, it emits Go interceptors that run in the client or server process. For EnvoyNative, it emits C++ filters that are compiled into Envoy. For EnvoyWasm, it emits Rust source that is compiled to WebAssembly and loaded through Envoy's proxy-wasm interface. EnvoyNative and EnvoyWasm modules can run in sidecar proxies or in a remote

proxy deployment [80]; gRPC interceptors cannot, because they execute inside the application process. Conversely, the in-process gRPC backend is the cheapest target but is unavailable for elements that must be enforced outside the untrusted application's code.

All backends assume that the target platform can inspect the RPC headers and payload fields named by the AppNet specification. gRPC interceptors satisfy this assumption because they run before transport encryption is applied. Envoy-based targets satisfy it when the proxy terminates the encrypted connection, as in standard sidecar and ambient mesh deployments. If an application requires end-to-end encryption through the proxy, AppNet restricts placement to platforms that can legally observe the relevant fields.

Shared state is implemented with the Redis-backed design described in Section 4.3.4. Strongly consistent `get` and `set` operations synchronously access Redis, while weakly consistent state uses the same store for background reconciliation. In the prototype, finding the best configuration for the five-element chains used in the evaluation takes about 1.4 seconds. Generating gRPC and EnvoyWasm modules takes a few seconds; generating EnvoyNative modules takes one to two minutes because it requires compiling the Envoy codebase.

4.5 Evaluation

The evaluation asks three questions. First, can AppNet express the ANFs that appear in real deployments? Second, can the compiler reduce RPC-processing overhead by choosing where and how those ANFs run? Third, do those local reductions translate into better end-to-end application performance?

Setup. All experiments run on CloudLab [64] machines with two 16-core Intel Xeon Gold 6142 CPUs (2.6 GHz) and 384 GB of RAM, running Ubuntu 20.04 (Linux 5.4.0), Kubernetes 1.28.13, and Envoy 1.30.5. To reduce measurement variance, we disable CPU Turbo Boost, C-states, and dynamic frequency scaling. Load is generated with `wrk` [45] and `wrk2` [46], both of which support high-precision tail-latency measurement. Each microservice is replicated five times unless otherwise noted; the qualitative results are stable across other replication factors. All applications in the evaluation are written in Go and use gRPC. Remote proxies, when used, are shared by all replicas of the client and server microservices.

The evaluation reports three metrics throughout. *Service time* is the latency of a single RPC under

ANF	Shared state	Lines (AppNet)
Fault Injection [28]		11
Cache [10]	✓	14
Rate Limiting [11]	✓	25
Load Balancing [12]	✓	13
Logging [8]		10
Mutation [21]		7
Application Firewall [38]	✓	12
Metrics [15]		12
Admission Control [52]		21
Encryption [19]		28
Bandwidth Limit [9]		21
Circuit Breaking [26]		16
Meta's ServiceRouter [114]	✓	62
Google's Prequal [131]	✓	88

Table 4.1: The 14 ANFs implemented in AppNet, with lines of AppNet code and whether they declare any shared state. The 12 common ANFs are drawn from production module libraries; the two industrial ANFs (ServiceRouter, Prequal) are reproductions from their respective published designs.

minimal load (at most one outstanding request), which isolates the per-message work performed by the chain. *Tail latency* is the 90th-percentile latency under moderate load (30–40% CPU utilization), which captures the queueing behavior the chain induces. *CPU usage* is the total number of virtual cores consumed by RPC processing across the application's microservices.

4.5.1 Expressiveness

The expressiveness study has two parts. The first asks whether AppNet can express the standard ANF library shipped by production service meshes. We implement the twelve common ANFs in Table 4.1, identified by surveying the official module collections for Envoy, Linkerd, and proxyless gRPC [17, 21, 23, 33]. Each element compiles to 7–28 lines of AppNet. For comparison, we implement the same ANFs in each platform's native extension language: Go for gRPC, C++ for EnvoyNative, and Rust for EnvoyWasm. These implementations require 5–60$\times$ as many lines as the corresponding AppNet elements. The C++ modules are the most verbose, requiring roughly 4$\times$ as many lines as the Go and Rust modules because Envoy's native filter API requires substantial boilerplate.

The second part targets more complex ANFs. We implement two recent industrial ANFs in AppNet: Meta's ServiceRouter [114], a routing-and-load-balancing system that applies power-of-two-choices over a shared shard map maintained by a control plane, and Google's Prequal [131], a load balancer that selects

replicas using estimated latency and active in-flight requests via a hot–cold lexicographic rule. ServiceRouter requires 62 lines of AppNet, and Prequal requires 88 lines. Both implementations are non-trivial. Each includes a client-side element that maintains a load map and applies the selection rule, a server-side element that monitors load and embeds it in response headers, and a control plane component that maintains either the shard map (ServiceRouter) or a probed-latency map (Prequal).

4.5.2 RPC Processing Overhead Reduction

To assess how well AppNet reduces RPC-processing overhead, we need a corpus of network specifications representative of those used in practice. To our knowledge, no such public dataset exists, so we synthesize a wide range of chain specifications and use them to benchmark the range of overhead reductions AppNet can provide. This method lets us study which specification characteristics lead to larger or smaller reductions. For each chain, we sample a random subset of the twelve common ANFs in Table 4.1, assign each element to a sub-chain (`client`, `server`, or `any`), and impose random platform constraints ("do not run in an RPC library," "requires dynamic upgrades"). The sampled ANFs capture the diversity of chains an operator might assemble, while the platform constraints capture the heterogeneity of trust and operational requirements within a deployment.

The overhead microbenchmarks use a simple Echo Server application: a frontend microservice sends an echo request, and a backend microservice returns an echo response. This setup isolates the RPC-processing cost of the generated chain; the application-level experiments below use richer benchmarks.

We compare three end-to-end configurations. *NoOpt* assigns each element a random valid platform and side, subject to the user's constraints; it approximates the result of configuring a chain without a cost model. *LocalOpt* is a stronger baseline: for each element independently, it picks the cheapest platform and aligns side assignments with state dependencies, but it does not optimize across elements. Both baselines preserve the input element order and make placement decisions sequentially, so upstream choices can constrain the choices available to later elements. *AppNet* is the compiler's output, evaluated in two modes: *strong consistency* (all state and observation constraints are strong) and *weak consistency* (all consistency knobs are relaxed).

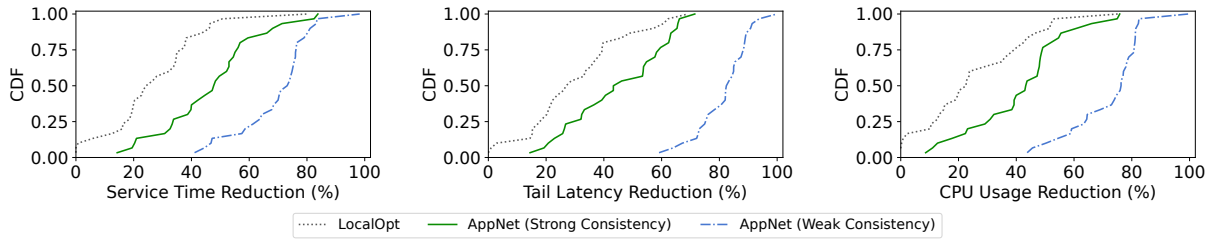


Figure 4.6: Reduction in RPC-processing overhead relative to NoOpt for thirty randomly sampled 5-element chains. *Local* is LocalOpt; *Strong* and *Weak* are AppNet under strong and weak consistency. AppNet under strong consistency roughly doubles LocalOpt's reduction; weak consistency widens the gap further by avoiding the cost of strongly consistent shared state. Figure adapted from [140].

All-platforms environment. Figure 4.6 reports the reduction in service time, tail latency, and CPU usage for thirty randomly sampled 5-element chains, relative to NoOpt. The absolute NoOpt numbers span 0.8–2.0 ms in service time, 2.5–12.1 ms in tail latency, and 10.9–28.6 virtual cores. LocalOpt reduces the median service time, tail latency, and CPU usage by 25%, 27%, and 22%, respectively. These gains are useful but limited because per-element heuristics cannot reason about cross-element interactions. AppNet under strong consistency nearly doubles the improvement, with median reductions of 47%, 44%, and 42%, driven by reordering and consolidation that LocalOpt cannot safely perform. Under weak consistency, AppNet reaches median reductions of 74–83% across the three metrics; most of the additional headroom comes from eliminating synchronous Redis round-trips for shared state.

The variation across chains is significant, but it follows the structure of the chain. The largest gains occur when multiple shared-state elements have dependencies that can be aligned, allowing them to be co-located and synchronized once, or when a dropping element can be moved upstream to avoid downstream work. The smallest gains occur when tight placement constraints force elements onto specific sides, or when the chain has no shared state to consolidate.

Single-platform environments. Some operators deploy only one platform: cost-sensitive deployments may standardize on gRPC interceptors, while legacy-compatible deployments may standardize on Envoy-Wasm. Figure 4.7 reports the reduction in each single-platform environment, alongside the all-platforms environment, for chains of size 3 and 5. The methodology is the same as above except that these runs remove trust and dynamic-upgrade constraints. AppNet reduces overhead in every environment, but the magnitude depends on which optimizations the environment permits. For 5-element chains, median reductions across

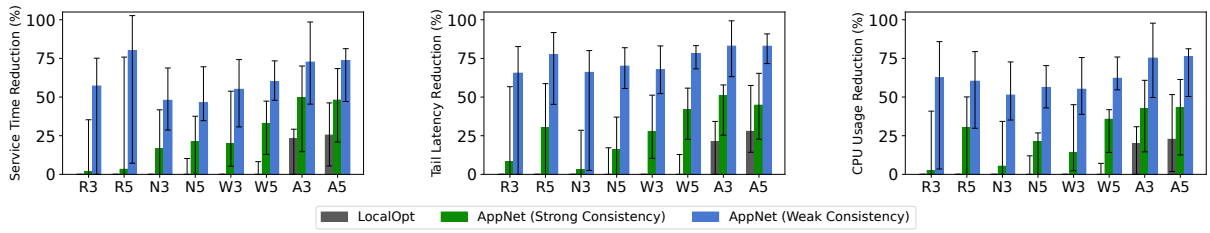


Figure 4.7: Reduction in RPC-processing overhead across four environments—gRPC-only (R), EnvoyNative-only (N), EnvoyWasm-only (W), and all platforms (A)—for chains of size 3 and 5. Bars are medians; error bars are 10th/90th percentiles over thirty random chains. Each environment exposes different optimization opportunities; AppNet still reduces overhead in every one. Figure adapted from [140].

the environments and metrics range from 3–48%; the 10th-percentile reductions range from 0–24%, and the 90th-percentile reductions range from 27–76%. The all-platforms environment has the largest reductions because it also permits platform substitution when the user's constraints allow it. LocalOpt often provides little benefit in single-platform environments because it mainly aligns state dependencies, which helps only when NoOpt randomly chose a poor side for an unconstrained element.

Weak consistency: a closer look. Figure 4.8 decomposes the additional benefit of weak mode into two sources: *weak state consistency*, which moves shared-state synchronization to a background task, and *weak observation consistency*, which lets the optimizer reorder elements across auxiliary-channel writes. Weak state consistency contributes the larger gains: for chains with at least one shared-state variable, it reduces the median metrics by 45–69% because it removes the per-call Redis round-trip. Weak observation consistency has a smaller effect, mainly because it helps when dropping elements can move upstream of auxiliary-channel writes, an optimization that strong observation forbids. Its magnitude therefore depends on the workload's drop rate, which is low in these experiments. Even so, weak observation yields improvements in 38–55% of cases, mostly through consolidation enabled once upstream/downstream constraints are relaxed.

The asymmetry between the two forms of weakening is useful operationally. Weak state consistency is usually worth enabling when the application can tolerate it, while weak observation consistency is most valuable when the workload's drop rate is non-trivial. The two knobs are independent, and the compiler treats them that way.

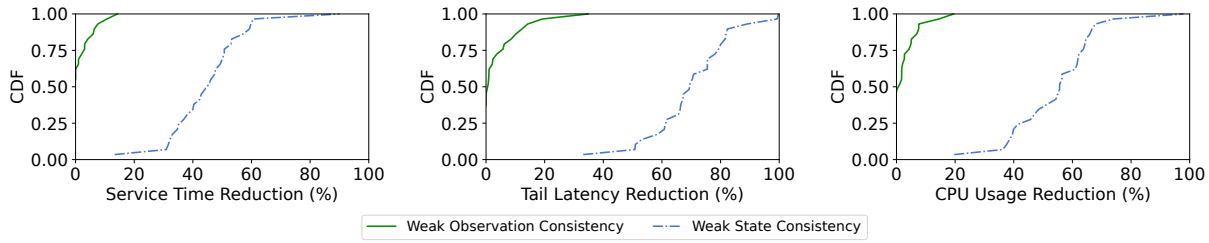


Figure 4.8: Additional overhead reduction from weak state consistency (left) and weak observation consistency (right), relative to AppNet under strong consistency, on the subsets of chains where each form of weakening is applicable. Weak state contributes the larger share; weak observation contributes a smaller share whose magnitude depends on drop rate. Figure adapted from [140].

4.5.3 Application-Level Performance

A reduction in RPC-processing overhead matters only if it improves application-level latency and CPU usage. To test that link, we run AppNet end-to-end on two production-shaped applications: *Hotel Reservation* [69] and *Online Boutique* [73], an eleven-microservice e-commerce benchmark published by Google. For each application, we assign every communication edge a random 5-element chain, sampled as in Section 4.5.2, then measure end-to-end application latency and CPU usage under the same load harness.

Figure 4.9 reports the reduction in end-to-end metrics for Hotel Reservation. The NoOpt baseline spans 7.2–14.1 ms in service time, 35.6–92.4 ms in tail latency, and 136–219 virtual cores. At the median, LocalOpt reduces the three metrics by 14%, 11%, and 11%; AppNet under strong consistency reduces them by 35%, 29%, and 26%; and AppNet under weak consistency reduces them by 49%, 41%, and 42%. Online Boutique shows the same qualitative pattern, with median service-time reductions of 8%, 22%, and 36% across the three configurations and comparable reductions in tail latency and CPU. The smaller magnitudes for Online Boutique reflect its larger application-side latency: more business logic per RPC dilutes the relative gain from RPC-processing optimization.

This result establishes the load-bearing claim of the chapter empirically: lifting ANF specifications to a level the compiler can analyze, and giving the compiler an equivalence procedure aggressive enough to optimize stateful chains, recovers a substantial fraction of the RPC tax measured in Chapter 2 without removing the layered stack itself. The remaining tax, as Chapter 5 will show, is paid by the layered stack and can only be reclaimed by replacing it.

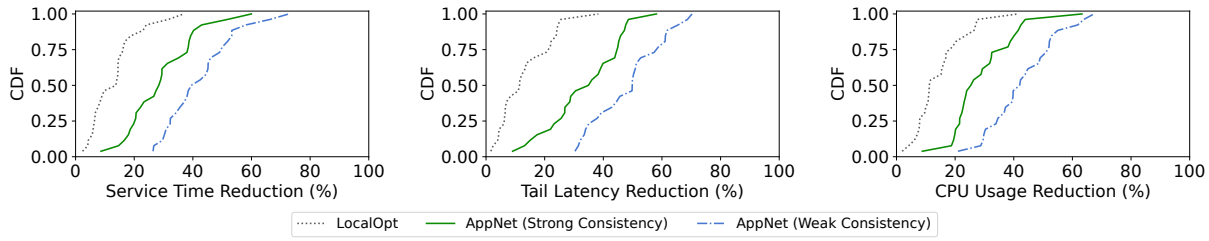


Figure 4.9: End-to-end performance improvement of Hotel Reservation relative to NoOpt, across LocalOpt and AppNet (strong, weak). Each communication edge in the application is given a random 5-element chain. The reductions in RPC-processing overhead (Section 4.5.2) translate cleanly into application-level reductions, with AppNet under weak consistency cutting median application latency by about half. Figure adapted from [140].

4.5.4 The Abstraction Tax

A high-level language pays a tax when its compiled output is slower than hand-written code for the same target. We quantify this tax for AppNet by running each element in isolation on each backend (gRPC interceptor, EnvoyNative, EnvoyWasm) and comparing its per-message latency and CPU usage against two baselines: *Envoy built-ins*, the equivalent modules shipped with Envoy itself and available for all twelve common ANFs except encryption; and *hand-written modules*, which we wrote for elements without an upstream built-in.

The distribution is tight: relative to Envoy built-ins, the median tax is 1–4% on both metrics and the worst case is 3–7%. The comparison against hand-written baselines is qualitatively similar.

We investigated the residual tax and found two main contributors. The first is coarse-grained locking on shared state: the AppNet runtime serializes accesses to a per-element state map, whereas a hand-tuned filter would shard the lock per key. The second is metadata bookkeeping for match-action over RPC fields, which a hand-tuned filter would avoid by hard-coding field offsets. Both costs are addressable with more compiler work, and neither is large enough to outweigh the optimizations from Section 4.5.2: the worst Envoy-built-in tax (7%) is small relative to the median reduction under strong consistency (42–47%) and negligible relative to weak consistency (74–83%).

4.5.5 Optimizer Scalability

To validate the optimizer's scalability, we compare the multi-start simulated-annealing search from Section 4.3.3 against brute-force enumeration for chain sizes 3–12, where brute force is still feasible, and then scale the search to chains of size 100. For chains of size 12, multi-start simulated annealing is 473× faster than brute force and finds the brute-force-optimal configuration in more than 98% of cases; the worst observed cost gap is below 1.5%. For chains of size 100—well beyond what an operator would write by hand, but within the size of synthesized chains once fRPC's delivery pipeline is folded in (Chapter 5)—the search remains practical.

Chains of length 100 are not common today. The point of this scalability check is that the dissertation argues for compiling more of the application network's behavior from a single specification. As the toolchain grows, the equivalence engine's search space grows with it. This experiment shows that the search remains tractable as chains grow.

4.6 Discussion and Limitations

What AppNet establishes. This chapter's central claim is that ANFs become more useful when their behavior is lifted into a form the compiler can analyze. The expressiveness study shows that this does not narrow the programming model: common ANFs in today's service mesh libraries fit, as do two sophisticated industrial designs. The overhead study shows that the compiler can turn that structure into performance: a cost-driven optimizer, enabled by symbolic equivalence, recovers 42–47% of RPC-processing overhead under strong consistency and 74–83% when the application opts into weak consistency. The real application study shows that these savings translate into end-to-end performance, and the abstraction-tax study shows that the high-level language gives away little to hand-written code.

What AppNet does not do. AppNet does not replace the layered stack. RPCs still traverse gRPC over HTTP/2 over TLS over TCP; sidecars still terminate the protocol and re-establish it on the other side; and the protocol-parsing overhead that Chapter 2 identified as the dominant cost of HTTP/gRPC sidecars remains. AppNet's role in the dissertation arc is to compile the policy layer within today's stack. Chapter 5 takes the next step: compiling across the stack by co-designing serialization, delivery, and ANF execution from the

same high-level communication specification.

There are two further limitations worth naming.

The DSL has expressiveness limits. AppNet's match-action language deliberately excludes general loops and arbitrary computation over payload bytes. The ANF survey in Section 4.5.1 suggests that most elements do not need this power. When they do, the parsing logic must be written in a host language and exposed to AppNet as a built-in. The set of built-ins is extensible, but the compiler can reason only about the AppNet-visible interface, not about the host-language implementation behind it. This boundary is similar to TensorFlow's graph model: optimization works over known graph operators, while custom operators preserve expressiveness at the cost of opacity to the compiler [48].

The equivalence procedure is sound but incomplete. As Section 4.3.2 explains, AppNet may reject some transformations that are in fact safe: two common examples are commuting elements that call distinct opaque functions and reorderings that depend on arithmetic identities. This incompleteness affects optimization opportunity, not correctness. In the experiments, it is rare enough that the optimizer finds the brute-force-optimal configuration for more than 98% of chains with up to twelve elements.

4.7 Related Work

AppNet builds on four lines of prior work: high-level network programming, network function optimization, state management for network functions, and alternatives to traditional service mesh sidecars.

High-level network programming. Modular packet-processing pipelines have a long history, beginning with Click [88]. AppNet inherits the chain-of-elements abstraction from this tradition, but extends it in two ways. First, Click elements are arbitrary C++ and therefore opaque to the compiler; AppNet elements expose state, field reads, and field writes as first-class syntax, enabling compiler-driven optimization backed by equivalence checking. Second, Click composes elements within one device, whereas AppNet composes them across a heterogeneous data plane.

A second strand—high-level programming languages for layer-3/4 forwarding (NetKAT [50], Frenetic [68], Pyretic [110], Maple [127], FlowBlaze [106])—develops abstractions for the SDN tier. These systems are powerful, but they operate below AppNet's layer: they describe forwarding behavior over packets, not RPC

processing over structured messages. AppNet borrows the match-action primitive from them (and from P4 [53], which targets switch hardware), but extends it to handle RPC fields, structured state, and built-in computation that lower-layer languages do not need to represent.

NF optimization. Optimizing network function chains for cost has been studied in a long line of systems—OpenBox [56], Metron [86], NFP [123], and the consolidated middlebox dataflow of Sekar et al. [116]. These systems target lower-level NFs and typically merge or parallelize raw-packet processing. AppNet's optimizer follows the same broad goal, but targets the application network layer: its cost model is shaped by data plane processing overhead (Chapter 2), its transformations range over a richer placement space, and its equivalence procedure must account for stateful elements, drops, and reorderings that lower-layer optimizers generally do not face.

State management in NFs. A complementary line of work has tackled state management as a first-class concern in NFs: Split/Merge [109], OpenNF [72], StatelessNF [83], S6 [130], SwiSh [133]. These systems develop techniques for migrating, replicating, and, in SwiSh, maintaining consistency over NF state under scaling and failure. AppNet adopts these lessons—state placement aligned with state dependencies, partitioning by indexed key, and eventually consistent state maintained through background reconciliation—but lifts them into language primitives that the compiler can reason about, rather than leaving them as properties an operator must infer from an implementation. The Redis-backed implementation of strongly consistent state directly follows StatelessNF.

Service mesh overhead and alternatives. The performance cost of service meshes is widely known [81, 87]. Several systems reduce this cost by re-architecting the mesh's software design: Ambient Mesh [80] moves the proxy off-pod, Cilium [6] pushes L4 features into the kernel, ServiceRouter [114] runs routing in-process, Canal Mesh [119] replaces per-pod sidecars with a remote proxy. Each is a point design that improves one placement in the design space. AppNet is complementary: it makes placement a compiler decision rather than an operator decision. Operators who write ANFs in AppNet can receive the benefits of ServiceRouter's in-process model when an element admits it, Canal Mesh's remote-proxy model when an element requires it, and the standard sidecar model otherwise, without rewriting the policy for each target.

4.8 Summary and the Path Ahead

This chapter developed AppNet, the first of the dissertation's two technical pillars: a high-level programming model for application network functions and an optimizing compiler that chooses order, side, and platform under symbolic equivalence. Its contributions are *(i)* an element abstraction expressive enough to capture widely used ANFs with far less code; *(ii)* an equivalence procedure for stateful application-network code with drops, reorderings, and arbitrary RPC streams; *(iii)* a cost-driven optimizer that cuts median RPC-processing overhead by 47%, 44%, and 42% under strong consistency and 74–83% under weak consistency; and *(iv)* a two-phase update protocol that avoids exposing any RPC to a mixed configuration.

AppNet answers half of the architectural question posed in Chapter 3. It does not remove the structural costs identified in Chapter 2—protocol termination on every hop and IPC with an out-of-process proxy. Instead, it accepts the layered stack and optimizes the policy layer above it. Even this narrower move reduces the RPC tax enough to justify compiled policy abstractions for application networks at non-trivial scale.

The other half is the stack itself: protocol termination, wire format, and encryption boundaries. The next chapter takes up that question. Where AppNet compiles *within* the existing stack, fRPC compiles *across* it.

Chapter 5

fRPC: Synthesizing Flat RPC Communication Stacks

AppNet optimized the policy layer of the application network, but deliberately left the layered RPC stack intact. Its proxies still terminate gRPC over HTTP/2 over TLS over TCP at every hop, parse each message, apply the ANF chain, and re-originate the protocol. Chapter 2 showed why that residual cost matters: in HTTP/gRPC mode, protocol parsing alone accounts for 63–77% of per-sidecar overhead. The overhead is structural: it follows from adding policy enforcement around a general-purpose stack rather than designing the stack around policy.

This chapter takes the next step. It contributes **fRPC** (*flat RPC*): a compiler that takes an RPC schema, an AppNet ANF pipeline, and a developer-supplied delivery pipeline, then synthesizes a *flat*, application-specific stack that co-designs serialization, delivery, and policy enforcement. Instead of parsing and re-originating a generic protocol stack at every hop, fRPC proxies process RPCs in flight on a custom wire format: ANF-visible bytes are directly accessible, application-owned bytes remain end-to-end encrypted, and delivery mechanisms such as reliability, ordering, congestion control, and flow control are coordinated with policy actions such as drops, retries, delays, and mutations. In our evaluation, the synthesized stack reduces application latency by up to 52% and CPU usage by up to 33% versus gRPC + Envoy with parity in semantics and security, while preserving out-of-process enforcement.

fRPC's source code is publicly available at `https://github.com/appnet-org/arpc`.

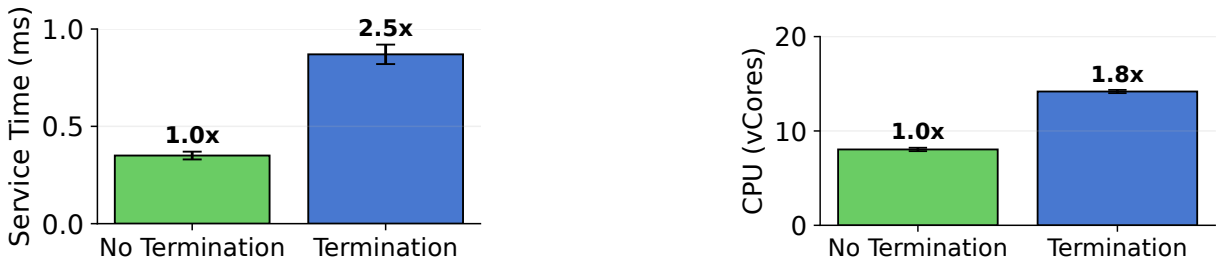
The chapter proceeds from cost to design to evidence. Section 5.1 isolates protocol termination and message buffering as the two structural costs AppNet cannot reach. Section 5.2 states the flat-stack response: developers specify an ANF pipeline and a delivery pipeline, and the compiler synthesizes the communication stack that realizes both. Sections 5.3–5.5 develop the abstractions, runtime, and compiler, including the delivery pipeline, policy-aware wire format, non-terminating proxy execution model, and shims that preserve layered-stack semantics. Sections 5.6 and 5.7 describe the implementation and evaluate fRPC against gRPC + Envoy, hand-tuned subsystem baselines, and the cost of the abstraction itself. Sections 5.8–5.10 cover limitations, related work, and the chapter's role in the dissertation's argument.

5.1 What AppNet Could Not Reach

fRPC starts from the residual RPC tax that AppNet cannot remove. Chapter 4 showed that compiling ANFs can substantially reduce policy-layer cost, but the layered stack beneath those ANFs is unchanged: gRPC, HTTP/2, TLS, and TCP still perform their generic roles on every RPC, at every hop. This section quantifies that remaining cost, decomposes it into protocol termination and message buffering, and uses the decomposition to motivate fRPC's design.

Where the remaining cost comes from. In the service mesh data path described in Section 2.2, each sidecar pays two costs before it can run policy. First, it *terminates* the protocol stack: decrypting TLS, reconstructing the TCP stream, decoding HTTP/2 frames, and parsing HTTP headers. Second, it *buffers* the message body to find the RPC fields its filters read or mutate. The costs are distinct: termination is required to split and re-originate the connection, while buffering is required because today's serialization libraries, such as Protobuf [39], expose fields only by parsing a linear byte stream. Both costs are paid per hop and are largely independent of policy complexity.

A microbenchmark that isolates the costs. To isolate the two costs (termination and buffering), we use a two-service key–value store over gRPC, replay the Meta CacheLib KV trace [51] with `wrk` [45], and vary only whether the proxy terminates the stack and whether it buffers the full message before forwarding. The proxy applies a *null* policy, so any overhead comes from termination or buffering rather than ANF logic.



(a) Service Time Overhead.

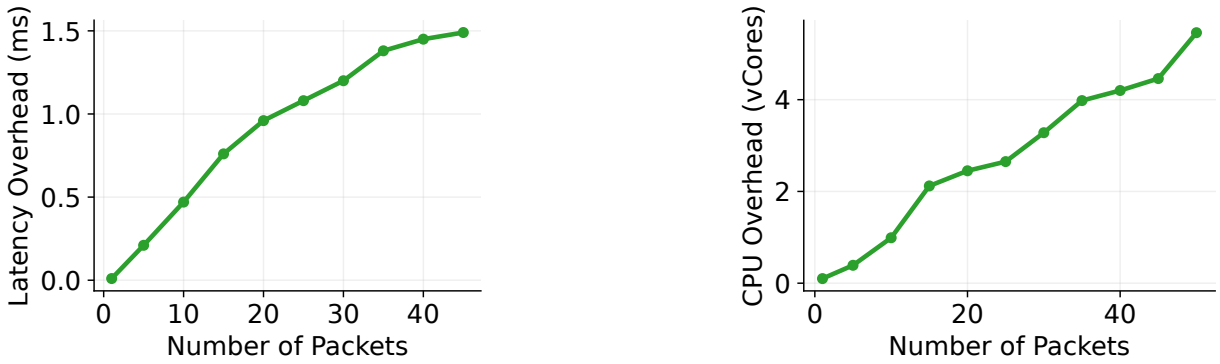
(b) CPU Overhead.

Figure 5.1: Overhead of protocol termination in today's layered stack, measured on a two-service key–value benchmark driven by the Meta CacheLib trace [51].

Termination is expensive. Figure 5.1 isolates the cost of protocol termination. Even with a null policy, a terminating proxy incurs $2.5\times$ the per-RPC service time and $1.8\times$ the per-RPC CPU of a forwarding proxy. The overhead comes from splitting one end-to-end exchange into two independent protocol contexts: separate TCP state, HTTP/2 streams, HPACK compression state, and the reassembly, demultiplexing, and re-origination needed between them. These per-layer costs compound even when the proxy performs no useful policy work.

Buffering scales with message size. Figure 5.2 measures the cost of full-message buffering on the same application. Across payloads from one to fifty packets—the production range reported by Seemakhupt et al. [115]—the overhead grows roughly linearly with message size. At fifteen packets per RPC, buffering adds about 0.8 ms of service time ($\sim 1.5\times$ the non-buffering baseline) and nearly two cores of CPU usage, mostly from allocation and copying.

Why this overhead is structural. The costs are not artifacts of one proxy implementation. They follow from composing serialization, delivery, encryption, and policy as mutually opaque layers. TCP hides RPC boundaries, HTTP parsing is required to recover request structure, Protobuf hides field offsets, and TLS hides payload bytes from any party without the session key. As a result, inspection proxies must either terminate the stack or join the trusted computing base [102, 129]. Each layer is reasonable on its own; the tax appears at their boundaries. This is why Chapter 2 found protocol parsing to dominate HTTP/gRPC sidecar overhead.



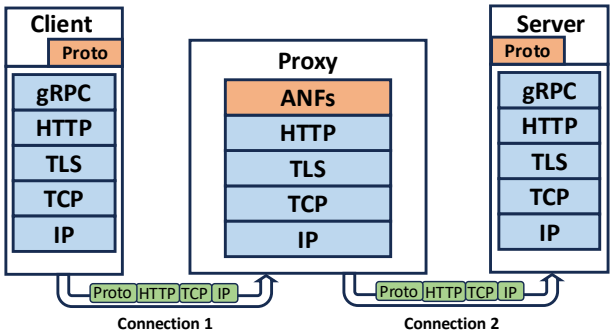
(a) Service Time Overhead.

(b) CPU Overhead.

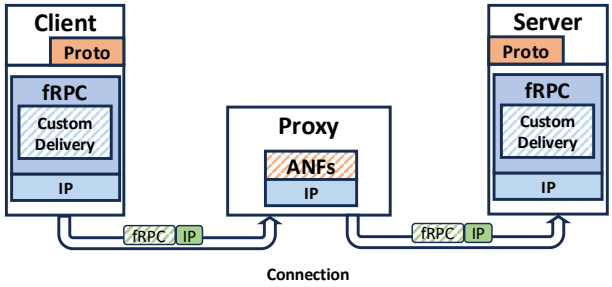
Figure 5.2: Overhead of full-message buffering versus a streaming, non-buffering baseline, as a function of message size in packets. The overhead grows linearly with message size and is paid even for a null policy.

What an ideal proxy would do. The target is therefore a proxy that is *non-terminating*, *non-buffering*, and able to access policy-visible fields in constant time without costly parsing. It should not own delivery state, split connections, or re-originate the stack; it should inspect and mutate only policy-visible bytes as they arrive, while forwarding application-owned bytes without reading them. Today's stack cannot provide those properties because the relevant bytes are buried inside compressed, serialized, or encrypted streams. fRPC changes the stack so those properties become design constraints rather than after-the-fact optimizations.

Why keep out-of-process enforcement? One alternative is to move ANFs into the application RPC library, where they can see native objects and avoid the proxy tax [90, 114]. As earlier chapters discussed, that model is useful in some settings. In most cases, however, out-of-process enforcement remains necessary because sidecars provide three properties that libraries cannot fully replace [47, 119]: policy decisions outside a compromised application, separate accounting for infrastructure cost, and a natural place for policies that depend on cross-replica state such as global rate limits, fleet-wide load balancing [114, 131], and distributed admission control. fRPC keeps those properties while removing the termination and buffering taxes that today's stack attaches to them.



(a) Today's RPC communication.



(b) fRPC's compiled stack.

Figure 5.3: Two RPC stack organizations. **(a)** Today's stack composes general-purpose layers with an out-of-process proxy that terminates and re-originates the stack on every hop. **(b)** fRPC replaces those layers with a compiler-synthesized flat stack: a policy-aware wire format, a message-aware delivery pipeline, and a non-terminating proxy that runs ANFs in flight. Blue boxes are protocol components; orange boxes are application components; hatched boxes are compiler-generated artifacts.

5.2 Compiling Across the Stack

The response is the second ADN principle from Chapter 3: *co-design across the stack*. Because termination and buffering costs arise at the boundaries between serialization, delivery, encryption, and policy, a compiler confined to any one layer cannot remove them. fRPC therefore treats the RPC schema, ANF pipeline, and delivery pipeline as inputs to one synthesis problem.

Figure 5.3 contrasts the layered baseline with fRPC's compiled stack. In the baseline, each protocol layer processes the bytes independently, and the proxy must terminate the full stack to run policy. In fRPC, the developer specifies the policy logic as an AppNet ANF pipeline and the required end-to-end semantics as a delivery pipeline. The compiler combines those specifications with the RPC schema to synthesize a custom wire format, a message-oriented delivery implementation, a non-terminating proxy that runs ANFs

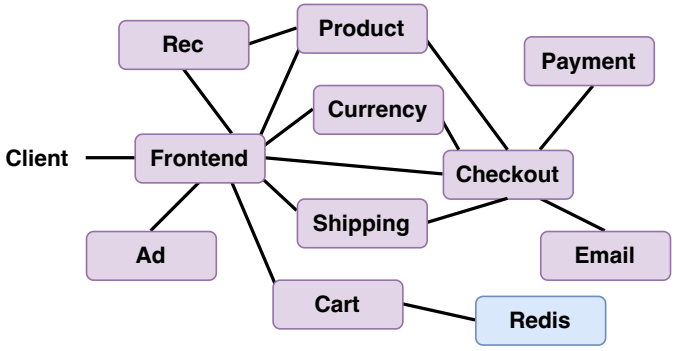


Figure 5.4: The microservice graph of the Online Boutique benchmark [73].

on directly reachable bytes, and shims that reconcile delivery actions with policy expectations. These pieces are generated together so that serialization, delivery, and policy remain consistent.

A running example. The chapter uses one example throughout: the `frontend–checkout` edge in Google's Online Boutique benchmark [73], shown in Figure 5.4 and evaluated in Section 5.7. The `frontend` sends `CheckoutRequest` messages containing `item_id`, `username`, `address`, and `item_info`. The `checkout` service is protected by two ANFs: a *firewall* that denies requests from blocked usernames and a *rate limiter* that enforces per-client request limits. The edge must also provide reliable and encrypted delivery. This example is small enough to follow end-to-end, but exercises each part of fRPC's design.

Three stages. Similar to AppNet, fRPC turns a developer specification into an executable communication stack in three stages.

Specification. The developer supplies three inputs: pointers to the application's RPC schema files (e.g., Protobuf definitions), an ANF pipeline written in the AppNet DSL of Chapter 4, and a delivery pipeline written in fRPC's event-driven delivery DSL (Section 5.3.2). In the running example, the ANF pipeline names the firewall and rate limiter, while the delivery pipeline names reliability and encryption. Together, the schema and the two pipelines define the behavior the compiler must preserve.

Compilation. The compiler turns the specification into a flat communication stack. First, it analyzes the ANF pipeline to identify which RPC fields any element reads or writes. It then emits a *policy-aware* serialization format that places those fields in a public partition, reachable without parsing the whole message, and places all remaining fields in a private partition encrypted end-to-end between the application endpoints.

Next, the compiler compares the delivery pipeline's semantic properties (for example, whether it retries, treats loss as congestion, or uses RTT as a control signal) with the ANF pipeline's interaction properties (for example, whether any element may drop a message or is sensitive to duplicate execution). When a pair would interfere in a flat stack, the compiler inserts a targeted *shim*: drop-notification for may-drop ANFs composed with loss-based reliability, deduplication for duplicate-sensitive ANFs composed with retrying delivery, or delay-feedback when ANF processing time would otherwise be interpreted as RTT. These shims make the flat stack match the developer's requested semantics without reinstating the full layered baseline.

Runtime execution. The synthesized stack runs across three components. The **fRPC library**, linked into the client and server, executes the custom serialization, the generated delivery pipeline (over UDP by default), and any in-process ANFs the compiler places in the library. The **fRPC proxy** is lightweight and non-terminating: it interposes on the byte stream, reads the public partition as soon as it arrives, executes the ANF pipeline over that partition, mutates it in place when required, and forwards the still-encrypted private partition without parsing or buffering it. The **fRPC controller** manages dynamic state—firewall denylists, rate-limit configurations, encryption keys—and coordinates rollouts of new compiler outputs using the version-tagged mechanism introduced for AppNet in Chapter 4.

Relationship to AppNet. fRPC reuses AppNet's ANF language and compiler machinery: element syntax, match-action rules over RPC fields, sub-chain organization, consistency knobs, property extraction, and placement optimization. In fRPC, an AppNet program is therefore the ANF half of the specification, supplying the field-access and interaction properties that wire-format synthesis and shim insertion depend on. This chapter adds the other half of the system: the delivery pipeline DSL, policy-aware serialization, mutation-friendly packetization, dual-key selective encryption, non-terminating proxy execution, and interference detection. The rest of the chapter follows that compiler order: specifications (Section 5.3), synthesized runtime (Section 5.4), and compiler implementation (Section 5.5).

5.3 Programming Abstractions

An fRPC specification has two interoperable, event-driven parts: an *ANF pipeline* for policies, written in the AppNet DSL of Chapter 4, and a *delivery pipeline* for end-to-end behavior, written in the DSL introduced

in this chapter. Both operate on individual RPC messages rather than packets or byte streams, giving the compiler an analyzable representation that the runtime can compose without an intervening compatibility layer.

The price of this flat composition is that policy and delivery can now interfere directly. In a layered stack, transport termination hides policy decisions behind a fresh connection: a firewall drop does not look like network loss to the sender, and a retransmission does not necessarily re-enter every policy element. In fRPC, the ANF pipeline and delivery pipeline share one synthesized stack. A policy drop can trigger loss-based retry, a retry can re-execute a rate limiter or logger, and ANF processing time can pollute a delay-based congestion signal. The abstractions below therefore expose enough structure for the compiler to detect and repair these cross-pipeline effects.

5.3.1 The ANF Pipeline

The ANF half of an fRPC specification is an AppNet program. We omit the details of the AppNet DSL here.

Elements. An ANF pipeline is a chain of *elements*. Each element may declare `state` variables, initialize them in `on_init`, and process RPCs in `on_request` and `on_response`. Element bodies use match expressions over RPC fields and state, `get`/`set` on declared state, and a small set of built-in primitives. Field reads and writes, state accesses, and drops are all explicit syntax the compiler can inspect.

Element properties. To compose ANFs with delivery modules, the compiler does not need full element semantics; it needs the parts of an element that are visible across the flat-stack boundary. It therefore extracts the fields read and written, whether the element may drop RPCs (`may-drop`), and whether duplicate execution can change behavior (`duplicate-sensitive`, analogous to AppNet's idempotence notion). The first two determine which fields must remain reachable to the proxy; the latter two identify ANF behaviors that can interfere with loss-based or retrying delivery. Extraction is conservative: read and write sets are over-approximated across all branches, and any `drop` action—including an error response—sets `may-drop`. Table 5.1 summarizes these properties for the ANFs used in the evaluation.

ANF	Reads	Writes	may-drop	duplicate-sensitive
Fault Injection			✓	
Cache	✓	✓		
Rate Limiting			✓	✓
Load Balancing	✓	✓		
Logging	✓			✓
Mutation	✓	✓		
Application Firewall	✓		✓	
Metrics	✓			✓
Admission Control			✓	✓
Bandwidth Limit			✓	✓
Circuit Breaking			✓	✓

Table 5.1: Common ANFs used in fRPC's evaluation and the properties extracted by the compiler. Reads and writes drive the public/private wire-format split; `may-drop` and `duplicate-sensitive` drive interference detection and shim insertion (Section 5.5).

5.3.2 The Delivery Pipeline

The delivery pipeline is fRPC's new contribution. Where AppNet's ANF pipeline describes the policy work the network should apply to each RPC, the delivery pipeline describes the end-to-end delivery policy the application needs for each RPC. Reliability, ordering, priority, encryption, congestion control, and flow control are common examples: which messages must not be lost, which messages must not be delivered out of sequence, how the sender responds to network feedback, and how the sender respects the receiver's processing capacity. But the pipeline is not limited to this list; it is a programmable place to define any end-to-end delivery semantics the application wants to make explicit. Today's stack provides these semantics as a fixed bundle. TCP and TLS together give applications reliable, in-order, encrypted delivery, along with TCP's loss-based congestion control and connection-scoped flow control. That bundle is broadly correct, but applications still pay for pieces they do not need: an application that does not need ordering still incurs ordering delay; an application with its own RPC-level retries also pays for TCP retransmission; and an application that does not need encrypted bulk transfer still pays the TLS handshake cost on each connection.

Example delivery element: reliable delivery. Figure 5.5 shows the client-side half of a message-level reliable-delivery module. The module keeps a `pending_buffer` keyed by `(rpc_id, seq_num)`. On send, it records the packet, emits it, and arms a per-packet retransmission timer. On receive, it treats both successful acknowledgments and explicit policy-drop notifications as terminal signals: an `ACK` means the server accepted the packet, while a `DropNotification` means a downstream ANF intentionally dropped

```
state:
  # Map: (rpcID, seqNum) -> Packet
  pending_buffer

on_send(pkt):
  # Start a timer for every packet sent
  packet_key = (pkt.rpc_id, pkt.seq_num)

  pending_buffer[packet_key] = pkt
  send(pkt)
  schedule_timer(packet_key, '1s')

on_receive(pkt):
  match pkt.type:
    # Stop retransmission on success (ACK)
    # or if the notification shim signals
    # an explicit drop
    case ACK | DropNotification:
      packet_key = (pkt.rpc_id, pkt.ack_seq)

      # Remove packet acked/dropped
      if packet_key in pending_buffer:
        pending_buffer.remove(packet_key)
        stop_timer(packet_key)

on_timer(packet_key):
  match pending_buffer[packet_key]:
    case None:
      return # Already ACKed or dropped
    case pkt:
      retransmit(pkt)
      schedule_timer(packet_key, '1s')
```

Figure 5.5: Client-side specification of reliable delivery. The server-side code that emits `ACK` packets and the packet-format definition for acknowledgments are omitted for brevity.

the RPC, so retrying would violate the policy decision. In both cases, the module removes the buffered packet and cancels the timer. If the timer fires first, the module retransmits the buffered packet and re-arms the timer. The corresponding server-side half generates the `ACK` packets; the `DropNotification` packet is the compiler-inserted shim of Section 5.5.

Pre-built modules, with extensibility. The delivery DSL ships with a small library of pre-implemented modules—reliable delivery, in-order delivery, CUBIC congestion control, credit-based flow control —each parameterized by a handful of constants (timeout, window size, priority levels). The library exists so that

the developer does not have to write a transport from scratch. But the language is open: developers can compose pre-built modules into a custom pipeline, can replace any module with a hand-written one, and can declare custom packet types whose layout the compiler emits along with the rest.

Delivery properties. As with ANF elements, the compiler statically extracts a small set of properties from each delivery module. The two that govern interference resolution are whether the module retries packets (`may-retry`) and how it interprets network signals: `loss-based` (the module reacts to packet loss as if it indicated network congestion or transport-level failure) and `delay-based` (the module reacts to round-trip-time deviations as if they indicated congestion or queue buildup). The extraction is again conservative; we defer details to Section 5.5, where the properties feed into the interference-detection procedure.

5.4 The Synthesized Runtime

The synthesized runtime turns the specifications into bytes on the wire. fRPC's runtime has three pieces: the library that endpoints link against, the proxy that processes RPCs in flight, and the wire format they share. The wire format is the central choice: it determines which bytes the proxy can inspect, which bytes remain opaque, and when an RPC can be processed without buffering the whole message.

5.4.1 Policy-Aware Split-Partition Serialization

The wire format follows from the requirement in Section 5.1: a proxy must be able to access the fields it processes without scanning the whole message, while fields that the proxy does not process should remain hidden from the network. Existing formats do not provide this combination. In Protobuf, fields are variable-length and self-describing, so a proxy must scan the message to find a field. With TLS, all bytes are encrypted under one key, so the proxy must either receive the application key or terminate, decrypt, and re-encrypt the connection.

fRPC addresses this by splitting each message into two partitions. The compiler analyzes the ANF pipeline's field reads and writes (Section 5.3.1) and produces, per message type, a layout with one partition for fields used by the proxy and one partition for the rest.

The public partition. The public partition contains the fields that any ANF in the chain reads or writes, plus routing and metadata fields. It uses a fixed table at the start of the partition. Primitive fields are stored inline in the table, so ANFs can read or write them with one indexed access. Variable-length fields, such as strings and lists, are stored in a payload region after the table, and the table stores 32-bit offsets into that region. Nested objects use the same structure, with their own table and payload region. Each variable-length field in the public partition is length-prefixed, so an ANF can update its value without parsing the rest of the message. This layout is similar to Cap'n Proto [5] and FlatBuffers [16]: it provides random access to fields. The key difference is that fRPC chooses the public fields using static analysis of the ANF specification, rather than a developer-written schema annotation.

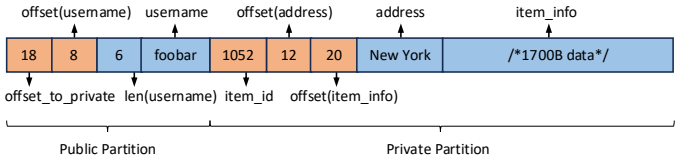
The private partition. The private partition contains all remaining fields. By default, it uses a similar table-based layout, but without the extra length fields used to support in-network updates in the public partition. A deployment can instead use a more compact layout based on Protobuf's variable-length encoding for the private partition. Because proxies never read this partition, the deployment can choose between lower bandwidth and lower serialization overhead. The private partition is encrypted end-to-end between the application endpoints and is hidden from every proxy on the path.

The link between the two. The first table slot in the public partition stores `offset_to_private`, which gives the byte offset where the private partition starts. This field also lets the proxy determine when the full public partition has arrived. If the bytes received so far cover `offset_to_private`, the proxy has all fields needed to run the ANF chain. The proxy does not parse the private partition; it forwards those bytes as they arrive.

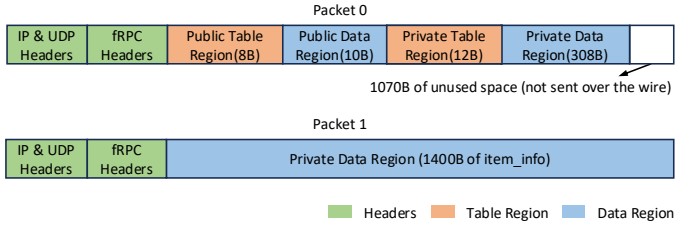
Example message. Figure 5.6a shows the schema for the running `CheckoutRequest` example, which has four fields. The compiler finds that the ANF chain only uses `username`: the firewall reads it, and the rate limiter does not read any message field. As a result, the public partition contains `username`, the header, and the `offset_to_private` slot; the private partition contains `item_id`, `address`, and `item_info`. Figure 5.6b shows the resulting serialized layout.

```
message CheckoutRequest {
    int32  item_id   = 1; # 1052
    string address   = 2; # New York
    string username  = 3; # foobar
    string item_info = 4; # A 1700B string
}
```

(a) `CheckoutRequest` message schema. Comments show example values.



(b) Serialized `CheckoutRequest` on the wire.



(c) Packet layout of the serialized message (1400-byte MSS). The private partition is right-aligned to the end of the packet allocation, leaving a slack region between the two partitions that proxies can grow into without displacing private bytes.

Figure 5.6: Lifecycle of a `CheckoutRequest` message: schema, split-partition serialization, and mutation-friendly packetization. The compiler places fields used by the ANF chain (`username`) in the public partition and encrypts the remaining fields in the private partition. `offset_to_private` marks where the private partition begins.

5.4.2 Mutation-Friendly Packetization

A proxy must not only locate `username` in constant time, but also *update* `username` (or any other public field) without re-packetizing the rest of the message. A simple layout places the public partition first and the private partition immediately after it. In this layout, growing a public field shifts every byte of the private partition by the same amount. This shift requires buffering in the worst case and packet rewriting even in common cases.

fRPC avoids this shift by placing the private partition at the *end* of the packet allocation, instead of at an offset determined by the public partition length. Let P be the size of the public partition, V the size of

the private partition, and MSS the maximum byte budget per packet after fRPC headers. The packetizer first emits $\lfloor P/\text{MSS} \rfloor$ packets that contain only public bytes; this count is usually zero because the public partition is small. It then emits a *boundary* packet. The boundary packet places the last $P \bmod \text{MSS}$ bytes of the public partition at the head and the first $V \bmod \text{MSS}$ bytes of the private partition at the tail. The bytes between them form a slack region of $\text{MSS} - (P \bmod \text{MSS}) - (V \bmod \text{MSS})$ bytes. If $P \bmod \text{MSS} + V \bmod \text{MSS} > \text{MSS}$, the boundary uses two packets and has no slack. Finally, the packetizer emits $\lfloor V/\text{MSS} \rfloor$ packets that contain only private bytes. The slack region is logical: it is not sent on the wire. It gives the proxy headroom for growing public fields.

Using the slack. A proxy that adds a tracing header, extends a routing field, or otherwise grows the public partition writes the new bytes into the slack region of the boundary packet and updates `offset_to_private`. As long as the growth fits in the slack, only the boundary packet changes, and the proxy can keep forwarding private packets as they arrive. The update is applied in flight, without buffering or re-packetizing the message.

When the slack is not enough. The packetizer does not remove all buffering risk, but it limits where the risk appears. When an update exceeds the available slack—for example, adding a 500-byte JSON Web Token [31] to a message whose boundary packet has only 200 bytes of slack—the proxy uses on-demand fragmentation. It sets the `MoreFragments` bit on the boundary packet, emits one or more extension packets for the overflow bytes, and continues to forward private packets without changes. The extension packets use the same sequence number as the boundary packet and a separate `FragmentIndex`, so the receiver can reassemble them in order without changing sequence numbers. This mechanism is similar to IP fragmentation and is hidden from upstream components.

5.4.3 Fine-Grained Selective Encryption

The split-partition layout also improves security. In today's stack, TLS encrypts the whole payload under one session key. To inspect one header, a proxy must receive that key. As a result, the proxy can decrypt any byte of any message it sees. This is the usual problem with layered stacks: every proxy on the path becomes part of the trusted computing base (TCB) for the payload. mcTLS [102] and MADTLS [129] reduce this

exposure by using different keys for different byte ranges. However, they treat the message as a byte stream rather than as application fields. This makes the mapping from bytes to fields hard to maintain and reduces the benefit for variable-length application data, where systems often need per-slice records or templates.

fRPC uses the same idea, but applies it to the compiler-generated layout. Each partition is a separate AES-GCM envelope. The public partition is encrypted with `key_public`, which the controller gives to authorized proxies and application endpoints. The private partition is encrypted with `key_private`, which the controller gives only to application endpoints. Each envelope has its own authentication tag. When an ANF modifies the public partition, it verifies the public envelope, updates the public fields, and recomputes the authentication tag without accessing `key_private`.

Security benefit. A compromised proxy that holds `key_public` can read or modify `username` and any other field in the public partition. It cannot read `address`, `item_info`, or any other private byte. Compared with TLS, fRPC reduces the payload TCB: a proxy with the public key is trusted only for the public fields it needs for policy, not for the full payload. The cost, measured in Section 5.7.4, is one extra AES-GCM envelope per message.

5.4.4 The Non-Terminating Streaming Proxy

The wire format and packetization define how fRPC represents messages on the wire. The remaining component is the proxy: a small data plane that runs the synthesized ANF chain and processes RPCs as packets arrive.

Per-packet processing. Figure 5.7 shows the proxy's per-packet control flow. When a packet arrives, the proxy looks up the `rpc_id` in its verdict cache. If a verdict (forward, drop, hold) is already known for this RPC, the proxy applies the verdict immediately. This case occurs when the ANF chain has already run on this RPC's public partition, possibly on an earlier packet. Otherwise, the proxy reads `offset_to_private` from the public partition header and compares it with the bytes received so far. If the public partition is complete, the proxy runs the ANF chain, caches the verdict under the `rpc_id`, and handles the packet using that verdict. If the public partition is incomplete, the proxy stores the public bytes received so far in a small per-`rpc_id` cache and waits for the next packet of this RPC. Private packets that arrive out of order be-

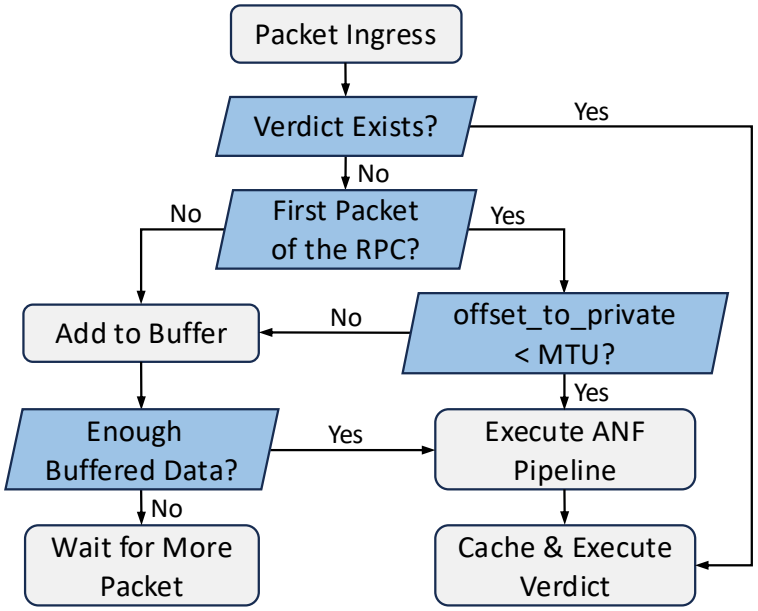


Figure 5.7: Control flow of the fRPC proxy on the receive path. The fast path applies when a verdict has already been computed for this `rpc_id`, or when the first packet contains the entire public partition. In this case, the proxy runs the ANF chain immediately and forwards the still-encrypted private partition as it arrives. The slow path applies when the public partition spans multiple packets or arrives out of order. In this case, the proxy temporarily buffers public bytes until it can run the ANF chain.

fore the public partition is complete are also held briefly. Once the public partition is complete, the verdict applies to the cached bytes and to all later bytes of this RPC.

Buffering. The proxy never buffers a full message. It buffers only the public partition bytes, which typically fit in one packet, and any private packet that arrives before its public predecessor. Such reordering is rare when ECMP preserves per-flow order. The verdict cache is bounded by the number of RPCs in flight, which is typically small compared with the total number of bytes in flight. A layered stack buffers the full message to parse it; fRPC buffers only the bytes needed by the policy chain.

Packet reordering. fRPC does not require packets to arrive in order. If a private packet arrives before its public predecessor, the proxy holds it briefly and forwards it once the public partition is complete. However, the fast path is most effective when the public partition arrives before or with the first private packet. This is the common case in datacenters, where ECMP flow hashing preserves per-flow order.

Proxy placement. The fRPC proxy currently runs in user space and intercepts traffic using `iptables` redirection, as in common service meshes [29, 44]. User space is an implementation choice, not a design requirement. Because the synthesized stack is specified at a high level, the same compiler output could target a kernel data path such as eBPF or eTran [58], or a SmartNIC. Chapter 6 discusses these targets.

5.5 The fRPC Compiler

Two specifications written separately may not compose in a flat stack. Today's layered stack hides this issue by terminating protocols at each hop. Termination separates policy actions, such as drops, retries, and mutations, from delivery mechanisms, such as reliability and congestion control. fRPC removes this termination, so these components can interact directly: a firewall drop can look like network loss to reliable delivery; a retransmission can look like a new request to a rate limiter; and ANF processing time can look like queueing delay to delay-based congestion control. The fRPC compiler handles these cases statically. It detects interference patterns, identifies the relevant properties of the ANF and delivery pipelines, and inserts a small shim without changing the developer's specifications.

This section describes the compiler in two steps. First, semantic modeling extracts a small set of properties from each ANF and delivery module. Second, interference detection matches these properties against known patterns and inserts the corresponding shim.

5.5.1 Semantic Modeling

The compiler first builds a property table for each pipeline. For ANFs, it reuses AppNet's property extractor. For delivery modules, it uses a similar extractor over each module's event handlers.

Properties of ANF elements.

- `reads(f)` and `writes(f)` for each RPC field f: the set of fields the element references in its `on_request`/`on_response` bodies (under the over-approximation described in Chapter 4).
- `may-drop`: the element's body has a syntactic path that calls `drop` or sends an error response in place of forwarding the RPC.

- `duplicate-sensitive`: executing the element twice on the same input can produce a different result than executing it once.

Properties of delivery modules.

- `may-retry`: the module's `on_timer` body (or a body otherwise reachable from `on_send`) calls `retransmit` or its equivalent on a previously buffered packet.
- `loss-based`: the module treats a missing packet or timeout as evidence of network congestion or transport failure.
- `delay-based`: the module uses RTT measurements and treats large RTTs as evidence of congestion or queueing.

These properties are conservative, as in AppNet. For `duplicate-sensitive`, the implementation uses symbolic execution with Z3 to check whether two executions of an ANF on the same input can differ in state, output RPC, or verdict; if so, the ANF is tagged as duplicate-sensitive. For delivery modules, the compiler checks for retransmission calls and for the loss- and RTT-oriented hooks that identify loss-based and delay-based behavior. The interference detector then uses this small property table rather than modeling the modules' numeric logic.

5.5.2 Interference Patterns

The compiler then matches the property tables against a small set of interference patterns. Each pattern is a pair of ANF and delivery properties that can make the synthesized stack differ from a layered baseline. Figure 5.8 summarizes these patterns.

Pattern 1: may-drop ANF $\times$ loss-based delivery. An ANF may drop an RPC for policy reasons, such as a firewall rejecting a user or a rate limiter denying a request. A loss-based delivery module may interpret the missing packet as network loss, retransmit it, and reduce its congestion window. Each module behaves correctly in isolation, but the combined behavior violates the policy. In our running example, the firewall and reliable-delivery combination triggers this pattern.

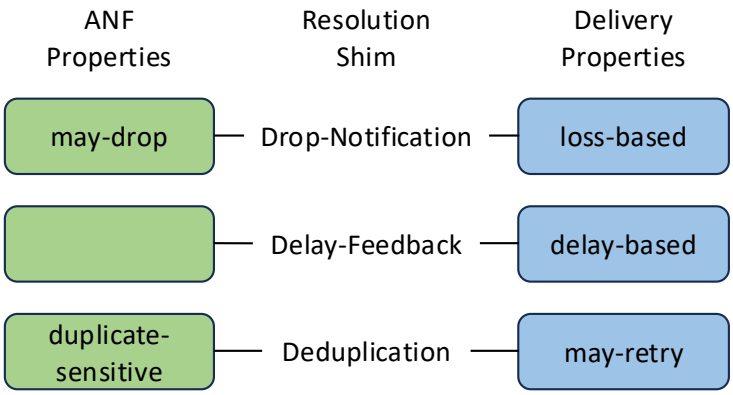


Figure 5.8: Interference patterns detected by fRPC's compiler. Each pattern is a pair of ANF and delivery properties that triggers a resolution shim.

Resolution: drop-notification shim. When the ANF drops an RPC, the proxy sends a `DropNotification` upstream. The reliable-delivery module treats this packet as a final verdict, removes the matching buffer entry, and does not retransmit. The congestion-control loss handler is not called. The compiler emits both sides of the shim: the proxy signal and the `DropNotification` handler in the delivery module.

Pattern 2: may-retry delivery $\times$ duplicate-sensitive ANF. A delivery module may retransmit a packet after a loss signal. The duplicate RPC can then run through the ANF chain again. A duplicate-sensitive ANF, such as a rate limiter, metrics element, or admission controller, may count both executions. This can over-count requests, inflate metrics, or consume extra tokens. In our running example, the reliable-delivery and rate limiter combination triggers this pattern.

Resolution: deduplication shim. The compiler inserts a deduplication shim before each duplicate-sensitive ANF. The shim is keyed by `rpc_id` and suppresses repeated executions for the same RPC. Its window is bounded by the delivery module's retry timer. The shim is a small hash set with sliding eviction.

Pattern 3: any-ANF $\times$ delay-based delivery. A delay-based congestion-control module estimates congestion from RTT. In fRPC, the sender's RTT includes time spent running the ANF chain at the proxy, even though that time is not network delay. The module may then reduce its rate for the wrong reason. This pattern applies whenever the delivery pipeline includes a delay-based module, such as a Vegas-style controller.

Resolution: delay-feedback shim. The compiler emits a proxy shim that records ANF processing time in a public header field. The delay-based module subtracts this value from its RTT sample to estimate network RTT. The shim records a timestamp on chain entry and updates the field on chain exit.

5.6 Implementation

fRPC has three components: a Go library and proxy (about 14K lines), a Python compiler (2K lines), and code-generation support (2.5K lines).

Compiler. The compiler is written in Python and builds on the AppNet frontend (Chapter 4). It reuses AppNet's parser, IR, and property-extraction passes for ANFs, and adds a new frontend for delivery modules. It emits Go code for the library and proxy, plus Kubernetes deployment scripts.

Library. The fRPC library implements serialization, the delivery pipeline, and in-process ANFs. It is API-compatible with Protobuf and gRPC-Go, so applications only need to regenerate RPC stubs. The serialization layer uses the public-partition layout from Section 5.4.1, with optional Protobuf encoding for the private partition. The delivery layer dispatches `on_send`, `on_receive`, and `on_timer` events through one event loop per endpoint.

Proxy. The proxy implements the streaming control flow in Figure 5.7. Deployment scripts redirect traffic to the proxy with `iptables`. The verdict cache is bounded and evicts entries when the corresponding RPC completes. The proxy can host multiple ANF chains and forward, drop, or hold packets per `rpc_id`.

5.7 Evaluation

Our evaluation answers three questions. *End-to-end:* how does fRPC compare to gRPC + Envoy, under the same policy and security settings, on realistic microservice applications? *Serialization and encryption:* can fRPC's policy-aware, partitioned serialization and encryption compete with standard formats and TLS-style encryption? *Delivery:* does the synthesized delivery layer achieve performance parity with hand-optimized

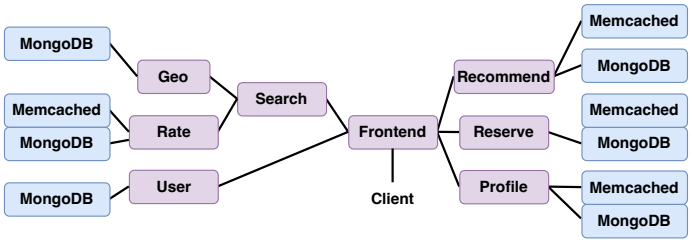


Figure 5.9: The seventeen-microservice topology of the Hotel Reservation benchmark [69], used alongside Online Boutique in the end-to-end study.

protocols? AppNet's evaluation already shows that compiled ANFs are competitive with hand-developed ones (Chapter 4); we omit that comparison from this chapter for brevity.

5.7.1 Experimental Setup

All experiments run on CloudLab [64] nodes with dual 16-core Intel Xeon Gold 6142 CPUs (2.6 GHz) and 384 GB of RAM. The software environment uses Ubuntu 20.04 (Linux 5.4.0), Kubernetes 1.28.15, and gRPC 1.72.2. We disable CPU Turbo Boost, C-states, and dynamic frequency scaling to reduce measurement variance. We use `wrk` [45] and `wrk2` [46] for load generation and high-precision latency measurement.

We evaluate fRPC using three microservice benchmarks.

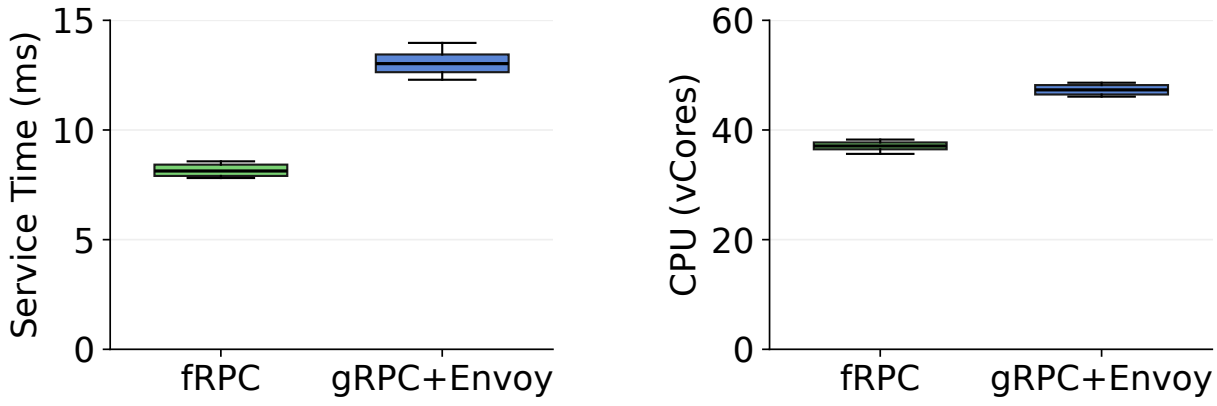
Online Boutique (Figure 5.4) is an eleven-microservice e-commerce application from Google [73]. We port all services to Go for compatibility with the current fRPC prototype.

Hotel Reservation (Figure 5.9) is a 17-microservice travel-booking application from DeathStarBench [69]. It has a more complex call graph, including geolocation-based search and recommendation logic that produces larger RPCs than Online Boutique.

Key-Value Store is a two-service benchmark with a frontend and a backend. We use it to evaluate serialization, encryption, packetization, and delivery in isolation. We replay the Meta CacheLib trace [51]; its skewed message-size distribution is typical of production caching workloads.

5.7.2 End-to-End Application Performance

We deploy each application twice: once with gRPC + Envoy and once with fRPC. We compare end-to-end performance under matched policy and security settings. For fRPC, we configure the delivery pipeline to



(a) Service time.

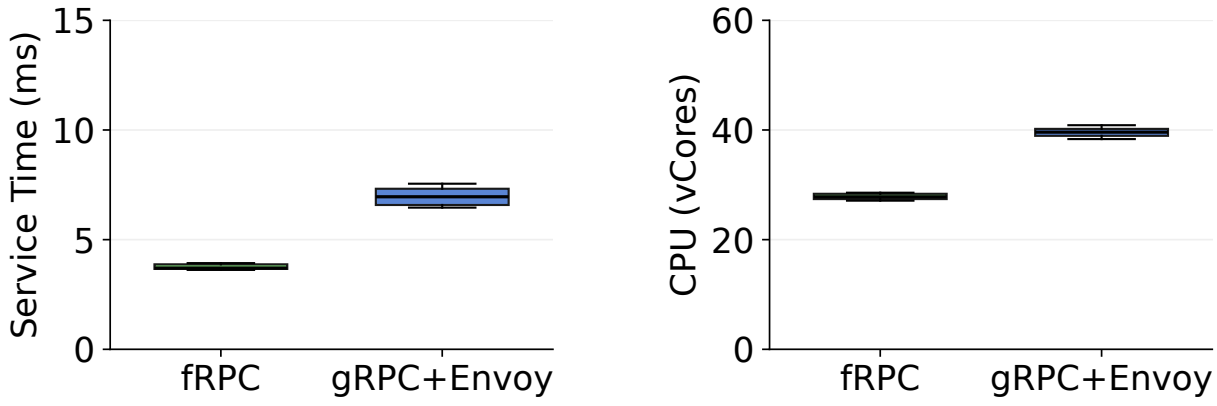
(b) CPU usage.

Figure 5.10: End-to-end performance of Online Boutique under fRPC and gRPC + Envoy, across 30 randomly sampled ANF chains per edge. Boxes show median and interquartile range; whiskers extend to $1.5\times$ IQR. fRPC achieves up to a 42% service-time reduction (median 38%) and up to a 27% CPU reduction (median 22%) while preserving the same semantics and security guarantees.

provide TCP-equivalent transport features: reliability through the reliable-delivery module of Section 5.3.2, CUBIC congestion control, and credit-based flow control. We also enable dual-key selective encryption. For gRPC + Envoy, we use the standard gRPC-over-HTTP/2-over-TLS-over-TCP stack. In both deployments, each microservice runs with a sidecar proxy. We generate ANF chains from the eleven common ANFs in Table 5.1. Following the AppNet evaluation in Chapter 4, we sample 30 random three-element chains independently for each communication edge.

We report two metrics: *service time*, the end-to-end user-request latency under minimal load (at most one outstanding request), and *CPU usage*, the total number of virtual cores used by the proxies and services.

Figures 5.10 and 5.11 show the results. On Online Boutique, fRPC reduces service time by up to 42% (median 38%) and CPU usage by up to 27% (median 22%). On Hotel Reservation, fRPC reduces service time by up to 52% (median 46%) and CPU usage by up to 33% (median 30%). The gains are larger for Hotel Reservation because its longer call paths increase the impact of per-hop overheads. The results are stable across the 30 ANF chains, with interquartile ranges of a few percentage points around the medians. Chains with larger gains tend to include may-drop ANFs; fRPC's drop-notification shim avoids retransmissions for messages that the ANF has already dropped.



(a) Service time.

(b) CPU usage.

Figure 5.11: End-to-end performance of Hotel Reservation under fRPC and gRPC + Envoy. fRPC achieves up to a 52% service-time reduction (median 46%) and up to a 33% CPU reduction (median 30%).

Source of the savings. Most of the gains come from avoiding protocol termination at each proxy hop. The split-partition wire format lets the proxy access `username` and other public fields in flight, eliminating the parse-modify-reserialize cycle that Chapter 2 found accounts for 63–77% of HTTP/gRPC sidecar cost. Dual-key selective encryption avoids redundant decryptions and re-encryptions at each hop. The non-terminating streaming proxy avoids per-hop TCP state. Together, these effects explain the end-to-end improvements.

5.7.3 Serialization

To evaluate fRPC's serialization format, we compare it against three industry-standard formats: Protobuf [39], FlatBuffers [16], and Cap'n Proto [5]. Protobuf optimizes for message size, while FlatBuffers and Cap'n Proto optimize for speed using zero-copy random access. We measure serialization latency, deserialization latency, and serialized message size using the schema definitions from Online Boutique and Hotel Reservation. Online Boutique has 26 distinct message types, and Hotel Reservation has 14. We port the original Protobuf schemas to equivalent Cap'n Proto and FlatBuffers schemas, and use Go implementations for all four formats. To produce realistic payloads, we instrument the services to record inter-service RPCs while driving each benchmark with 100K randomly generated end-to-end requests, repeated across the ANF pipelines above.

Figures 5.12 and 5.13 show the results. fRPC's default format produces messages similar in size to Flat-

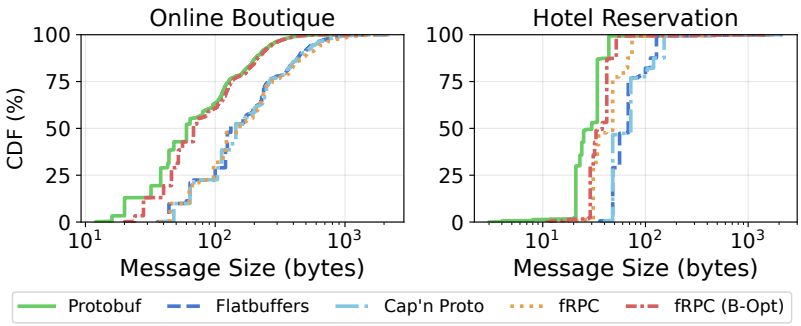


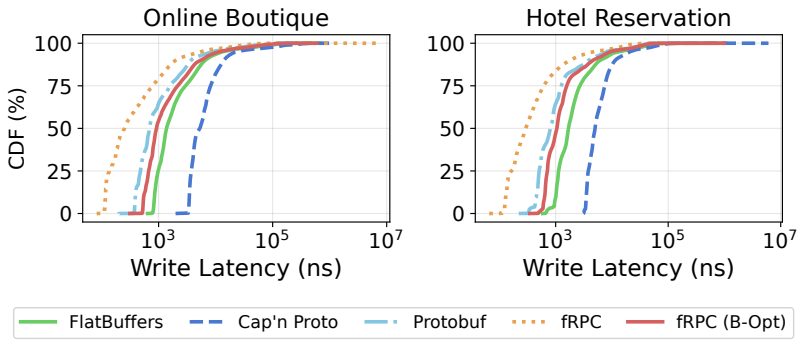
Figure 5.12: CDF of serialized message size across formats for Online Boutique and Hotel Reservation. fRPC's default serialization (table-based for both partitions) produces messages similar in size to FlatBuffers and Cap'n Proto. fRPC's byte-optimized (B-Opt) variant, which uses Protobuf encoding for the private partition, produces sizes within a few percent of Protobuf.

Buffers and Cap'n Proto, the two formats that also prioritize speed over compactness, and all three produce larger messages than Protobuf. fRPC is competitive with FlatBuffers on serialization and deserialization latency because its table-based layout provides direct field access. The byte-optimized variant produces sizes nearly identical to Protobuf while retaining the public partition's efficient in-network processing. Overall, fRPC avoids the usual bandwidth–compute trade-off of existing formats: developers can use the default format for speed or the byte-optimized variant for compactness.

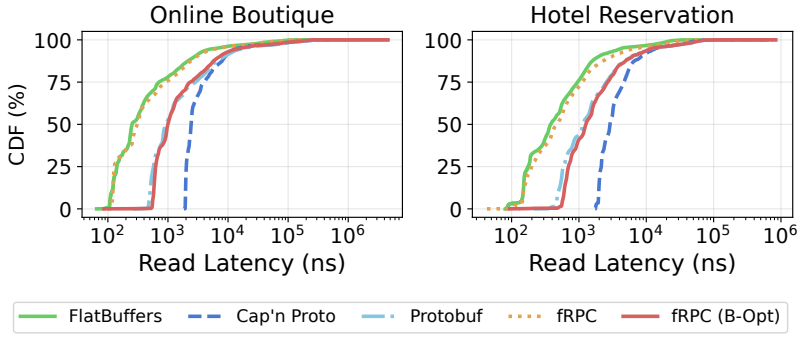
5.7.4 Encryption

Selective encryption splits each message into two AES-GCM envelopes, one per partition. This design reduces the proxy TCB (Section 5.4.3) but adds a fixed AEAD-invocation cost. We quantify this cost on the Online Boutique and Hotel Reservation corpora, using the Go standard library AES-GCM implementation, and compare against standard TLS-style single-envelope encryption. In the TLS-style baseline, the full RPC payload is encrypted and decrypted at the end host and at both sidecars. In fRPC, the public partition is encrypted and decrypted at each hop, while the private partition is processed only at the endpoints.

Figure 5.14 shows the results. On these benchmarks, which mostly have small messages, the fixed cost of invoking AES-GCM separately for the public and private partitions outweighs the savings from processing fewer bytes: fRPC adds up to 388 ns for Online Boutique and up to 401 ns for Hotel Reservation, or 25–31% relative overhead. Decryption shows a similar trend. For larger messages, such as the Meta CacheLib trace where private payloads are large relative to the public partition, fRPC reduces cryptographic work and



(a) Serialization latency.



(b) Deserialization latency.

Figure 5.13: CDF of serialization and deserialization latency for fRPC, Protobuf, FlatBuffers, and Cap'n Proto on Online Boutique and Hotel Reservation messages. fRPC's table-based layout is competitive with FlatBuffers on both operations while admitting the policy-aware split that the rest of the system depends on; Protobuf is the slowest because its variable-length encoding requires linear scanning.

outperforms the single-envelope baseline. Compared with the larger end-to-end gains in Section 5.7.2, the few-hundred-nanosecond cost for small RPCs is comparatively minor.

5.7.5 Mutation-Friendly Packetization

The packetization heuristic adds logical slack between the public and private partitions. This slack is useful when it is large enough to support in-place proxy mutations. We measure the available slack on the Meta CacheLib trace, considering only multi-packet messages, which are the target of the heuristic. We assume the cache key is in the public partition and the value is in the private partition.

Figure 5.15 shows the slack-space distribution. More than 75% of boundary packets have at least 150 bytes of slack. Nearly all packets can fit GPS coordinates, 96% can fit a W3C Trace Context header, and 56% can fit a 500-byte JWT without fragmentation. For larger mutations, the on-demand fragmentation path

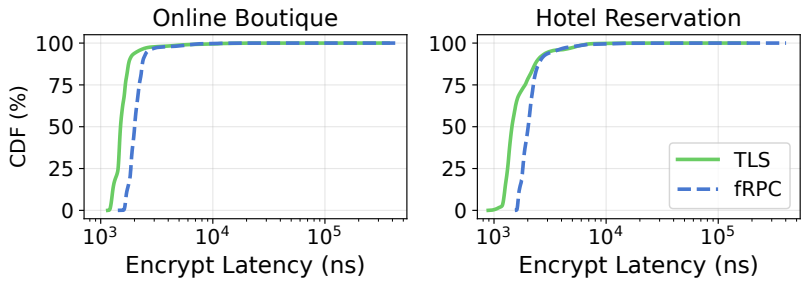


Figure 5.14: CDF of encryption latency for fRPC's dual-key selective encryption versus standard TLS-style single-envelope encryption, on Online Boutique and Hotel Reservation messages. On these workloads—whose messages are small—fRPC adds up to 388 ns for Online Boutique and up to 401 ns for Hotel Reservation, a 25–31% relative increase.

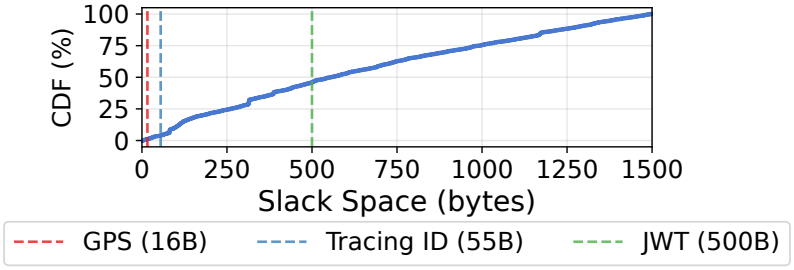
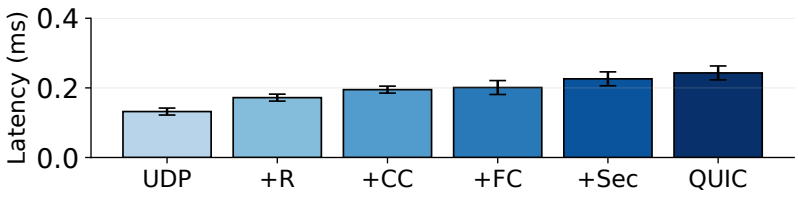


Figure 5.15: CDF of per-boundary-packet slack space in the Meta CacheLib trace. Vertical dashed lines mark three common mutation sizes: 16 bytes (GPS coordinates), 55 bytes (a W3C Trace Context header [128]), and 500 bytes (a JSON Web Token [31]). More than 75% of boundary packets have at least 150 bytes of slack; nearly all accommodate GPS coordinates, 96% accommodate Trace Context, and 56% accommodate full-sized JWTs without on-demand fragmentation.

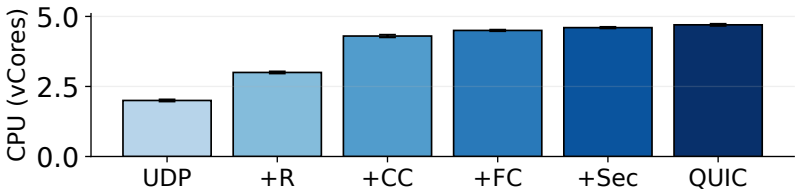
of Section 5.4.2 handles the overflow deterministically, without buffering the full message. Thus, in-place mutation is the common case, while fragmentation handles the remaining cases.

5.7.6 Delivery

The delivery DSL of Section 5.3.2 is designed to provide a pay-for-what-you-use model: an application that needs only best-effort delivery should pay only for best-effort delivery, and each added feature should add only its own cost. We test this on the Key-Value Store benchmark by compiling a sequence of delivery pipelines: UDP, +Reliable, +CongestionControl, +FlowControl, and +SelectiveEncryption. We compare the full pipeline, which provides TCP+TLS-equivalent semantics, against the Go implementation of QUIC [40], the closest hand-optimized baseline with the same set of features. For a fair comparison, fRPC's congestion-



(a) Latency.



(b) CPU.

Figure 5.16: Latency and CPU of fRPC's synthesized delivery pipelines as features are added incrementally, compared against quic-go's hand-tuned implementation of QUIC. Each feature adds only the cost of that feature; the full pipeline (with reliability, congestion control, flow control, and selective encryption) matches quic-go's performance within experimental variance.

control and flow-control modules use the same CUBIC and credit-based algorithms as quic-go.

Figure 5.16 shows the results. Each added feature adds only the expected cost: +Reliable adds buffering and retransmission, +CongestionControl adds congestion-window updates, +FlowControl adds credit accounting, and +SelectiveEncryption adds AEAD operations. The full pipeline matches quic-go's latency ($\sim$0.22 ms per message at 10K messages/s) and CPU usage ($\sim$4.7 vCPUs) within experimental variance. This result shows that the compiled delivery code achieves performance parity with a hand-optimized implementation. Applications that need fewer features than QUIC, such as a cache that tolerates loss or a logging service that does not need ordering, can configure the delivery pipeline to include only the features they need.

5.8 Discussion and Limitations

External interoperability. fRPC replaces the standard layered stack with an application-specific stack. External clients running gRPC over HTTP/2 therefore cannot connect directly to an fRPC service. As discussed in Chapter 3, deployments can route ingress traffic through a translation gateway, similar to a Kubernetes Ingress Controller [24], that terminates standard connections and converts them to fRPC traffic.

Services that must expose a public API can also run a standard gRPC socket alongside the fRPC socket.

Coupled upgrades. Because fRPC generates the wire format from the application schema and ANF specification, schema and ANF changes must be deployed with the proxies and endpoints that use them. This requirement is similar to Protobuf schema updates, but it also applies to the network: if an ANF reads a new public field, the proxy must understand that field. fRPC uses the version-tagged rollout protocol from Chapter 4, extended to cover both proxies and endpoints. This prevents mixed configurations, at the cost of coordinating each update.

Out-of-order delivery. fRPC does not require in-order packet delivery. The proxy buffers public bytes until the public partition is complete, and the verdict cache makes the per-RPC decision independent of which packet completes the public partition. The common fast path is when the first packet contains the full public partition and ECMP preserves per-flow ordering. Frequent packet reordering can slow down delivery, but does not affect correctness.

5.9 Related Work

This section focuses on prior systems closest to fRPC. Chapter 3 already gives the broader argument for compiling application-specific communication stacks from high-level specifications.

Application-level framing. Application Layer Framing (ALF) argues that network processing should operate on application objects rather than transport packets [60]. Later ALF systems applied this idea in specific settings [59, 75, 89]. fRPC follows the same principle, but adds a compiler that generates the wire format, encryption boundaries, and delivery path from the RPC schema and ANF specification.

End-host transport optimization. NetBlocks [55], eTran [58], Snap [100], and specialized host stacks [99] improve RPC delivery by changing the host networking stack. These systems optimize packet processing, kernel paths, or transport mechanisms. fRPC is complementary: it combines delivery with the ANF policy chain, and the compiler handles interactions such as drops, retransmissions, and duplicate-sensitive ANFs.

ANF and service mesh optimization. Cilium [6], Canal Mesh [119], ServiceRouter [114], AppNet [140], Lyra [71], ClickINC [132], and NetRPC [137] reduce the cost of service mesh or ANF processing. They move policies into the kernel, into the application, or onto accelerators. fRPC targets a different part of the stack: it changes the RPC wire format and delivery path so that ANFs can run without full protocol termination.

Middlebox-aware encryption. mcTLS [102] and MADTLS [129] let middleboxes access authorized parts of an encrypted byte stream. fRPC also exposes only the bytes a proxy is allowed to read, but it derives those bytes from the compiler-generated message layout. This avoids per-deployment templates and adds only one extra AEAD invocation per message.

5.10 Summary

This chapter presented fRPC, the second main system in this dissertation. While AppNet optimizes the ANF layer, fRPC targets the entire RPC communication stack. fRPC provides four mechanisms: *(i)* a policy-aware split-partition wire format that exposes only the fields needed by ANFs and keeps the rest end-to-end encrypted; *(ii)* a streaming proxy that runs ANFs once the public partition is available, without parsing the private partition or terminating transport state; *(iii)* mutation-friendly packetization that lets common header updates happen in place; and *(iv)* an interference-detection pass that inserts shims such as drop-notification, deduplication, and delay-feedback when delivery semantics and policy semantics interact.

Together, AppNet and fRPC cover the two main sources of RPC overhead identified by MeshInsight (Chapter 2). AppNet reduces the cost of ANF processing, while fRPC reduces the cost of the layered RPC stack. The end-to-end results show that fRPC reduces latency by up to 52% and CPU usage by up to 33% compared with gRPC + Envoy, while preserving the same security and delivery semantics.

Chapter 6

Discussion and Future Work

The previous chapters establish the main result of this dissertation. The layered RPC stack adds a measurable RPC tax, and the Application Defined Networks (ADN) model reduces this tax by compiling the RPC stack from a specification. AppNet and fRPC show that this idea can be built as a compiler framework with a common IR, equivalence procedure, and cost model.

This chapter discusses what remains beyond the prototype. The current system uses a simple cost model, runs in a user-space proxy over a UDP datapath, and assumes one administrative domain. A more complete ADN system should model contention, target kernel and hardware datapaths, and handle traffic that crosses an organizational boundary.

The chapter covers three areas of future work and then states the main limits of the framework. Section 6.1 discusses cost modeling under contention. Section 6.2 discusses compiler backends for kernel datapaths, SmartNICs, and programmable switches. Section 6.3 discusses ingress, egress, and cross-organization calls. Section 6.4 states the limits of the current framework.

6.1 Better Cost Modeling Under Contention

MeshInsight models the RPC tax by measuring each datapath component in isolation and adding the costs together. This keeps the model simple. It can explain where overhead comes from and predict the effect of many design changes without testing every full application. It is also a good starting point for the ADN optimizer, because the compiler needs a cost for each possible placement.

The model is a lower bound, not a full performance model. At high load, components compete for shared resources [91, 94]. Filters can share and evict cache lines. Sidecars and applications can compete for CPU time. Kernel queues can build up. A small increase in service time can become a large increase in tail latency once the system is close to saturation. MeshInsight leaves these effects out, so its predictions are most reliable at moderate load and should be treated as a floor near saturation.

A useful next step is to make the cost model aware of contention. The profiler could measure each component at several utilization levels, not only under isolated conditions. The model could add queueing terms for shared CPU cores, shared kernel queues, and shared proxies. It could also model average latency and tail latency separately, since a placement that saves CPU may still hurt p99 latency if it creates a bottleneck. Runtime telemetry could update the cost profile when the workload or deployment changes. In that design, MeshInsight would not only calibrate the compiler once; it would keep the cost model current as load shifts over time.

6.2 Compiling onto Kernel, NIC, and Switch Substrates

The evaluated AppNet and fRPC prototypes run entirely in user space, as an RPC library and a proxy. The proxy is a Go process, the default delivery pipeline uses UDP through the host socket API, ANFs run as proxy plugins, and the wire format is parsed and generated in user-space code. This is a prototype choice, not a requirement of the architecture. As Chapter 2 shows, IPC, system calls, and user–kernel crossings still account for a large part of the per-RPC cost after protocol parsing has been removed. A natural next step is to compile ADN programs to targets closer to the packet path: eBPF in the kernel, SmartNICs and FPGAs at the NIC, and P4 switches in the rack.

These targets share the same trade-off. They can avoid host overhead, but they often provide less compute, memory, and runtime support than a user-space process. eBPF [79] is the most flexible of these targets because it has maps, helpers, and a C-like programming model, but it must pass the kernel verifier. SmartNICs and switches are stricter: they offer fewer cycles per packet, smaller state, and more fixed pipelines. ADN should therefore treat offload as a compiler decision. The developer specification stays the same, while each backend adds two target-specific inputs: a measured cost profile and a check that says which elements the target can run.

6.2.1 Kernel Datapaths via eBPF

The first target is the host kernel. In the prototype, each RPC still enters a user-space proxy, so bytes cross the user–kernel boundary and pass through the host socket API before ANFs run. eBPF [79] lets applications install verified packet-processing code in the kernel datapath, before or near the socket layer, using hooks such as XDP, TC, and socket filters. eTran [58] shows that transports such as Homa and TCP can be implemented as modular eBPF programs attached to ingress, egress, and timer hooks. For ADN, eBPF is therefore more than a faster filter path: it is a backend for moving the fast path of ANFs and delivery modules into the kernel.

Past attempts at ANF offload were difficult because offloaded code first had to recover application data and maintain cross-packet state. In the normal layered stack, an ANF may need to terminate the transport, parse HTTP and gRPC, decode Protobuf, and buffer the message before it can read one field. ADN removes much of that work. AppNet tells the compiler which fields and state each ANF needs. fRPC places those fields in a public partition that can be read without parsing or buffering the full RPC. Its proxy is also non-terminating, so offloaded code does not need to maintain transport state created by breaking and re-creating connections.

ADN's language constructs also map naturally to eBPF constructs. An eBPF backend could map ADN events to kernel hooks: **request** and **response** events to tail-called programs on socket-layer hooks; **send** and **receive** events to TC or XDP programs on the egress and ingress paths; and **timer** events to the timer hooks used by eTran. The compiler would emit eBPF C only for elements that satisfy verifier limits on bounded loops, stack size, instruction count, helper calls, and pointer use. Some ANFs will not fit those limits, so the compiler must decide which elements can be offloaded instead of treating kernel execution as an all-or-nothing choice.

Some elements may also need to be split. As in eTran, the fast data-plane path can run as eBPF code, while the control loop that updates maps, refills buckets, or reacts to load remains in a user-space process. The controller would then load, attach, update, and roll back eBPF programs using the versioned rollout protocol from Chapter 4. These steps do not require changing ADN's equivalence procedure. The eBPF verifier checks safety properties of one program; the ADN compiler checks equivalence properties of a composition of programs. The two checks are complementary.

6.2.2 SmartNICs and Programmable Switches

The second class of targets is hardware. Modern SmartNICs, such as Mellanox BlueField and Pensando DPUs, provide programmable cores near the NIC datapath. Many top-of-rack switches provide P4 data-planes [53]. Prior systems have already placed network functions on these targets. SilkRoad [101] runs layer-4 load balancing in switch hardware. FlowBlaze [106] supports stateful packet processing on programmable NICs. ClickNP [93] maps the Click model to FPGAs. Other systems offload specific ANFs to SmartNIC datapaths [62, 77, 135]. ADN does not claim that offloading a single ANF is new. Its goal is to make placement a compiler decision instead of a manual operator choice.

The same parsing and buffering problem is even sharper in hardware. A P4 switch has a limited parser and a fixed pipeline. A SmartNIC is more flexible than a switch, but usually less flexible than eBPF on the host. It has limited memory, limited per-packet time, and a smaller runtime than a host CPU. Asking these devices to parse a full RPC stack, buffer large messages, or maintain transport state is usually a poor fit.

ADN changes the work that hardware has to do. Because fRPC exposes the ANF-visible fields in a public partition, a hardware backend can generate a small parser for that partition, read fields at fixed offsets or from simple length-prefixed regions, and forward the private payload without touching it. Because the proxy is non-terminating, the hardware does not need to own TCP, HTTP/2, or gRPC connection state just to run a policy. This makes simple data-plane ANFs, such as routing decisions, access checks over fixed fields, counters, and small rewrites, better fits for hardware.

The limits still matter. Some ANFs need large maps, timers, complex string processing, cryptography, or updates that grow a message past the available slack. Some need control loops, such as rate-limit refill, load measurement, or policy updates, that should stay in a host process or controller. The hardware backend therefore needs the same kind of check as the eBPF backend, but with stricter rules. If the hardware cannot parse the needed fields, hold the needed state, or finish the work within its pipeline budget, the compiler should leave that ANF in eBPF, a proxy, or the RPC library. If only the fast path fits, the compiler should split the ANF: the hardware performs the simple test or rewrite, and software keeps the control state.

6.2.3 Heterogeneous Deployment and Cost-Driven Placement

These targets are complementary. A cloud deployment may contain hosts with eBPF datapaths, servers with SmartNICs, top-of-rack switches with P4 pipelines, and normal user-space proxies. No one target is best for every ANF. A switch may be the best place for a simple routing rule. A SmartNIC may be a good place for a small counter or rewrite. eBPF may be better for logic that needs maps or timers. A proxy or RPC library may still be the right place for code that needs large state, complex parsing, or application runtime support.

The compiler's job is to choose among these options. For each element, it should first ask which targets can run it at all: whether the target can read the needed fields, hold the needed state, finish within its compute budget, and satisfy the developer's trust and update constraints. Among the valid targets, the optimizer can then use the cost model to choose the cheapest placement. The cost must include not only the work done by the element, but also the cost of moving between targets. For example, placing one element on a switch and the next in a proxy may save work in the switch but still require the RPC to enter the host later.

This is the same kind of placement problem as in Chapter 4, but with more targets and stricter target rules. ADN keeps it manageable because the specification is already split into small elements with known field use, state use, and side effects. The open work is to build the target checks and cost profiles, then let the existing optimizer choose a mixed deployment instead of asking the operator to hand-place every ANF.

6.3 Ingress, Egress, and Cross-Organization Calls

This dissertation assumes one administrative domain. Inside that boundary, the compiler can know the RPC schemas, update endpoints and proxies together, and decide which fields are public to the application network. Many real deployments also have traffic that crosses this boundary: public APIs, partner calls, inter-cloud calls, and calls to third-party services. Those peers should not be expected to run the same generated stack or trust the same compiler.

The practical answer is a gateway [13]. Ingress traffic can arrive over standard protocols such as gRPC over HTTP/2 and be translated at the edge into the compiled ADN stack. Egress traffic can make the reverse transition before leaving the organization. This is similar to today's Kubernetes Ingress Controllers and service mesh gateways, but the gateway would be generated from the same specification as the internal

stack. It would know which fields are needed for policy, which fields remain private, and which delivery semantics must be preserved across the boundary.

Gateways raise their own research questions. Translation can reintroduce parsing and buffering cost at the boundary, even if the internal path stays flat. The compiler must decide where that cost is acceptable and which policies should run before or after translation. The rollout protocol also becomes harder, because only one side of the boundary may upgrade at a time. Finally, cross-organization calls need a different trust model: one organization may expose only a subset of its schema, hide some policies, or require a contract that both sides can audit. Future work should treat gateways as first-class compiler outputs rather than hand-written compatibility code.

6.4 What the Framework Does Not Support

The directions above describe what the framework could support. It is also important to state what it does not support.

The single-administrative-domain assumption. The dissertation assumes a single administrative domain. This assumption is central. It allows the system to upgrade proxies and endpoints together. It allows the compiler to know the full set of RPC schemas and policies, as discussed in Chapter 3. It also supports the trust assumptions behind the public/private wire-format split. Section 6.3 describes gateways as a partial answer for boundary traffic, but a full cross-organization compiler is a different research problem.

Workloads with unstable schemas. The wire-format synthesis assumes a stable RPC schema, similar to Protobuf. Applications with highly dynamic schemas, such as ad-hoc analytics queries, schemaless data exchange, or data-lake workloads, do not fit the current framework. Some support may be possible through runtime field discovery or opt-out dynamic fields, but the current design targets schema-stable microservice deployments.

Workloads not dominated by communication. The main benefit of ADN is lower communication cost. Applications dominated by computation, such as inference serving, video transcoding, or CPU-bound batch analytics, will see smaller end-to-end gains because communication is a smaller part of total cost. The

framework still applies, but the expected benefit is smaller. The evaluation in this dissertation focuses on communication-heavy services such as online services, search, recommendation, and real-time analytics.

Stronger adversaries. The threat model matches common service mesh assumptions: the operator trusts the substrate, the application trusts the operator, and the network adversary is passive. Substrate compromise is out of scope. This includes a malicious sidecar, compromised SmartNIC firmware, or a compromised kernel. Defending against these attacks requires a different design, such as attestation or proof-carrying datapaths.

6.5 Summary

The main thesis is that the RPC communication stack should be compiled, not configured, within a single administrative domain. This chapter describes the next steps for that compiler. It needs a cost model that remains useful under contention, backends for kernel and hardware datapaths, and generated gateways for traffic that crosses the organizational boundary. It also needs clear limits: the current design assumes stable schemas, communication-heavy workloads, and trusted substrates. These limits do not change the core result, but they mark where future systems work should focus.

Chapter 7

Conclusion

This dissertation began from a simple observation: in modern microservice applications, communication is no longer a background detail. Every typed RPC passes through serialization, delivery, security, and policy code, and that cost is paid on every edge of a large service graph. Today's layered, proxy-centric stack turns this cost into a substantial RPC tax. The central claim of the dissertation is that this tax is not inevitable. Within a single administrative domain, where schemas, policies, trust boundaries, and deployment targets are known together, the RPC communication stack can be specified once at a high level and compiled into an application-specific implementation.

The chapters developed that claim in four steps. MeshInsight first made the tax visible, replacing black-box service mesh benchmarking with a compositional model that attributes overhead to protocol parsing, IPC, kernel crossings, and ANF execution. The ADN chapter then drew the architectural conclusion: the right abstraction boundary is not a fixed sequence of runtime layers, but a semantic specification from which the implementation can be synthesized. AppNet realized this idea for application network functions, showing that stateful policies can be written as analyzable elements and optimized under symbolic equivalence. fRPC then extended the same principle across serialization, delivery, encryption, and proxy execution, producing a flat stack that preserves today's application-network semantics while avoiding full protocol termination and buffering on the common path.

Together, these systems show that application networks can be treated as compiler outputs rather than infrastructure configured around an application after the fact. The point is not only lower latency or lower

CPU use, though the evaluations show both. The larger point is that once communication behavior is expressed as a high-level program, the compiler can reason about placement, state, security boundaries, delivery semantics, and cost together, instead of forcing developers and operators to encode those decisions indirectly through layers, headers, proxies, and hand-written ANFs.

Bibliography

[1] 2018. Embracing eventual consistency in SoA networking. https://blog.envoyproxy.io/embracing-eventual-consistency-in-soa-networking-32a5ee5d443d.

[2] 2022. Cilium service mesh — everything you need to know. https://isovalent.com/blog/post/cilium-service-mesh/.

[3] [n.d.]. Apache Dubbo: A high-performance, Java-based open-source RPC framework. https://dubbo.apache.org/. Accessed: 2026-06.

[4] [n.d.]. Apache Thrift. https://thrift.apache.org/. Accessed: 2026-06.

[5] [n.d.]. Cap'n proto. https://capnproto.org/. Accessed: 2026-06.

[6] [n.d.]. Cilium service mesh. https://cilium.io/use-cases/service-mesh/. Accessed: 2026-06.

[7] [n.d.]. Demonstrations of funclatency, the Linux eBPF/BCC version. https://github.com/iovisor/bcc/blob/master/tools/funclatency_example.txt. Accessed: 2026-06.

[8] [n.d.](). Envoy: Application logging. https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/application_logging. Accessed: 2026-06.

[9] [n.d.](). Envoy: Bandwidth limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/bandwidth_limiting. Accessed: 2026-06.

[10] [n.d.](). Envoy: Cache filter. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/cache_filter. Accessed: 2026-06.

[11] [n.d.](). Envoy: Global rate limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/global_rate_limiting. Accessed: 2026-06.

[12] [n.d.](). Envoy: Load balancing. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing. Accessed: 2026-06.

[13] [n.d.](). Envoy proxy. https://www.envoyproxy.io/. Accessed: 2026-06.

[14] [n.d.](). Envoy: Rate limit. https://github.com/envoyproxy/ratelimit. Accessed: 2026-06.

[15] [n.d.](). Envoy statistics. https://istio.io/latest/docs/ops/configuration/telemetry/envoy-stats/. Accessed: 2026-06.

[16] [n.d.]. FlatBuffers: Memory-efficient serialization library. https://flatbuffers.dev/. Accessed: 2026-06.

[17] [n.d.](). Go gRPC middleware. https://github.com/grpc-ecosystem/go-grpc-middleware. Accessed: 2026-06.

[18] [n.d.](). gRPC: A high performance, open source universal RPC framework. https://grpc.io. Accessed: 2026-06.

[19] [n.d.](). gRPC encryption. https://github.com/grpc/grpc-go/blob/master/examples/features/encryption/README.md. Accessed: 2026-06.

[20] [n.d.](). gRPC interceptors. https://grpc.io/docs/guides/interceptors/. Accessed: 2026-06.

[21] [n.d.](). gRPC-tools. https://github.com/bradleyjkemp/grpc-tools. Accessed: 2026-06.

[22] [n.d.]. HAProxy: The reliable, high performance TCP/HTTP load balancer. http://www.haproxy.org/. Accessed: 2026-06.

[23] [n.d.](). HTTP filters in Envoy. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters. Accessed: 2026-06.

[24] [n.d.]. Ingress controllers. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/. Accessed: 2026-06.

[25] [n.d.](). Ingress gateways. https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/. Accessed: 2026-06.

[26] [n.d.](). Istio: Circuit breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/. Accessed: 2026-06.

[27] [n.d.](). Istio data plane modes. https://istio.io/latest/docs/overview/dataplane-modes/. Accessed: 2026-06.

[28] [n.d.](). Istio: Fault injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/. Accessed: 2026-06.

[29] [n.d.](). The Istio service mesh. https://istio.io/. Accessed: 2026-06.

[30] [n.d.]. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. Accessed: 2026-06.

[31] [n.d.]. JSON web tokens. https://jwt.io/introduction. Accessed: 2026-06.

[32] [n.d.]. Kubernetes. https://kubernetes.io/. Accessed: 2026-06.

[33] [n.d.](). Linkerd features. https://linkerd.io/2.14/features/. Accessed: 2026-06.

[34] [n.d.]. Linux perf examples. https://www.brendangregg.com/perf.html. Accessed: 2026-06.

[35] [n.d.]. Lua filter in Envoy. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/lua_filter. Accessed: 2026-06.

[36] [n.d.](). MSG_ZEROCOPY. https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html. Accessed: 2026-06.

[37] [n.d.]. NGINX: Advanced load balancer, web server, and reverse proxy. https://www.nginx.com/. Accessed: 2026-06.

[38] [n.d.]. OWASP Coraza web application firewall. https://coraza.io/. Accessed: 2026-06.

[39] [n.d.](). Protocol buffers. https://protobuf.dev/. Accessed: 2026-06.

[40] [n.d.]. quic-go: A QUIC implementation in pure Go. https://github.com/quic-go/quic-go. Accessed: 2026-06.

[41] [n.d.]. Redis. https://redis.io/. Accessed: 2026-06.

[42] [n.d.]. Unix domain socket. https://en.wikipedia.org/wiki/Unix_domain_socket. Accessed: 2026-06.

[43] [n.d.](). WebAssembly in Envoy. https://github.com/proxy-wasm/spec/blob/master/docs/WebAssembly-in-Envoy.md. Accessed: 2026-06.

[44] [n.d.](). The world's lightest, fastest service mesh. https://linkerd.io/. Accessed: 2026-06.

[45] [n.d.](). wrk: a HTTP benchmarking tool. https://github.com/wg/wrk. Accessed: 2026-06.

[46] [n.d.](). wrk2: a constant-throughput, correct-latency HTTP benchmarking tool. https://github.com/giltene/wrk2. Accessed: 2026-06.

[47] [n.d.]. Zero trust networking with Istio. https://istio.io/latest/docs/concepts/security/. Accessed: 2026-06.

[48] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine

learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283.

[49] Amazon Web Services. [n.d.]. Amazon virtual private cloud (Amazon VPC). `https://aws.amazon.com/vpc/`. Accessed: 2026-06.

[50] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126.

[51] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Kim, Aaron Taylor, Hugh Mageshwaran, Manas Murthy, Tom Vincent, et al. 2020. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 753–768.

[52] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media.

[53] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[54] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "micro" back in microservice. In *USENIX Annual Technical Conference (ATC)*, pages 645–650.

[55] Ajay Brahmakshatriya, Chris Rinard, Manya Ghobadi, and Saman Amarasinghe. 2024. NetBlocks: Staging layouts for high-performance custom host network stacks. In *Proceedings of the ACM on Programming Languages (PLDI)*, volume 8, pages 467–491.

[56] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A software-defined framework for developing, deploying, and managing network functions. In *ACM SIGCOMM Conference*, pages 511–524.

[57] Buoyant. 2022. eBPF, sidecars, and the future of the service mesh. https://buoyant.io/blog/ebpf-sidecars-and-the-future-of-the-service-mesh.

[58] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible kernel transport with eBPF. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[59] Isabelle Chrisment, Delphine Kaplan, and Christophe Diot. 1998. An ALF communication architecture: Design and automated implementation. *IEEE Journal on Selected Areas in Communications*, 16(3):332–344.

[60] David D. Clark and David L. Tennenhouse. 1990. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208.

[61] Cloud Native Computing Foundation. 2022. Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.

[62] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 56–62.

[63] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220.

[64] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, pages 1–14.

[65] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–150.

[66] Envoy Project. 2022. How fast is Envoy? https://www.envoyproxy.io/docs/envoy/latest/faq/performance/how_fast_is_envoy.

[67] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr, and Dejan Kostić. 2020. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *USENIX Annual Technical Conference (ATC)*, pages 673–689.

[68] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291.

[69] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18.

[70] 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 *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 19–33.

[71] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *Proceedings of the ACM SIGCOMM Conference*, pages 435–450.

[72] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174.

[73] Google Cloud Platform. [n.d.]. Online boutique — a cloud-native microservices demo appli-

cation. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed: 2026-06.

[74] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 861–877.

[75] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, et al. 2012. XIA: Efficient support for evolvable internetworking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322.

[76] HashiCorp. [n.d.]. Consul service mesh. https://www.consul.io/docs/connect. Accessed: 2026-06.

[77] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the pain. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 535–549.

[78] Patricia K. Immich, Ravi S. Bhagavatula, and Ravi Pendse. 2003. Performance analysis of five interprocess communication mechanisms across UNIX operating systems. *Journal of Systems and Software*, 68(1):27–43.

[79] InfoQ. 2022. eBPF and the service mesh: Don't dismiss the sidecar yet. https://www.infoq.com/articles/ebpf-service-mesh/.

[80] Istio. 2022. Introducing ambient mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/.

[81] Istio. [n.d.]. Istio: Performance and scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/. Accessed: 2026-06.

[82] IstioCon. 2022. Building a simplified service mesh API for developers. https://events.istio.io/istiocon-2022/sessions/building-simplified-service-mesh-api-for-developers/.

[83] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 97–112.

[84] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. 2018. Service fabric: a distributed platform for building microservices in the cloud. *Proceedings of the Thirteenth EuroSys Conference*.

[85] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference*, pages 1–16.

[86] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–186.

[87] Kinvolk. 2019. Performance benchmark analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/.

[88] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297.

[89] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A data-oriented (and beyond) network architecture. In *Proceedings of the ACM SIGCOMM Conference*, pages 181–192.

[90] Steven Landow. 2021. gRPC proxyless service mesh. https://istio.io/latest/blog/2021/proxyless-grpc/.

[91] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14.

[92] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking microservice observability with eBPF. In *IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8.

[93] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the ACM SIGCOMM Conference*, pages 1–14.

[94] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14.

[95] Jochen Liedtke. 1993. Improving IPC by kernel design. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188.

[96] Linkerd. 2021. Benchmarking Linkerd and Istio: 2021 Redux. `https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/`.

[97] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *USENIX Annual Technical Conference (ATC)*, pages 363–378.

[98] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 412–426.

[99] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM Conference*, pages 175–186.

[100] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413.

[101] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the ACM SIGCOMM Conference*, pages 15–28.

[102] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *Proceedings of the ACM SIGCOMM Conference*, pages 199–212.

[103] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the ACM SIGCOMM Conference*, pages 327–341.

[104] Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 30–36.

[105] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16.

[106] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. FlowBlaze: Stateful packet processing in hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 531–548.

[107] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM Conference*, pages 780–794.

[108] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 805–825.

[109] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System support for elastic execution in virtual middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 227–240.

[110] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular SDN programming with Pyretic. Technical report, USENIX.

[111] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334.

[112] Eric Rescorla. 2018. The transport layer security (tls) protocol version 1.3. RFC 8446.

[113] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288.

[114] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 969–985.

[115] 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 498–514.

[116] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and implementation of a consolidated middlebox architecture. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336.

[117] Service Mesh Performance. [n.d.]. Cloud native value measurement (MeshMark). https://smp-spec.io/meshmark. Accessed: 2026-06.

[118] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400.

[119] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, et al. 2024. Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture. In *ACM SIGCOMM Conference*, pages 860–875.

[120] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-tuned threading for OLDI microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–194.

[121] Christopher Stewart and Kai Shen. 2005. Performance modeling and system management for multi-component online services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 71–84.

[122] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 65–76.

[123] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling network function parallelism in NFV. In *ACM SIGCOMM Conference*, pages 43–56.

[124] Aditya Venkataraman and Kishore Kumar Jagadeesha. 2015. Evaluation of inter-process communication mechanisms. *Architecture*, 86:64.

[125] Wlamir Olivares Loesch Vianna, Leonardo Ramos Rodrigues, Takashi Yoneyama, and David Issa Mattos. 2016. Troubleshooting optimization using multi-start simulated annealing. In *Annual IEEE Systems Conference (SysCon)*, pages 1–6.

[126] VMware. [n.d.]. What is network virtualization? https://www.vmware.com/topics/glossary/content/network-virtualization.html. Accessed: 2026-06.

[127] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review*, 43(4):87–98.

[128] W3C. 2021. W3C trace context. https://www.w3.org/TR/trace-context/.

[129] Eric Wagner, David Heye, Martin Serror, Ike Kunze, Klaus Wehrle, and Martin Henze. 2024. MADTLS: Fine-grained middlebox-aware end-to-end security for industrial communication. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*.

[130] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic scaling of stateful network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 299–312.

[131] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. 2024. Load is not what you should balance: Introducing Prequal. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1285–1299.

[132] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. 2023. ClickINC: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM Conference*, pages 798–815.

[133] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. 2022. SwiSh: Distributed shared state abstractions for programmable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–191.

[134] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems*, 35(4):1–37.

[135] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the ACM SIGCOMM Conference*, pages 283–295.

[136] 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 Annual Technical Conference (ATC)*, pages 655–672.

[137] Bohan Zhao, Wenfei Wu, and Wei Xu. 2023. NetRPC: Enabling in-network computation in remote procedure calls. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 199–217.

[138] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. 2023. Application defined networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets)*, pages 87–94.

[139] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting overheads of service mesh sidecars. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[140] Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, Nikola Bojanic, Jingrong Chen, Gilbert Bernstein, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. 2025. High-level programming for application networks. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[141] Xiangfeng Zhu, Yang Zhou, Yuyao Wang, Xiangyu Gao, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. 2025. Rethinking RPC communication for microservices-based applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.