High-level Programming for Application Networks

Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, and Nikola Bojanic, *University of Washington;* Jingrong Chen, *Duke University;* Gilbert Louis Bernstein and Arvind Krishnamurthy, *University of Washington;* Sam Kumar, *University of Washington and UCLA;* Ratul Mahajan, *University of Washington;* Danyang Zhuo, *Duke University*

https://www.usenix.org/conference/nsdi25/presentation/zhu

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

High-level Programming for Application Networks

Xiangfeng Zhu[w] Yuyao Wang[w] Banruo Liu[w]* Yongtong Wu [w] † Nikola Bojanic[w]
Jingrong Chen[d] Gilbert Louis Bernstein[w] Arvind Krishnamurthy[w] Sam Kumar[w][l]
Ratul Mahajan[w] Danyang Zhuo[d]

[w] *University of Washington* [d] *Duke University* [l] *UCLA*

Abstract

Application networks facilitate RPC communication between the microservices of cloud applications. They are built today using service meshes which employ low-level specifications and have high overhead—it is difficult to express even simple application-specific functionality (e.g., access control based on RPC fields) and RPC latency can more than double. We develop AppNet, a framework for building expressive and high-performance application networks. Developers specify rich application network functions (ANFs) in a high-level language with generalized match-action rules. We compile the specifications to high-performance code after optimizing *where* (e.g., client, server) and *how* (e.g., RPC library, proxy) each function runs. The optimization uses symbolic abstraction and execution to judge if different runtime configurations of possibly stateful functions are semantically equivalent for arbitrary RPC streams. Our experiments show that AppNet can express common ANFs in only 7-28 lines of code. Its optimizations lower RPC processing latency by up to 82%.

1 Introduction

Modern cloud applications are composed of a collection of microservices instead of monolithic binaries [10,50,56]. This architecture speeds up software development because each microservice can use the language and libraries best suited to its task. It also improves scaling because each service can be scaled (via replicas) independently.

However, when applications are disaggregated, many simple function calls become remote procedure calls (RPCs) over the network. To secure and manage this communication, developers build *application networks*, also known as service meshes, using chains of application network functions (ANFs) that implement authentication, fault tolerance, logging, load balancing, etc. Enabled by RPC traffic processing platforms (e.g., Envoy proxy [19], RPC library plugins [18]) and by controllers such as Istio [39], application networks are ubiquitous today. Nearly all cloud application developers use them [7].

In theory, it should be easy to build performant application networks that are specialized to application needs. After all, an application network is meant to support known application(s) and functionality, in contrast to the Internet which is meant to support arbitrary applications. But the current reality is that implementing even simple application-specific functionality, such as access control or routing based on RPC fields, is challenging [1,70,76], and application networks can increase RPC latency as well as CPU usage by 2-7x [3,5,6,66,76,86].

These shortcomings stem from the low-level programming model of current application networks. Developers can either write custom ANFs in general-purpose languages such as C++ and Rust or use generic HTTP modules. Implementing application-specific functionality is difficult either way. Custom ANFs are difficult to write because high-level abstractions do not exist, and a separate ANF must be written for each RPC processing platform. That is why HTTP modules, used after wrapping RPC traffic in HTTP, are more popular. But using them requires modifying the microservice code to expose relevant RPC information as HTTP headers [25,70,76]. Such modifications are onerous, especially when the microservice code is authored by third parties (a common case).

Low-level programming of application networks also hurts performance. There are different locations, including the client, the server, or another remote host [13], and different platforms where ANFs can be run. Different options offer different trade-offs [70,76]—e.g., RPC libraries have low overhead but cannot be used with untrusted microservice code; and rate limiting total requests to a microservice is better done at the server-side to eliminate the coordination overhead among clients. Today, developers must explicitly configure the order, location, and platform for each ANF. However, it is difficult to manually determine high-performance configurations because performance depends on the interactions between various ANFs and on the (dynamically varying) number of client and server replicas. Since the semantics of ANFs are unknown to current network controllers, they must execute exactly what the user provides, even if semantically-equivalent configurations with significantly better performance exist.

*Now with UIUC.
†Now with Peking University.

Our goal is to enable expressive, easy-to-build, and high-performance application networks. We explore an approach based on high-level programming, inspired by works that achieve a similar goal for layer-3/4 networks [43,44,65,68,80] and switch forwarding [46,47,54,74]. We develop AppNet, a framework for building application networks which has (1) a high-level language to specify ANFs, (2) a compiler that generates an optimized configuration across multiple platforms, and (3) a controller that instantiates the configuration.

With AppNet, developers specify network functionality as a chain of *elements*, where each element expresses some type of RPC processing (e.g., load balancing). Our language offers a generalized match-action primitive where matches can be performed on RPC fields, state, or results of built-in functions (e.g., *random*) and actions can be reading or writing RPC fields and state. This primitive is expressive enough for RPC processing (unlike others we tried, such as SQL dataflow [9]) and yet can be compiled to high-performance code for multiple platforms. AppNet also provides primitives for managing shared state between multiple instances of the same element and shared state with the control plane. These primitives make it trivial to express functionality such as global rate limiting that may require coordination across element instances.

Based on element specifications, the AppNet compiler outputs runnable modules for various platforms. The mapping between elements and runnable modules is not one-to-one and is optimized for performance. A module may contain multiple elements to reduce overhead, the element order may be altered (e.g., moving elements that drop RPCs upstream reduces work for downstream elements), elements may be migrated based on state synchronization costs, or elements may be co-located in selected locations to completely bypass the (expensive) invocation of some platforms.

It is critical that the output of the compiler be semantically equivalent to the user's specification. Determining equivalence is challenging because elements are stateful (thus, processing an RPC impacts subsequent RPCs) and RPCs can be dropped or reordered (which can impact downstream state). AppNet uses a form of symbolic abstraction and symbolic execution [59] to determine when two chains of elements, each a possible runtime configuration, are equivalent for arbitrary streams of RPCs. It abstracts each element as symbolic transfer functions that capture its side effects (e.g., RPC field modifications and state updates). It then uses symbolic execution over these functions to compute the processing semantics of a chain, including changes to the element state and auxiliary outputs such as logs. The outputs of this execution help determine whether two chains are equivalent.

As part of its compilation, AppNet allows users to opt for weak consistency and improve performance. It can weaken consistency in two ways: *state consistency*, which controls the consistency of shared state among element instances, and *observation consistency*, which controls whether auxiliary outputs such as logs record the same information. Weak consistency may not always be appropriate, but applications that can tolerate them can experience higher performance. Such modulation of network behavior based on application characteristics is a unique opportunity for application networks.

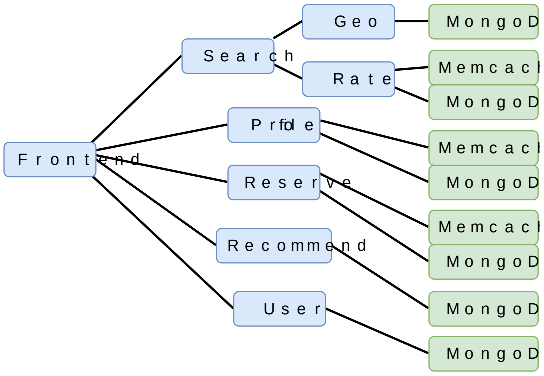


Figure 1: Hotel Reservation application's microservices [53].

Our current implementation of AppNet [16] supports three popular RPC processing platforms: EnvoyNative [31], Envoy-Wasm [40] (respectively, C++ and WebAssembly-based execution environments in Envoy), and gRPC interceptors [29]. The Envoy platforms support deployment as a proxy located with the client or server ("sidecar") or as a remote proxy.

We evaluate AppNet by implementing 14 ANFs and studying a range of chain specifications. We find that 12 common ANFs need only 7–28 lines of code, and 2 more complex ANFs which integrate load balancing and routing—Meta's ServiceRouter [70] and Google's Prequal [82]—need only 62 and 88 lines. AppNet's compiler optimizations lower RPC processing latency by up to 82% and CPU usage by up to 75%, compared to unoptimized chain configurations.

Our results show that it is possible to build application networks that are both expressive and performant. Decoupling network specification from the running implementation opens up new capabilities too. One such capability, which AppNet already supports, is consistently upgrading the network when specifications change or microservices scale. The upgrade ensures that no RPC ever experiences a mix of old and new network policies, which can lead to performance anomalies and policy violations [4,69]. We are exploring others, including offloading RPC processing to kernel or hardware and serializing RPCs based on fields accessed in the network.

2 Background

Microservices are the dominant paradigm for modern cloud applications. Figure 1 shows a typical microservices-based application. Intended to enable hotel reservations, its functionality is split across eight types of microservices. Clients contact (a replica of) the Frontend service, which then communicates with a middle-tier service depending on the query. The middle-tier services handle tasks such as searching for hotels, completing hotel bookings, updating user profiles, and provid-

Function	Description
Fault Injection [23]	Randomly drop an RPC
Cache [22]	Generic cache for RPC responses
Rate Limiting [24]	Control the rate of RPC requests
Load Balancing [26]	Load balance across service replicas
Logging [20]	Record the RPC content to disk
Mutation [30]	Modify the RPC content
Application Firewall [15]	Rejects an RPC based on pattern matching
Metrics [28]	Monitor the RPC latency and success rate
Admission Control [45]	Drop RPC requests based on success rates
Encryption [12]	Encrypt certain RPC fields
Bandwidth Limit [21]	Control the amount of data to the service
Circuit Breaking [32]	Limit the impact of undesirable network peculiarities

Table 1: Common ANFs for microservices.

ing recommendations. They may contact other microservices and storage infrastructure. Production settings can have anywhere from 1 to 100s of replicas of each microservice [10,56], communicating via remote procedure calls (RPCs).

To manage communication between microservices, developers build application networks with rich functionality. Table 1 lists 12 common ANFs which we identified by surveying popular platforms to build application networks [11, 14, 30, 31, 39].

As Figure 2 shows, ANFs can be run in different locations and using different platforms. Common platforms include: (1) RPC library [18, 70], which is embedded directly into the client or server microservice's code to manage communication without external proxies, (2) sidecar [19, 35], a proxy co-located with the client and server that intercepts all traffic, and (3) remote proxy [13, 76], a standalone proxy that handles traffic between services. Simple RPC processing may be located in the kernel (using eBPF) as well [17]. We will extend our work to this platform in the future.

ANFs are implemented in a platform-specific manner, and developers typically invoke them via *service mesh* controllers [35, 39]. The controllers configure the target platform and also configure RPC traffic interception and redirection. Regardless of whether ANFs run inside an RPC library, a proxy, or both, developers must manually decide their execution order, platform, and placement.

This decision is complex and significantly impacts performance. As an example, consider the communication between two replicas of Search and one of Geo (Figure 3). The developer wants to run Logging, Load Balancing, Rate Limiting, and Application Firewall ANFs from Table 1 between the microservices. Rate Limiting uses a stateful token bucket to control the RPC request rate to a Geo replica. Figure 3a shows a possible configuration provided by the developer.

However, factoring in the properties of the platforms and ANFs' semantics, configurations with better performance exist. We could move Rate Limiting to the server side, which eliminates the need for synchronizing state between multiple ANF instances because the state depends on server replica, and we could move Logging to the RPC library because that platform has lower overhead. Figure 3b shows an optimized configuration for the chain.

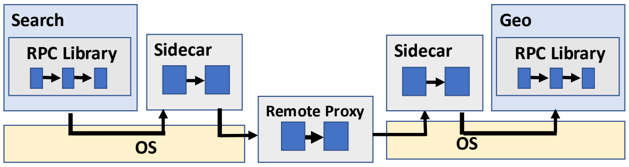
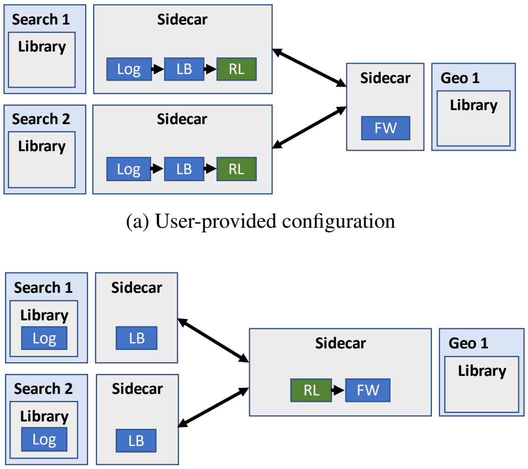


Figure 2: RPC processing between Search (client) and Geo microservices. Dark blue rectangles are ANFs.



(a) User-provided configuration

(b) Optimized, semantic-preserving configuration

Figure 3: Two possible configurations for an application network. 'Log' is Logging, 'LB' is Load Balancing, 'FW' is Application Firewall, and 'RL' is Rate Limiting (stateful).

Not all performance-enhancing configurations preserve the semantics of the input chain. Identifying optimized configurations that preserve semantics requires careful reasoning about semantics and state dependencies. For example, reordering Application Firewall and Rate Limiting and moving the Application Firewall to the client side can improve performance by preventing RPCs that would eventually be dropped from traversing the network. Yet, this reordering would violate the input semantics, as Rate Limiting's internal state could be affected depending on whether Application Firewall drops a packet before or after it.

The current architecture of application networks—with low-level, platform-specific ANFs whose ordering, location, and platform are manually determined—has evolved organically, and it has the limitations mentioned earlier: difficult-to-express application-specific functionality and poor performance. We ask what a principled design for a framework to build application networks might look like and if it can effectively address these limitations.

3 AppNet Overview

AppNet provides high-level programming abstractions for application networks. The abstractions simplify the specification of desired network functionality and enable optimizations that help realize the functionality with low overhead. This section overviews the abstractions, the optimizing compiler, and the runtime controller.

3.1 Programming abstractions

Like high-level languages for layer-3/4 networks [43], the AppNet language focuses on the data plane, which has complex, performance-sensitive processing. Some ANFs have a control plane as well. For load balancing, the data plane forwards the RPCs to different replicas, and the control plane maintains the list of active replicas. AppNet provides shared state abstractions that enable the control plane to communicate easily with the data plane, but it does not provide high-level abstractions for control plane logic.

With AppNet, users specify network functionality between a pair of microservices using chains of *elements*, where each element encodes an ANF. Key aspects of the specification language are:

Generalized match-action rules RPC processing is specified as match-action rules that operate over RPC fields (not arbitrary low-level code for standard protocols), state, and built-in functions. RPC fields (both metadata and payload) are represented as key-value pairs, which the elements can easily read/write without worrying about (de)serialization. The state is also represented as key-value pairs.

Match-action rules are natural for RPC processing, and they give the compiler visibility into which RPC fields are accessed and what the state depends on. At first glance, the match-action paradigm may seem limiting for layer-7 processing, but we have found it to be rich enough to encode a wide range of ANFs. This encoding is made possible by allowing richer match expressions and actions than switch languages like P4 [46]. Match expressions go beyond static table lookups and can use a suite of operators and built-in functions, and actions can read/write state multiple times.

Shared-state management Shared state, between multiple instances of the same ANF (e.g., for global rate limiting) or between control and data planes, is common in application networks. Developers manage this state manually today, which is onerous. AppNet abstracts away this state management and provides APIs to read-write state. This abstraction lets us reduce synchronization costs by placing elements and state based on what the state depends on. If the state depends on server replica, we prefer placing the element on the server side so there will be only one reader-writer to this shard of the state; and if we can do that, we can locate the shard on the server as well. Shared state management also lets us trade off consistency and performance, as we discuss next.

Performance-consistency tradeoff AppNet lets users trade consistency for performance in two ways. First, they can opt for weak (eventual) consistency for shared state. This trade-off is well-known for distributed systems [49,85]. Users can also specify weak observation consistency. Two network configurations have strong observation consistency when all outputs—the microservices themselves and auxiliary outputs like logs—get the same content in both configurations. Weak observation consistency permits the auxiliary output channels, but not microservices, to get different content, e.g., logs will differ when logging occurs before versus after an ANF that modifies the RPC's *geo-region* field. Applications that can tolerate weak observation consistency can get higher performance because it creates more optimization opportunities.

Location & platform agnostic elements, paired elements AppNet does not require users to specify an element's location or platform explicitly (though they can pin if they want). This under-specification enables the compiler to determine the location and platform based on performance. Thus, while AppNet uses element chains as a specification primitive like application networks today, a key difference is that AppNet chains specify processing semantics, not execution order.

AppNet also lets users define pairs of elements that must be used in concert (e.g., compression/decompression). For paired elements, AppNet ensures that one element is allocated to the client side and the other to the server side, and conflicting elements are not placed between them. This primitive simplifies specification and prevents errors such as forgetting the decompression element or placing compression between encryption and decryption [1].

3.2 Compiler

The AppNet compiler takes in the network specifications and RPC definitions (e.g., Protobuf [37]) used by the application code. Using RPC definitions during compilation frees the developer from having to write RPC (de-)serialization code, helps validate the specifications (e.g., RPC field names), and ensures that the generated code is up-to-date with RPC definitions. After parsing and sanity-checking the specifications, the compiler tags each state variable with what it depends on (e.g., client-replica, server-replica, global, etc.) based on what the state is indexed on. This information is used for optimizing element and state placement.

The compiler outputs modules that can run on one or more platforms (e.g., RPC library, sidecar). The heart of this process is computing a performant runtime configuration that is semantically equivalent to the user input. In the output configuration, elements may be reordered and combined into one module. The challenge in this computation is that elements can be stateful, which means that we must analyze equivalence for arbitrary streams of RPCs, not individual RPCs;

state updates caused by an RPC impact all subsequent RPCs. The analysis must also account for RPCs getting dropped or reordered—a stateful element that is upstream of a rate limiter, which drops RPCs, will see a different RPC stream than when it is moved downstream, leading to different state updates. Statefulness and the possibility of dropped/reordered streams make it hard for us to reuse equivalence analysis developed in domains such as stateless packet forwarding [43] or instruction reordering [77].

AppNet solves this challenge by first symbolically abstracting elements. The abstraction captures all of an element's side-effects and dependencies using transfer functions, without capturing the exact processing logic. For the rate limiter, it will capture that it may drop an RPC, that its internal state is updated when an RPC appears, and that this update does not depend on the RPC content. AppNet then uses symbolic execution over chains of element-level transfer functions to determine end-to-end transfer functions of a chain, including output RPCs, state updates, and what is sent to auxiliary outputs such as logs. Two chains are deemed equivalent if their end-to-end transfer functions are equivalent.

3.3 Controller

The AppNet controller maintains an up-to-date, global view of all service replica locations by integrating with cluster managers such as Kubernetes [33]. It orchestrates the deployment of runnable modules generated by the compiler, ensuring they are deployed to the appropriate replicas and platforms. It also ensures that configuration updates are consistent.

4 AppNet design

We now describe the AppNet language and compiler in detail. Appendix A describes code generation and consistent updates.

4.1 Specification language

Figure 4 shows the core of the AppNet grammar, without type annotations and syntactic features that enhance usability. Figure 5 show an example specification. At the highest level, RPC processing between two microservices is described using four sub-chains of elements: two for those that must run on the client or server sides, one for those that can run anywhere, and one for paired elements (e.g., encrypt/decrypt). For paired elements, the first element is run on the client side and the second on the server side. It is not mandatory to have an element in any of these sub-chains. Users can include all elements in the "any" sub-chain. The implied processing order for elements is client, pair(client), any, pair(server), server; the compiler may re-order the elements, but it will preserve this semantics. Users can optionally specify weak observation consistency. Figure 5a shows the AppNet chain specification for the example in Figure 3, where only the "client" and "any" chain are specified.

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

Figure 4: AppNet specification language.

An AppNet element specification has four sections. The `state` section declares variables that hold local or shared state among all running instances of the element (serving different microservice replicas). Shared state declarations optionally accompany desired consistency and aggregation methods. By default, all reads and writes to shared state are synchronous. If users specify weak consistency, AppNet maintains a local copy and syncs this copy in the background (thus providing eventual consistency). During the sync, the default semantics is last-writer-wins, but users can specify custom aggregations (akin to CRDTs [73]). Sum (useful for counters) is a supported aggregator currently, and we will add more as needed.

The `init` section initializes the state variables. Initialization may use user input. The `req` and `resp` sections have match-action-based processing for RPC requests and responses. AppNet invokes these methods with a map that represents the content of the RPC payload and metadata (which includes RPC sender, destination, and method type). Users can match on any valid expression (Exp), and they can specify multiple actions for each case. Actions either update state or send a message. The message content can be the RPC (with its content represented by Var) or an Error. Messages can be sent downstream (`down`) to the next element or, if it is the last element in the chain, to the microservice; they can also be

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

(a) An element chain specification with four elements.

```
 1 state:
 2     bucket shared
 3
 4 init():
 5
 6 req(rpc):
 7     key = get(rpc, 'dst')
 8     has_token = get(bucket, key, lambda(token):
 9         match token > 0:
10             true =>
11                 token = token - 1
12                 return true
13             false =>
14                 return false
15     )
16
17     match has_token:
18         true =>
19             send(rpc, down)
20         false =>
21             send(err('rate limited'), up)
22
23 resp(rpc):
24     send(rpc, up)
```

(b) The element specification for rate limiting.

Figure 5: Example AppNet specifications.

sent upstream (up) to the previous element, which is used for sending back errors or cached responses. AppNet checks at runtime if transmitted content can be cast into the expected RPC. In addition to the up and down channels, users can declare (in state section) and use auxiliary channels (e.g., for logging to disk).

To access state, AppNet provides get and set functions, enabling the compiler to infer state dependencies and determine whether operations should be local or remote. The get function can accept an optional lambda for computations, with updates to shared state being atomic. When a lambda is provided, the original return value of the get function is passed as an argument to the lambda. AppNet also supports various computations on RPC fields and states using built-in functions, including mathematical operators, random number generators, and utilities for encryption and compression.

Figure 5b shows the AppNet specification for an element that limits the rate of RPCs to each server replica using a token bucket. The state section defines a shared state variable called bucket that tracks available tokens. This element assumes that the control plane (not shown) refreshes the token bucket periodically by writing to this shared state. During request processing (Lines 6-21), the element checks if a token is available in the specified key of the bucket (state variables are key-value maps). The third argument of the get function specifies the logic that the data store should execute during the get operation. If a token is available, it is decremented.

4.2 Optimization

The AppNet compiler finds a high-performance configuration (location, platform, and execution order of each element) that respects users' constraints and is semantically equivalent to the input specification. We face a few challenges in computing such a configuration. First, there is a large space of possible configurations. For a chain of n elements with k placement options (location/platform combination) for each, there are $O(n!\cdot n^k)$ possible configurations. Second, we need to know which configurations are semantically equivalent to the user input, which is difficult for reasons mentioned in § 3.2. Third, multiple competing concerns (e.g., network traffic, CPU usage, latency, synchronization cost) make performance estimation difficult. For instance, positioning elements that drop RPCs early in the chain can reduce CPU usage and network traffic, but it can increase synchronization costs by preventing the consolidation of stateful elements.

AppNet addresses these challenges using a multi-start simulated annealing framework because it works well with large, intractable search space. Starting from an initial configuration with all elements randomly mapped to locations and platforms, the algorithm iteratively applies semantic-preserving mutations to the current solution and evaluates the new solution using a cost function. It decides whether to adopt the new solution based on the cost difference and a probability function. Finally, it periodically resets the temperature to avoid getting trapped in local optima.

We describe the cost function below, which is a key determiner of the final output. Appendix E provides more details on mutation strategies, termination conditions, and evaluation results on the optimality and scalability of the framework.

Cost function. Our cost function is a heuristic that combines six preferences. In priority order, they are: (1) Prefer platforms with lower processing costs (e.g., RPC library). (2) Prefer chains where all elements are located on a single platform, as this enables bypassing other platforms (e.g., server-side sidecar), significantly reducing processing overhead. (3) Prefer placements that align with state dependency to avoid synchronization cost. So, elements sharing a state indexed on a client replica should be located on the client side. (4) Prefer co-locating elements that require strong state synchronization (because dependencies are not aligned). Co-location facilitates the consolidation of state synchronization messages, reducing synchronization costs. (5) Same preference, but for elements with weak state consistency. (6) Prefer placing elements that may drop RPCs earlier because it reduces work for downstream elements.

We combine these preferences into a cost function in three steps. First, we compute element-level costs, which are represented as tuples with three components: the estimated fraction of RPCs reaching the element, a binary value indicating whether the element requires strong state synchronization, and a binary value for weak state synchronization. The fraction of RPCs reaching an element is determined by the expected drop rates of upstream elements, set to 0 for elements that do not drop RPCs and a configurable value for those that might drop them. The default value is 0.05, but it can be provided by the developer or informed via runtime telemetry.

Second, we divide the input chains into sub-chains based on location and platform, and compute the cost of each sub-chain by aggregating the element-level cost tuples into a new tuple (with the same three components). The first component of the tuple sums the RPC fractions, reflecting the total processing cost. The second and third components apply a boolean *or*, indicating that synchronization costs are paid once per platform, regardless of how many elements require synchronization.

Finally, we aggregate sub-chain costs into the final chain-level cost. We convert the sub-chain cost tuples into scalar values by treating the binary components as 0/1 and computing a weighted sum of the three values using weights (0.5, 2, 1) for the three dimensions. We then add the baseline platform processing cost of S_p if there are any elements in the sub-chain for platform p. The chain-level cost is the sum of these five sub-chain-level costs. We use $S_p = 2$ for the out-of-process proxy (either sidecar or remote proxy) and $S_p = 0.1$ for the RPC library. We have found our results to be robust to exact values as long as the ordering between parameters is maintained. In the future, we plan to inform these parameters based on runtime profiling of different overheads [86].

4.3 Equivalence checking

There are three steps for deciding if two chains being considered during the optimization are equivalent: (1) symbolically abstracting what each element does; (2) symbolic execution of the chains based on the abstract elements; (3) comparing the outputs of this execution for two chains. We provide an overview of these steps below. Appendix B has the details and a proof of correctness.

Symbolic abstraction of elements This analysis creates an abstract representation of an element that captures the side effects of the element based on its dependencies. The side effects include modifications to the RPC, updates to the internal state, and writes to auxiliary output channels. The representation does not capture the actual processing.

The abstract representation is a transfer function whose parameters include the incoming RPC and the pre-state (prior to RPC processing). We model RPCs using their metadata and payload fields plus two special fields called *drops* and *reorders*. These fields capture whether the RPC could have been dropped or reordered.

Figure 6 (left) shows the transfer functions for three elements inside the green boxes. It assumes that incoming RPCs have two fields *dst* (for destination address) and *user* (for caller's identity) and the special field *drops* (we elide the special field *reorder* for simplicity). Logger logs the incoming RPC to an auxiliary output channel *out*, and so its transfer function appends the incoming RPC to the channel. Load Balancer is a round-robin load balancer that determines the destination using a state variable *idx* that indexes into a list of replicas and is incremented (with wrap around) after each RPC. Load Balancer has two transfer functions. One updates *RPC.dst* based on *idx*—$\mathbf{LB_R}$ is an opaque function that captures this dependency. The other updates *idx* based on an opaque function of *idx* and the *RPC.drops* field because if the RPC were dropped *idx* will not be updated. The transfer function of Firewall updates *RPC.drops* based on *RPC.user*.

We derive these transfer functions using static analysis on the element specification. For each RPC field and state variable modified by the element, a transfer function is generated that replaces the corresponding value with an opaque function representing the internal logic of the element, together with all symbols that the element reads, which serve as the dependencies for the write. For each element that may drop RPCs, a transfer function is generated that puts a new symbol representing the potential dropping event into the RPC special field. Reordering is handled in a similar way.

Symbolic execution of chains Symbolic execution of a chain produces a chain-level transfer function for the output RPC and for auxiliary channel outputs and state updates for all elements. This analysis essentially combined the element-level transfer functions.

Figure 6 (left) shows the end result of an RPC passing through that example chain. The auxiliary channel *out* outputs the exact RPC that came in; the internal state *idx* of the load balancer is updated by the opaque function $\mathbf{LB_S}$ whose parameters are the value of *idx* prior to RPC's arrival and the empty set for *RPC.drops* field (because the update will be different based on whether the RPC was dropped); and the final RPC has field values that depend on two opaque functions ($\mathbf{LB_R}$ and **FW**).

Comparing two chains Comparing two chains means comparing all of the chain-level transfer functions (result of symbolic execution). When the transfer functions have opaque functions, they are deemed equal iff their parameters are (recursively) equal; thus, this is basically a syntactic check. For strong observation consistency, we look for all transfer functions being equal. For weak observation consistency, we ignore the transfer functions for auxiliary channel outputs.

Figure 6 shows the results of this comparison when the chain on the left is compared against two other options with different element ordering. Option 1 is deemed weakly equivalent because the two transfer functions differ only for the auxiliary output of Logger. These transfer functions differ because the Logging element will record RPCs with a different *dst* field value once it is placed downstream of the Load Balancer. Option 2 is deemed not equivalent even though it

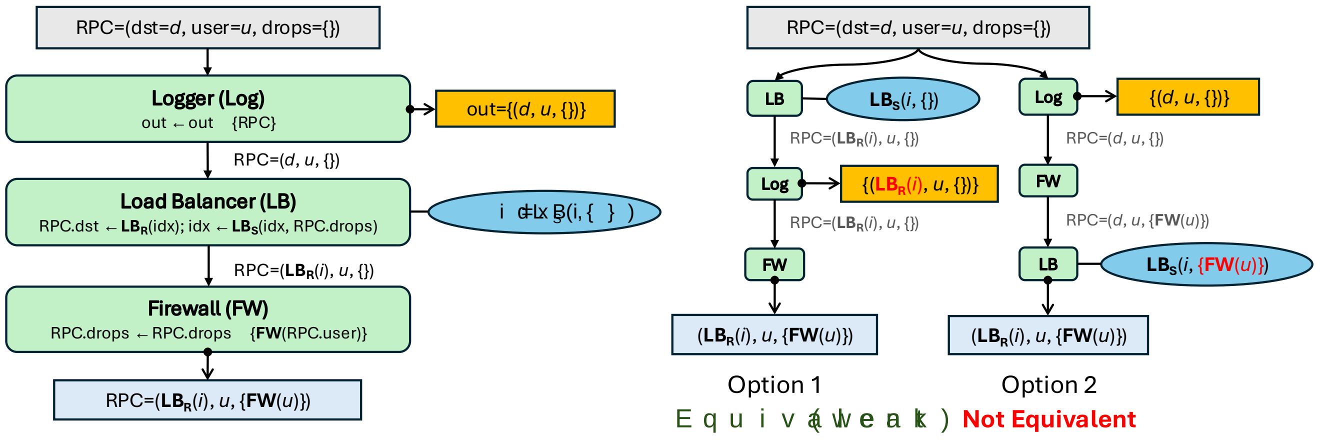


Figure 6: Equivalence-checking example. The transfer functions of the elements are inside the rounded green boxes. The rectangular boxes are channel outputs, and the oval ones are state updates determined by symbolic execution. Equivalence with strong semantics checks all outputs and state updates, and equivalence with weak semantics ignores yellow boxes (auxiliary outputs). The symbols marked red are those different from the original ones.

has identical transfer functions for Logger output and the final output RPC. The problem is that the transfer function for the state update for Load Balancer changes when it is placed downstream of the Firewall. This difference is surfacing the subtlety that Load Balancer will send subsequent RPCs to different destinations because the behavior of its round-robin logic is impacted by dropped RPCs.

5 Implementation

We implemented the AppNet compiler using 16K lines of Python and the controller using 3.1K lines of Go that integrates with Istio [39] and Kubernetes [33]. Developers use a custom Kubernetes resource [34] to provide AppNet specifications. The controller watches for changes to this resource and the application deployment (e.g., a new replica is launched) and updates the data plane platform appropriately.

AppNet currently supports three RPC processing platforms: (1) gRPC, a popular RPC library, for which we produce modules in Go that can be deployed as gRPC interceptors [29]; (2) EnvoyNative, for which we produce modules that C++ Envoy filters [31]; and (3) EnvoyWasm, a WebAssembly-based execution environment provided by Envoy [40], for which we produce modules in Rust that are later compiled to WebAssembly using the Rust compiler.

The overhead of these platforms increases in the listed order and the platforms differ in other ways as well (which makes the choice of the right platform a complex one). The gRPC modules run within the client or server address space, and so this platform cannot be used to run elements for which microservice code cannot be trusted. gRPC and EnvoyWasm allow dynamically updating running modules (without downtime); EnvoyNative does not. EnvoyNative and EnvoyWasm can be co-located with the client or server (as a sidecar proxy) or run in a remote proxy [13]; gRPC cannot be executed via a remote proxy.

AppNet assumes that its target platform provides access to RPC headers and payloads in plaintext (i.e., without mTLS). gRPC interceptors are always executed before mTLS is applied, and Envoy (in both sidecar and remote deployments) can be configured to terminate the mTLS connection [8]. In scenarios where end-to-end mTLS is required, AppNet will limit its target platforms to gRPC interceptors to ensure compliance with encryption requirements.

To synchronize shared state with strong consistency, we adopted the approach used by Envoy's global rate limiter [27] and StatelessNF [57]. Specifically, we use Redis [38] and route every state access through it. The synchronization for weak state consistency uses the same approach, but the synchronization is moved to the background.

Compiler benchmarks Finding the optimal runtime configuration takes 1.4s for 5-element chains that we use in § 6.2. Generating modules for gRPC interceptors and EnvoyWasm takes a few seconds, but it takes 1-2 minutes for EnvoyNative because that involves compiling the Envoy codebase.

6 Evaluation

Our evaluation aims to answer the following: (1) Can AppNet easily express common ANFs? (2) Can it reduce the overhead of application networks? (3) Does that reduction improve application performance?

6.1 Expressiveness

To evaluate the expressiveness of AppNet, we implement the 12 common ANFs in Table 1 which we identified based on a survey of common ANFs. As examples of more complex functionality, we also implement routing and load balanc-

	Shared state	LoC (AppNet)
Fault Injection [23]		11
Cache [22]	✓	14
Rate Limiting [24]	✓	25
Load Balancing [26]	✓	13
Logging [20]		10
Mutation [30]		7
Application Firewall [15]	✓	12
Metrics [28]		12
Admission Control [45]		21
Encryption [12]		28
Bandwidth Limit [21]		21
Circuit Breaking [32]		16
ServiceRouter [70]	✓	62
Prequal [82]	✓	88

Table 2: ANFs implemented in AppNet.

ing in Meta's ServiceRouter [70] and Google's Prequal [82]. Appendix F details these implementations.

We are able to express all 14 ANFs in AppNet. As a proxy for ease of use, Table 2 shows the lines of code (LoC) of each. The 12 common ANFs need only 7-28 lines, and ServiceRouter and Prequal need only 62 and 88 lines. Some elements have a control plane component (e.g., for maintaining the replica list for load balancing). These lines do not include those components. We find it encouraging that so few lines are needed to express the data plane of ANFs.

For comparison, we also implement the same ANFs in the native languages of the platforms (Go for gRPC, C++ for EnvoyNative, Rust for EnvoyWasm). These ANFs need 5-60x more lines than AppNet elements. C++ modules are the worst, needing 4x more lines than Go and Rust modules.

High-level languages can hurt performance if the generated code is not as efficient as hand-optimized code. We find that the processing time and CPU usage of AppNet elements is only 1-4% higher than similar filters [31] bundled with Envoy. See Appendix D for details.

The value of AppNet specifications lies not only in their compactness but also in enabling optimizations that lower overhead. We evaluate this next.

6.2 RPC Processing Overhead

To assess how well AppNet reduces the RPC processing overhead, we need a corpus of network specifications used in practice. But such a dataset does not exist to our knowledge, so we create a wide array of network specifications randomly and benchmark the range of overhead reduction. This method helps understand specification characteristics that lead to higher or lower overhead.

We create a dataset by randomly picking elements in the chain from the 12 common ANFs in Table 2. We study chains of sizes 3 and 5 and randomly assign each element to a subchain (client, server, or any). We also randomly assign platform constraints, which include "do not run in an RPC library" and "requires dynamic upgrades." The first constraint disallows gRPC, and the second constraint disallows EnvoyNative. Modulo these location and platform constraints, AppNet is free to optimize the chain as it sees fit.

Experimental setup We consider three metrics: (1) Service Time is the time to complete an RPC request when the service has a minimal load (at most one outstanding request); (2) Tail Latency is the 90th percentile latency with moderate load (30-40% CPU utilization); and (3) CPU Usage is the number of virtual cores used for processing RPCs.

We use Echo Server as the application which has a frontend microservice that sends an echo request and a backend microservice that responds with an echo response. This simple application helps us microbenchmark RPC processing overhead. We study more complex applications later. All applications in this paper use Go and gRPC, and all experiments use five replicas per microservice. The results are qualitatively similar for different replica counts.

We quantify AppNet's overhead reduction by comparing against two baselines: (1) *NoOpt* randomly assigns elements while honoring explicit location or platform constraints, (2) *LocalOpt* favors platforms with lower processing costs (first gRPC, then EnvoyNative, then EnvoyWasm) and prioritizes placements that align with state dependencies (e.g., placing elements that depend on client replicas on the client side). NoOpt captures what may happen when developers do not attempt any performance optimization, and LocalOpt captures what an informed developer might do—optimize for individual elements—excludes chain-wide optimizations that are difficult to do manually. Both baselines maintain the element ordering of the input specification and make placement decisions for elements in order. Thus, choices available for an element depend on the choices made for upstream elements.

We use Cloudlab [51] machines with two 16-core Intel Xeon Gold 6142 CPUs (2.6 GHz) and 384GB RAM, Ubuntu 20.04 (Linux kernel v5.4.0), Kubernetes v1.28.13, and Envoy 1.30.5. We disable TurboBoost, CPU C-states, and dynamic CPU frequency scaling to reduce measurement variance. We use wrk [41] and wrk2 [42] for load generation as well as high-precision measurement of latency. The remote proxy is shared by all replicas of the client and server microservices.

6.2.1 Overhead reduction

Figure 7 compares the performance of LocalOpt and AppNet with strong and weak consistency against NoOpt. It plots the reduction in the three metrics for 30 randomly selected 5-element chains. The absolute metrics of NoOpt (not shown in the graph) are 0.8–2.0 ms for service time, 2.5–12.1 ms for tail latency, and 10.9–28.6 virtual cores for CPU usage. These numbers exclude the processing time and CPU usage of the Echo Server application itself.

Both LocalOpt and AppNet show improvements across all three metrics. LocalOpt achieves a median reduction of 25% in service time, 27% in tail latency, and 22% in CPU usage.

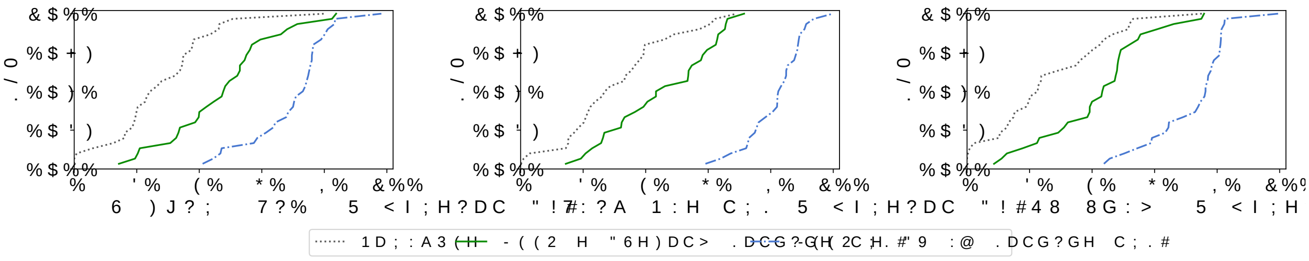


Figure 7: Reduction in RPC processing overhead compared to NoOpt. There are 5 elements in each chain.

AppNet, even when enforcing strong consistency, performs substantially better, with median reductions of 47% in service time, 44% in tail latency, and 42% in CPU usage. With weak consistency, AppNet further reduces overhead. Median reduction is 74-83%.

The degree of overhead reduction varies significantly across chains, influenced by which elements are present and location, platform constraints. By looking at the underlying data, we draw the following conclusions: the presence and placement of shared state elements within the chosen chain determine the potential for reducing synchronization overhead. The greatest reductions occur when multiple shared state elements are present and can be co-located and combined. Chains that allow more flexible element placement tend to yield higher gains. In contrast, chains with strict placement constraints limit the ability to group elements on one side and bypass a platform (e.g., a sidecar) on the other. Noticeable gains are also observed when elements that drop RPCs are positioned early in the chain, enabling them to execute sooner. AppNet's optimization algorithm is able to sift through these opportunities and find performant configurations. In contrast, LocalOpt uses a simplistic approach—selecting platforms based on processing costs and aligning state dependencies—and misses cross-element and cross-platform optimization opportunities.

6.2.2 Impact of available optimization opportunities

To shed light on overhead reduction as a function of available optimization opportunities, we now consider deployment environments that are limited to a single platform (but still multiple locations) or have shorter chains. These experiments are interesting also because some organization may not employ multiple platforms. The optimization opportunities that AppNet can tap in each platform differ. RPC libraries, which can only be used with fully trusted applications, are always in the path and have low overhead. So, they only benefit from reordering and moving elements between clients and servers (e.g., based on state synchronization costs). EnvoyNative has additional benefits when it can be completely bypassed (by moving all elements to one location). EnvoyWasm, which is used when dynamic upgrades are preferred, additionally benefits from consolidating multiple elements into one module because that reduces the number of invocations of the WebAssembly VM.

Figure 8 plots the overhead reduction for all three single-platform environment and the all-platforms environment. The experimental methodology is similar to the last section except that there are no constraints related to application trust and dynamic upgrades. In addition to 5-element chains, it plots 3-element chains which offer fewer optimization opportunities.

For 5-element chains, the median overhead reduction is 3–48% for the median, p10 is 0–24%, and p90 is 27–76%. The extent of the reduction varies across platforms due to the distinct characteristics of each as noted above. Even when reordering and client-vs-server placement are the only optimization opportunities (as in gRPC), we see a noticeable overhead reduction, especially for tail latency and CPU usage. AppNet significantly reduces overhead even for 3-element chains,though the reduction tends to be lower than for 5-element chains.

The reduction is generally higher as optimization opportunities increase. It is highest for the all-platforms environment because that also opens up the opportunity of using the platform with the least overhead when permitted by user policies. Due to implementation-level differences (Go vs. Rust), the state synchronization cost in gRPC is higher than in Envoy, which explains why, in some cases, gRPC optimizations are greater than those of Envoy.

LocalOpt often provides little to no benefit for single-platform environments. Here, it primarily aligns state dependencies, which helps only when an element is unconstrained and NoOpt (randomly) picks suboptimal placement. Such occurrences are rare.

6.2.3 Impact of weak state and observation consistency

Figure 9 shows the impact of using weak state consistency and weak observation consistency separately. It plots the reduction in overhead relative to strong consistency for both. For each curve, the set of chains is limited to those where that consistency type is relevant, i.e., there is at least one shared state variable for state consistency and at least one auxiliary output for observation consistency.

We see that weak state consistency leads to substantial reductions across a variety of chains, with median reduction

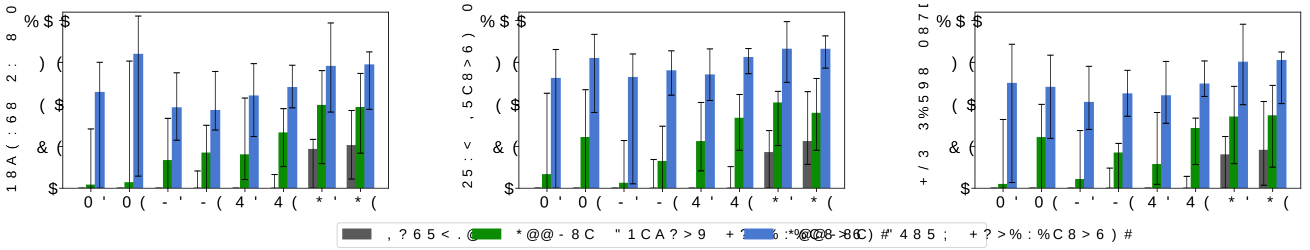


Figure 8: Reduction in RPC processing overhead compared to NoOpt. In the bar labels, the letter denotes available platforms (R = gRPC, N = EnvoyNative, W = EnvoyWasm, and A = all platforms), and the number denotes the chain length. The bar height denotes the median reduction and the error bars denote the 10th and 90th percentiles.

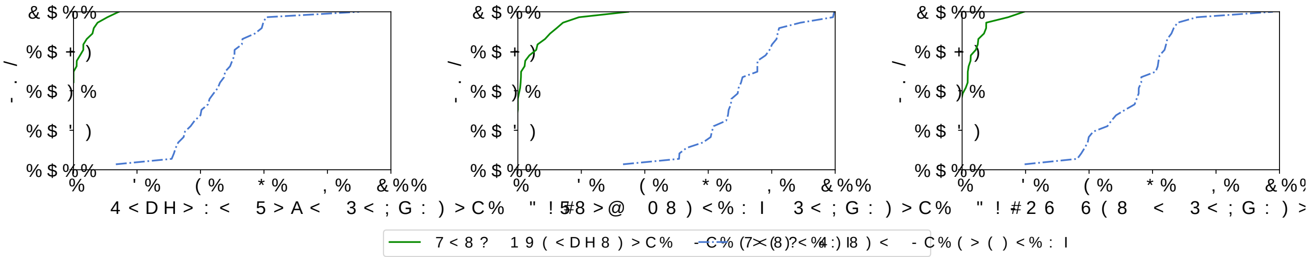


Figure 9: RPC processing overhead reduction with the two types of weak consistency relative to strong consistency.

ranging from 45% to 69%. This reduction stems from making state synchronization a background task. The degree of reduction depends on the number of shared-state elements included and whether they can be consolidated based on their placement constraints.

The overhead reduction from weak observation consistency is lower. A primary source of gain from weak observation consistency is moving a dropping element upstream, which reduces downstream work. But when the drop rate is low, as in our experiments, the gain is small. The gain will be higher in scenarios with higher drop rates (e.g., a rate limiter under heavy load). Despite low drop rates, we see a reduction in service time, tail latency, and CPU usage in 38-55% of the cases. Much of this reduction stems from weak observation enabling AppNet to reorder and consolidate more elements.

6.3 Application performance

We now evaluate the extent to which the reduction in RPC processing overhead in AppNet translates to end-to-end improvement in application-level performance. This section studies the Hotel Reservation application [53] (Figure 1), and Appendix C studies the Online Boutique application [36]. We use Hotel Reservation's search query, which retrieves available hotels based on location and check-in/check-out dates and involves calls to the Frontend, Search, Rates, Geo, Profile, Reserve microservices, and their associated MongoDB and Memcached storage systems. For each communication edge between microservices, we generate 5-element chains randomly (as in § 6.2.1). We measure end-to-end latency and total CPU.

Figure 10 shows the reduction in latency and CPU compared to NoOpt. The baseline metrics (not shown) are 7.2–14.1 ms for service time, 35.6–92.4 ms for tail latency, and 136.4-219.3 virtual cores for CPU usage.

LocalOpt demonstrates moderate performance gains across all metrics and chain configurations, with median reductions of 14% in service time and 11% in both tail latency and CPU usage. AppNet achieves more substantial improvements: with strong consistency, median reductions are 35% for service time, 29% for tail latency, and 26% for CPU usage. Weak consistency improves these numbers to 49%, 41%, and 42%, respectively.

Our results show that reducing RPC processing overhead leads to substantial improvements in end-to-end application performance. This is because this overhead is a substantial contributor toward application latency and CPU usage [5, 66, 71, 86]. Lowering this overhead directly helps applications by alleviating the tax of the microservices architecture.

7 Related Work

AppNet draws on four themes of prior work.

High-level network programming We draw heavily from the large body of work of high-level network programming, which has been explored in several contexts. Early work such as Click [60] focused on packet-processing in software switches. We borrow from this domain the idea of expressing

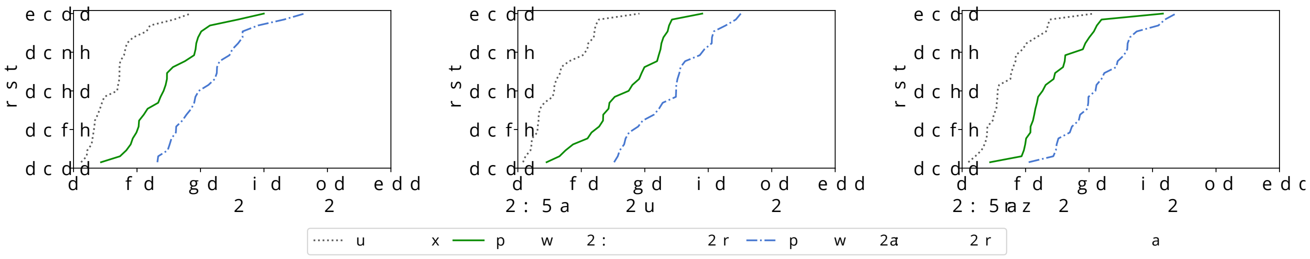


Figure 10: Performance improvement of Hotel Reservation compared to the NoOpt.

network functionality as a chain of elements. A key difference between AppNet and these systems is that AppNet includes abstractions for processing *inside* an element, which allows it to optimize the chain end-to-end.

A second wave focused on network-wide packet forwarding, catalyzed by software-defined networking (SDN) [43, 52, 68], where the focus was on layer-3/4 packet forwarding with capabilities like determining next hops and filtering. In contrast, AppNet targets rich, stateful functions and a setting where the network can scale dynamically.

The advent of programmable switches engendered work on high-level programming for them [46, 54, 75]. We borrow the match-action primitive from this work. Our match-action rules are richer, corresponding to the need for rich, layer-7 processing and aided by not being bound by hardware constraints. AppNet abstractions include shared state handling and optimizations across elements in a chain.

A parallel line of work focuses on NFs (network functions) in middleboxes. NetBricks [63] improves performance by composing multiple functions using a safe runtime, so they do not have to use separate VMs. It does not provide abstractions for individual functions, which allow us to safely combine multiple functions. Rubik [62] provides a language for efficient, low-level packet handling (e.g., assembling IP fragments and reconstructing TCP streams). AppNet abstractions focus on the RPC layer.

Reducing the overhead of application networks. There is broad awareness in the industry about the overhead of application networks [2, 5]. Several projects [13, 17] target this challenge by changing the software design of service meshes. Meta built ServiceRouter [70] to lower application networking overhead for specific functions, such as routing requests based on applications' sharding keys, and Alibaba built Canal Mesh [76] to address the overhead of sidecars by moving processing to a remote proxy. Unlike AppNet, none of these works focus on high-level programming to support diverse applications, deployment modes, and capabilities.

NF Optimizations. Optimizing NF chains for performance and resource efficiency has been an active area of research [48, 58, 64, 72, 78]. OpenBox [48] reduces redundancy in NF chains by eliminating overlapping logic across functions. Metron [58] builds on this by offloading parts of the merged logic to programmable switches. NFP [78] and Maestro [64] leverage parallelization to accelerate both inter- and intra-NF processing. In general, these systems focus on processing lower-level protocols and raw network packets, which have goals distinct from those of AppNet. Additionally, traditional NFs typically operate within dedicated middleboxes, while AppNet targets RPC processing tasks that can run in diverse environments, including RPC libraries, sidecar proxies, or middlebox-style remote proxies.

State management in NFs Works on scaling NFs show that managing state is a central challenge. Split/Merge [67], S6 [81], StatelessNF [57], SwiSh [84] manage different types of state based on its properties, and OpenNF [55] develops techniques to (approximately) infer properties of state in existing NFs. While these works do not focus on high-level message processing abstractions, they have influenced our design—our primitives enable a compiler to automatically infer important properties of state, and our state handling (placement and partitioning) draws on lessons from them.

8 Conclusions

AppNet enables expressive and high-performance application networks by decoupling the specification of ANF from its implementation. Its specification language is easy to use—common ANFs can be expressed in only 7–28 lines of code. It generates high-performance implementations based on an approach that reasons about estimated performance and semantic equivalence of possible implementations. Our experiments show that this approach lowers RPC processing latency and CPU usage by, respectively, 82% and 75%.

Acknowledgements

We thank the NSDI reviewers and our shepherd, Jiaqi Gao, for their feedback. This work was supported in part by UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware), by NSF Grant 2402695 and 2402696, and by ACE, a center that is part of DARPA's JUMP 2.0.

References

[1] Enabling GZIP Response Compression with EnvoyFilter. https://karlstoney.com/enabling-gzip-compression-with-envoyfilter/, 2019.

[2] Performance Benchmark Analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/, 2019.

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

[4] Embracing eventual consistency in SoA networking. https://blog.envoyproxy.io/embracing-eventual-consistency-in-soa-networking-32a5ee5d443d, 2022.

[5] Istio: Performance and Scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2022.

[6] Performance Impacts of an Istio Service Mesh. https://pklinker.medium.com/performance-impacts-of-an-istio-service-mesh-63957a0000b, 2022.

[7] 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, 2022.

[8] Understanding Istio Ambient Ztunnel and Secure Overlay. https://www.solo.io/blog/understanding-istio-ambient-ztunnel-and-secure-overlay, 2022.

[9] Use Dataflow SQL. https://cloud.google.com/dataflow/docs/guides/sql/dataflow-sql-intro, 2022.

[10] Better Load Balancing: Real-Time Dynamic Subsetting. https://www.uber.com/en-US/blog/better-load-balancing-real-time-dynamic-subsetting/, 2023.

[11] Go gRPC Middleware. https://github.com/grpc-ecosystem/go-grpc-middleware, 2023.

[12] gRPC Encryption. https://github.com/grpc/grpc-go/blob/master/examples/features/encryption/README.md, 2023.

[13] Introducing Ambient Mesh. https://istio.io/v1.15/blog/2022/introducing-ambient-mesh/, 2023.

[14] Linkerd Features. https://linkerd.io/2.14/features/, 2023.

[15] OWASP Coraza Web Application Firewall. https://coraza.io/, 2023.

[16] AppNet: Expressing, easy-to-build, and high-performance application networks. https://appnet.wiki/, 2024.

[17] Cilium Service Mesh. https://cilium.io/use-cases/service-mesh/, 2024.

[18] Cloud Service Mesh with proxyless gRPC. https://cloud.google.com/service-mesh/docs/service-routing/proxyless-overview, 2024.

[19] Envoy. https://www.envoyproxy.io/, 2024.

[20] Envoy: Application logging. https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/application_logging, 2024.

[21] Envoy: Bandwidth limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/bandwidth_limiting, 2024.

[22] Envoy: Cache filter. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/cache_filter, 2024.

[23] Envoy: Fault Injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/, 2024.

[24] Envoy: Global rate limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/global_rate_limiting, 2024.

[25] Envoy: HTTP Routing. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/http/http_routing, 2024.

[26] Envoy: Load Balancing. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing, 2024.

[27] Envoy: Rate limit. https://github.com/envoyproxy/ratelimit, 2024.

[28] Envoy Statistics. https://istio.io/latest/docs/ops/configuration/telemetry/envoy-stats/, 2024.

[29] gRPC Interceptors. https://grpc.io/docs/guides/interceptors/, 2024.

[30] gRPC-tools. https://github.com/bradleyjkemp/grpc-tools, 2024.

[31] HTTP filters. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters, 2024.

[32] Istio: Circuit Breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/, 2024.

[33] Kubernetes. https://kubernetes.io/, 2024.

[34] Kubernetes: Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 2024.

[35] Linkerd: the world's most advanced service mesh. https://linkerd.io/, 2024.

[36] Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2024.

[37] Protocol Buffers. https://protobuf.dev/, 2024.

[38] Redis. https://redis.io/, 2024.

[39] The Istio Service Mesh. https://istio.io/, 2024.

[40] WebAssembly in Envoy. https://github.com/proxy-wasm/spec/blob/master/docs/WebAssembly-in-Envoy.md, 2024.

[41] wrk. https://github.com/wg/wrk, 2024.

[42] wrk2. https://github.com/giltene/wrk2, 2024.

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

[44] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 29–43, 2016.

[45] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. O'Reilly Media, Inc., 2016.

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

[47] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

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

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

[50] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[51] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 1–14, 2019.

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

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

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

[55] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid,

Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2014.

[56] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (ATC 23)*, pages 419–432, 2023.

[57] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, 2017.

[58] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron:NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, 2018.

[59] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[61] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. LemonNFV: Consolidating heterogeneous network functions at line speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1451–1468, 2023.

[62] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming network stack for middleboxes with rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 551–570, 2021.

[63] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, 2016.

[64] Francisco Pereira, Fernando MV Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1531–1550, 2024.

[65] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[66] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[67] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.

[68] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN programming with pyretic. *Technical Report of USENIX*, 2013.

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

[70] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[71] 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. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[72] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[73] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types.

In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[74] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 15–28, 2016.

[75] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

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

[77] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 65–76, 2014.

[78] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling network function parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 43–56, 2017.

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

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

[81] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, 2018.

[82] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. Load is not what you should balance: Introducing Prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, 2024.

[83] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, 2022.

[84] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

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

[86] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 142–157, 2023.

A Additional AppNet Design

A.1 Code generation

The AppNet compiler takes in the optimized runtime configuration and produces platform-specific software modules as well as deployment scripts. To handle requests for strongly consistent state, it translates these accesses into blocking network calls to external storage. In contrast, reads from weakly consistent state are performed locally, and writes are applied asynchronously. The compiler further reduces overhead by consolidating co-located elements, effectively removing boundaries introduced by virtualization or sandbox approaches [61, 63]. This consolidation applies to both the processing logic and state synchronization messages.

The generated deployment scripts configure traffic flow for each communication edge (microservices pair) such that sidecars are bypassed when the runtime configuration does not place any element there. (Today, all traffic of a microservice is intercepted by the sidecar, which imposes unnecessary overhead for RPC traffic that does not require processing.)

A.2 Consistent Updates

Application networks evolve frequently due to policy and workload changes (scaling). Service meshes today provide atomic updates for individual ANF, but ensuring consistency across a communication edge with multiple ANFs is left to developers. This approach is vulnerable to application downtime or policy violations [4]. For example, when migrating an application firewall, the developer may first detach the module on one side and then attach it on the other, potentially causing policy violations.

AppNet ensures consistent configuration updates such that RPCs traversing a communication edge are processed entirely by the old configuration or the new, and never a blend of the two. It uses a two-phase update approach [69, 83]. Each RPC is tagged with a version number, and configurations are updated to process only RPCs with the corresponding version. To transition from one configuration to the next, AppNet controller first installs the new configuration, guarded by the next version number. Once that is done, it enables the new configuration by stamping new RPCs with the next version number. Finally, once all RPCs with the old version number have completed, the old configuration is removed.

The header to support consistent upgrades adds 23 bytes to RPCs.

B Equivalence Checking Formalization

This appendix describes equivalence checking of AppNet in more detail. Table 3 lists all types of symbols with their explanations.

Symbolic abstraction of an element We symbolically abstract an element as a set of transfer functions in two steps. In the first step, we convert the element into a tuple of properties (`rf`, `wf`, `rs`, `ws`, `drop`, `reorder`, `channel`), where

- `rf`(read fields), `wf`(write fields), `rs`(read states), `ws`(write states) are sets of RPC field/state names that the element reads/modifies.
- `drop`, `reorder` are boolean values indicating whether the element is likely to drop RPCs or reorder input RPC stream. An element may reorder the RPC stream if it does concurrent processes or delays RPCs by differing amounts.
- `channel` is a set of auxiliary output channels in the element. Elements such as Logging and Metrics will send RPC records to these channels.

All of these properties are obtained by analyzing the (post-parsing) IR of AppNet specifications.

The second step converts these properties into transfer functions. Algorithm 1 illustrates this process. **wf**, **ws**, **ord** and **dr** in the transfer functions are opaque function symbols representing the dependencies of field modification, state modification, reordering decision, and dropping decision, respectively. The meaning of symbol subscripts are explained in Table 3.

Algorithm 1 Symbolic Abstraction Algorithm

Input: *AppNet specification of an element*
Output: *A set of transfer functions for the element*

```
Function SymbolicAbstraction(e):
    (rf, wf, rs, ws, drop, reorder, channel) ← get_property_tuple(e)
    $TFs \leftarrow \{\}$    # TFs is a set for collecting transfer functions
    $rv \leftarrow \{\tilde{r}.f : f \in \texttt{rf}\} \cup \{\tilde{s}.v : v \in \texttt{rs}\}$
    foreach $f \in$ wf do
        $TFs$.add($\tilde{r}.f \leftarrow \mathbf{wf}_{(\mathbf{i},\mathbf{f})}(rv)$)
    end
    foreach $v \in$ ws do
        $TFs$.add($\tilde{s}.v \leftarrow \mathbf{ws}_{(\mathbf{i},\mathbf{v})}(rv \cup \tilde{r}.drops \cup \tilde{r}.reorders)$)
    end
    foreach $ch \in$ channel do
        $TFs$.add($ch \leftarrow ch \cup \{\tilde{r}\}$)
    end
    if reorder $=$ true then
        $TFs$.add($\tilde{r}.reorders \leftarrow \tilde{r}.reorders \cup \{\mathbf{ord_i}(rv)\}$)
    end
    if drop $=$ true then
        $TFs$.add($\tilde{r}.drops \leftarrow \tilde{r}.drops \cup \{\mathbf{dr_i}(rv)\}$)
    end
    return $TFs$
```

Symbolic execution of a chain of elements Algorithm 2 shows the process of sequentially iterating over the chain and concatenating element-level transfer functions to obtain chain-level transfer functions. $\mathbf{F_{f_x}}$ and $\mathbf{S_{v_x}}$ are primitive symbols representing the initial values of RPC fields and state variables.

Comparing two chains Algorithm 3 shows how we compare two chains under desired consistency level. For weak consistency, we require the transfer functions of output RPC and final states to be identical. For strong consistency, we also compare the transfer functions of auxiliary channels.

Correctness Proof Two chains are equivalent if and only if for any input RPC stream, the contents of output RPCs, the outputs of auxiliary channels, and all RPC-related behaviors such as dropping, are identical. The only differences permitted are those that could occur between two copies of the same chain, such as the differences caused by randomness. In our formalization, we assume that all opaque functions additionally take an implicit seed that makes the function deterministic, and the same seed is used for both chains under comparison to eliminate randomness. We also assume that the RPC stream is not reordered (e.g., during network transmission—by explicitly including the network as a reordering element, we can remove this assumption) unless it

Type	Explanation	Examples
Properties	obtained from AppNet specifications, used for symbolic abstraction	`rf` (read fields, a set of field names), `ws` (write states, a set of state variable names), `channel`
Primitive Symbols	symbols representing initial values of fields and states	$\mathbf{F_f}$ (the initial value of $RPC.f$), $\mathbf{S_v}$ (the initial value of state variable v)
Function Symbols	symbols representing the opaque internal logic of elements, the parameters of which are symbolic values that it depends on	$\mathbf{wf_{(i,f)}}(S)$ (the logic of the i_{th} element writing to $RPC.f$, where S is a set of symbols the element reads)
Dictionary	A dictionary maps field name/variable name to its value symbol. If $a : b \in \tilde{s}$, then $\tilde{s}.a \triangleq b$.	$\tilde{r} = \{f_1 : \mathbf{F_{f_1}}, \cdots, f_n : \mathbf{F_{f_n}}, drops : \{\}, reorders : \{\}\}$.

Table 3: Symbol Explanation and Examples

```
Algorithm 2 Symbolic Execution Algorithm
Input: a chain configuration $c = (e_1, \cdots, e_n)$, i.e., a list of AppNet specifications
Output: chain-level transfer functions of output RPC, states and auxiliary channels

Function SymbolicExecution($c$):
    $chainTFs \leftarrow \{$
    $\tilde{r} \leftarrow (f_1 : \mathbf{F_{f_1}}, \cdots, f_n : \mathbf{F_{f_n}}, drops : \{\}, reorders : \{\}),$
    $\tilde{s} \leftarrow (v_1 : \mathbf{S_{v_1}}, \cdots, v_m : \mathbf{S_{v_m}})$
    $\}$
    foreach $e_i \in c$ do
        $TFs \leftarrow$ SymbolicAbstraction($e_i$)
        $chainTFs \leftarrow$ concatenate($chainTFs$, $TFs$)
    end
    $rpcTFs, stateTFs, channelTFs \leftarrow$ split(ChainTFs)
    return $rpcTFs, stateTFs, channelTFs$
```

```
Algorithm 3 Equivalence Checking
Input: Two chain configurations $c_1, c_2$ and desired consistency level $L$
Output: A boolean indicating whether $c_1$ and $c_2$ are equivalent under $L$

$rpc_1, state_1, channel_1 \leftarrow$ SymbolicExecution($c_1$)
$rpc_2, state_2, channel_2 \leftarrow$ SymbolicExecution($c_2$)
if $L$ = 'weak' then
    return $rpc_1 = rpc_2$ and $state_1 = state_2$
else
    return $rpc_1 = rpc_2$ and $state_1 = state_2$ and $channel_1 = channel_2$
end
```

meets elements with property $reorder = \textbf{true}$.

We need to prove that for two chains, if the algorithm produces the same transfer functions, then the two configurations are equivalent. The proof below focuses on weak consistency, i.e., analyzing the output RPC and states; strong consistency just additionally checks the auxiliary channels, which follows a similar approach.

We start by showing that if two symbolic transfer functions are identical, then the concrete value or dropping/reordering decisions they represent are equivalent. We prove the statement by structural induction.

- Base case: primitive symbols include $\mathbf{F_{f_1}}, \cdots, \mathbf{F_{f_n}}$ and $\mathbf{S_{v_1}}, \cdots, \mathbf{S_{v_m}}$, which represent initial values of RPC fields and state variables. Identical primitive symbols indicate that they represent the exact same concrete initial value, which are equivalent.
- Inductive case: our symbolic execution system has two kinds of compound symbol structures:
    - $\mathbf{wf/dr/ord_{(i,f)}}(\{\tilde{r}.f : f \in rf\} \cup \{\tilde{s}.v : v \in rs\})$: these three symbols represent the modified field value that the i_{th} element writes into the RPC, the dropping decision, and the reordering decision, respectively. Considering that the essence of RPC processing inside an ANF is a black-box function that maps input RPCs and states to output RPCs and RPC-related operations, if all inner symbols in the parenthesis are identical, which means all the RPC fields and states the element reads are equivalent, then the modified fields or dropping/reordering decisions are also equivalent.
    - $\mathbf{ws_{(i,v)}}(\{\tilde{r}.f : f \in rf\} \cup \{\tilde{s}.v : v \in rs\} \cup RPC.drops \cup RPC.reorders)$: this symbol represents the modified value of state variables. State updates are different from RPC processing in that not only RPC contents and previous state values matter, but also whether an RPC arrives at the element and the order of RPC stream will influence the result. Therefore, our algorithm additionally includes the set of previous dropping/reordering decisions into the parentheses, and if all inner symbols are identical, then the modified state value should also be equivalent.

Therefore, if the transfer functions of output RPC are identical, then equivalence of single input RPC could be guaranteed.

The next step is to show why it is correct to apply analysis

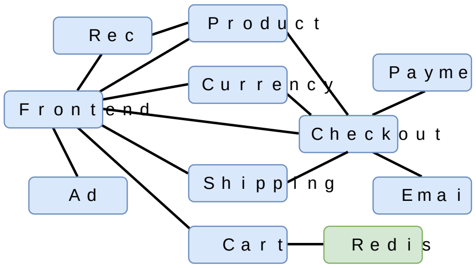


Figure 11: Online Boutique Microservice [36].

on single input RPC to stateful chains that receive a stream of RPCs. Let's consider two scenarios:

- The chain is stateless, i.e., there is no stateful element inside the chain. In this case, RPCs will not interfere with each other, and therefore reasoning about a stream of input RPCs is equivalent to reasoning about each RPC independently.

- The chain contains stateful elements. Considering that (1) The reordering events symbols inside the transfer functions of states are identical, which guarantees that stateful elements see RPCs in the same order in two configurations, and (2) The transfer functions of states, in a whole, are identical, which means given the same pre-states, the n_{th} RPC will produce the same post-states which serve as the pre-states of the $(n+1)_{th}$ RPC, we can inductively conclude that all RPCs in the stream will result in identical symbolic output RPCs.

Therefore, we can conclude that if the algorithm produces identical symbolic output RPC, dropping events and modified states, two chain configurations are equivalent.

C Additional Application Benchmark

In this section, we demonstrate AppNet's ability to reduce RPC processing overhead using the Online Boutique application [36]. It consists of 11 microservices that communicate via gRPC (Figure 11). Since it is implemented in multiple languages (Python, C#, Java, Node.js and Go) and AppNet supports only the Go version of gRPC, we port all services to Go. Our experimental methodology follows the same approach as in § 6.3.

Figure 12 presents the results, showing the performance improvements of the LocalOpt and AppNet relative to the "Fully Pinned" baseline. The baseline metrics (not shown) across all chain configurations range from 14.1–19.2 milliseconds for service time, 90.1–162.2 milliseconds for tail latency, and 106.4–163.8 virtual cores for CPU usage.

Consistent with the results from the Hotel Reservation benchmark, the LocalOpt achieves moderate performance gains across all metrics and chain configurations, with median reductions of 8% in service time and tail latency and 9% in CPU usage. In contrast, AppNet delivers greater improvements: under strong consistency, median reductions reach 22% for service time, 20% for tail latency, and 24% for CPU usage. When weak consistency is applied, these gains increase further to 36%, 37%, and 44%, respectively.

The overall improvements achieved by both the LocalOpt and AppNet are smaller in the Online Boutique application compared to the Hotel Reservation benchmark. This is primarily due to the larger baseline application latency and CPU usage, which reduce the relative impact of RPC processing optimizations. Nevertheless, AppNet still provide substantial performance benefits, demonstrating its effectiveness across different application workloads.

D Abstraction Tax

Systems based on high-level languages sometimes pay a performance tax when the generated code is not as efficient as hand-written code. We quantify this tax for AppNet by running our generated modules against two sets of handwritten modules. Each module is run in isolation (i.e., not as part of a chain), and we measure the same three metrics.

The first set of handwritten modules is Envoy's built-in filters. Envoy has a set of built-in filters written in C++ that users can configure to use with Envoy. Versions of all 12 filters in Table 2 are available except for encryption. Figure 13a shows how our generated EnvoyNative modules compare to these filters. We see that the tax is low. The median increase in latency and CPU is 1-4%, with the worst-case overhead being 3-7%.

The second set of handwritten modules are gRPC interceptors and Envoy WebAssembly filters that we wrote (because we were not able to find usable third-party implementations. Figure 13b presents the results. We see a similar tax to what we saw for Envoy built-in filters.

We investigated the sources of this tax and found that significant contributors include coarse-grained locking of shared state and the data structures used to maintain metadata for matching RPC messages across processing functions. We speculate that this can be reduced with a more careful analysis of the processing logic. Nevertheless, as demonstrated earlier, this low tax is more than compensated for by the optimizations enabled by AppNet's abstractions.

E Details of Optimization Algorithm

As the number of elements in the chain increases, the search space for potential optimal configurations soon becomes large and intractable. One of the possible approaches is to formulate the optimization into an integer programming (IP) problem and invoke a solver to generate solutions. However, solver-based methods are not scalable enough and struggle to incorporate validity checks as IP constraints. To address these challenges, AppNet leverages a multi-start simulated annealing (MSA) algorithm [79] to efficiently find high-quality solutions for long chains. Specifically, during each cooling

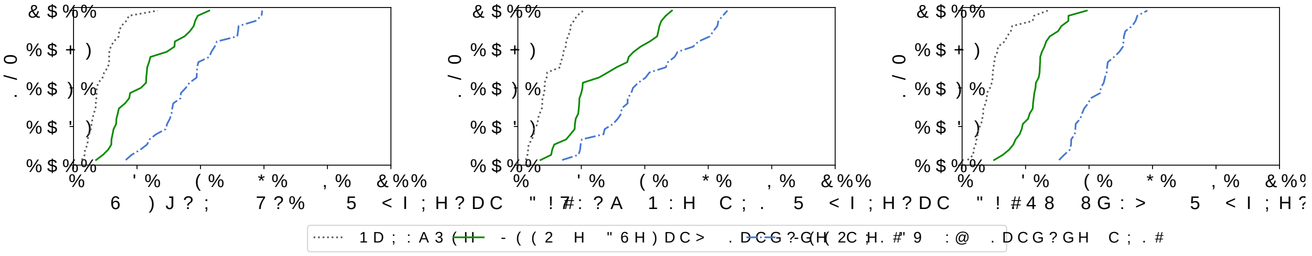
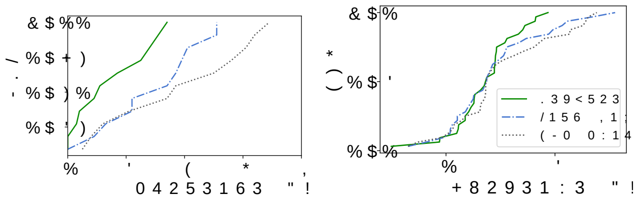


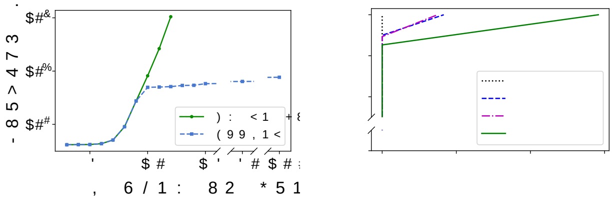
Figure 12: Performance improvement of Online Boutique compared to NoOpt.



(a) Envoy built-in.

(b) Hand-coded.

Figure 13: Performance of AppNet generated elements compared to Envoy built-in and hand-coded elements.



(a) Solving time comparison

(b) Cost ratio CDF

Figure 14: Efficiency and optimality comparison between multi-start simulated annealing and brute-force searching.

process, we iteratively consider valid (i.e., semantics preserving) mutations of the current solution and decide whether to adopt the new configuration according to some probabilities related to the cost difference and dropping temperature. Algorithm 4 shows our strategy of configuration mutation. To further mitigate the risk of local optima, we repeatedly restart the cooling process by resetting the temperature. The algorithm terminates when there's no improvement in cost over the last 15 cooling processes or the time budget runs out. Our final optimization algorithm combines MSA with brute-force searching: for chains with fewer than 10 elements, we exhaustively enumerate all possible combinations of configurations to guarantee optimality; for longer chains, we apply the MSA heuristics to achieve near-optimal solutions efficiently.

To evaluate the efficiency and solution quality of our approach, we compare multi-start simulated annealing against brute-force searching by randomly generating 100 specifications for each chain size ranging from 3 to 12 and recording the solving time and final cost. Figure 14a shows the speed

Algorithm 4 Solution Mutator in Simulated Annealing

```
Input: The current solution
Output: A new valid solution mutated from the original one

Function SolMutation(s):
    repeat
        $s' \leftarrow s$
        switch $coin \leftarrow random(0,1)$ do
            case $0 \le coin < 0.4$:
                change the position of one element in $s'$
            case $0.4 \le coin < 0.8$:
                swap the positions of two elements in $s'$
            case $0.8 \le coin \le 1$:
                change the platform of one element in $s'$
        end
    until $PassValidityCheck(s')$;
    return $s'$
```

comparison. Brute-force searching becomes computationally prohibitive due to its factorial complexity, whereas MSA is 473x faster for 12-element chains and remains practical even when the chain size increases to 100. Figure 14b presents the CDF of the cost of multi-start simulated annealing relative to brute-force searching. MSA successfully identifies the optimal configuration in more than 98% of cases when the chain size is no more than 12, and the worst-case cost increase is lower than 1.5%.

F Details of ServiceRouter and Prequal Implementation

In this section, we describe the implementation details of ServiceRouter and Prequal in AppNet.

ServiceRouter [70] is Meta's global service mesh system. We implement the routing and load balancing as described in the paper. Specifically, it routes RPCs based on their key (for sharded services) and uses adaptive load estimation along with the power of two choices to balance load across service replicas. The AppNet implementation consists of a control plane component, a client-side element, and a server-side element. The control plane component maintains the shard information for each service and maintains an up-to-date global knowledge about the load of each service replica. The client-side element maintains two states: a shard map and a load map, which store the mapping between request keys and shard replicas, and replicas to their current load, respectively. For each request, the client-side element first checks the freshness of its local states, queries the control plane if they are stale, and applies the power-of-two choices on the corresponding shard replicas. Upon receiving a response, the client-side element updates the load map using the piggybacked load information embedded in the response. The server-side element monitors load locally by maintaining a counter. This counter is incremented when a request is received and decremented when the application sends a response. The current load value is embedded as an RPC header in outgoing responses

Prequal [82] is a load balancer used by YouTube and other production systems at Google. It selects server replicas based on estimated latency and active in-flight requests. The AppNet implementation of Prequal also comprises three main components: a control plane, a client-side element, and a server-side element. The control plane actively probes and stores the load of each replica along with its request latency for various load levels. The client maintains a local cache of the control plane's state. For each request, it applies the power-of-N choices to sample a subset of replicas and retrieves their load information from either the local cache or the control plane, depending on the cache's freshness. It then uses the hot-cold lexicographic rule, as described in the paper, to select the replica for routing the request. The server-side component monitors the current load and maintains a mapping between load levels and the latencies of requests processed at each load point.