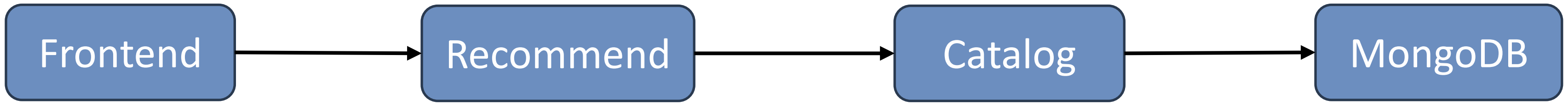
Bridging Expressiveness and Performance for Service Mesh Policies

Divyanshu Saxena, William Zhang, Shankara Pailoor,
Işıl Dillig, Aditya Akella

TEXAS
The University of Texas at Austin

Increasing Adoption of Microservices

- More than 85% of large enterprises (5000+ employees) are already using microservice architecture for their applications [1].
- Software developers, on average, develop 50% of their applications using microservices [2].

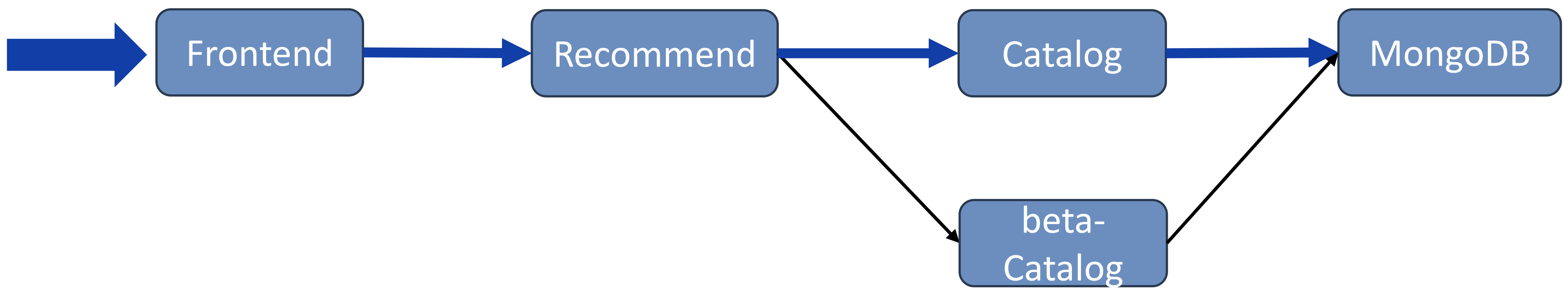


[1] Global usage of microservices in organizations 2021, by organization size. https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/

[2] Microservices in the enterprise, 2021: Real benefits, worth the challenges. https://www.ibm.com/downloads/documents/us-en/10a99803ce2fdd73

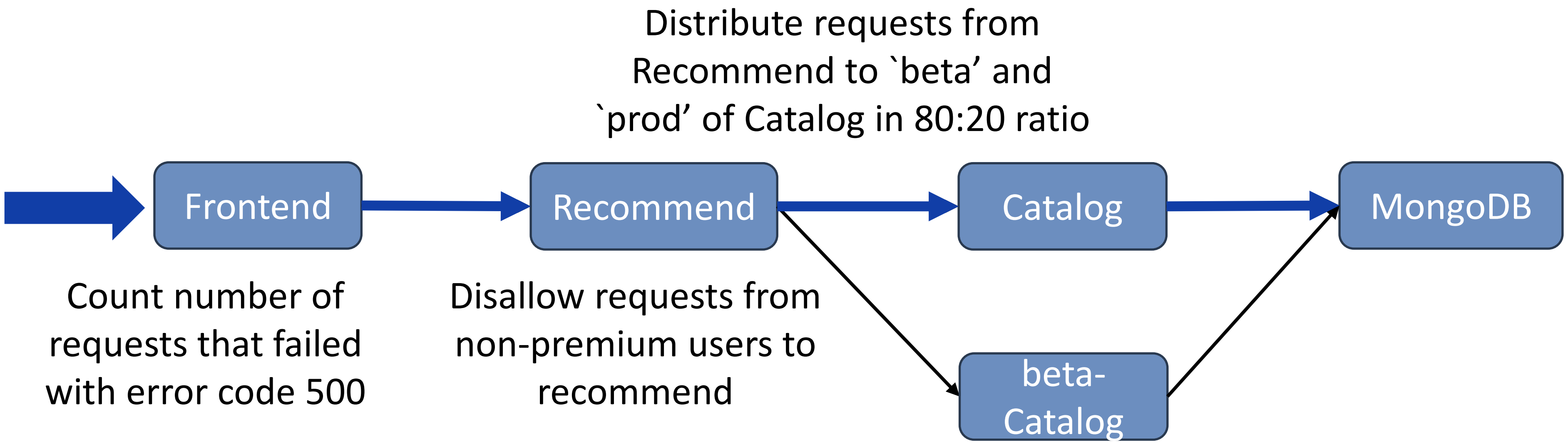
Communication in Microservice Applications

- Complex traffic patterns necessitate communication policies.



Communication in Microservice Applications

- Complex traffic patterns necessitate communication policies.

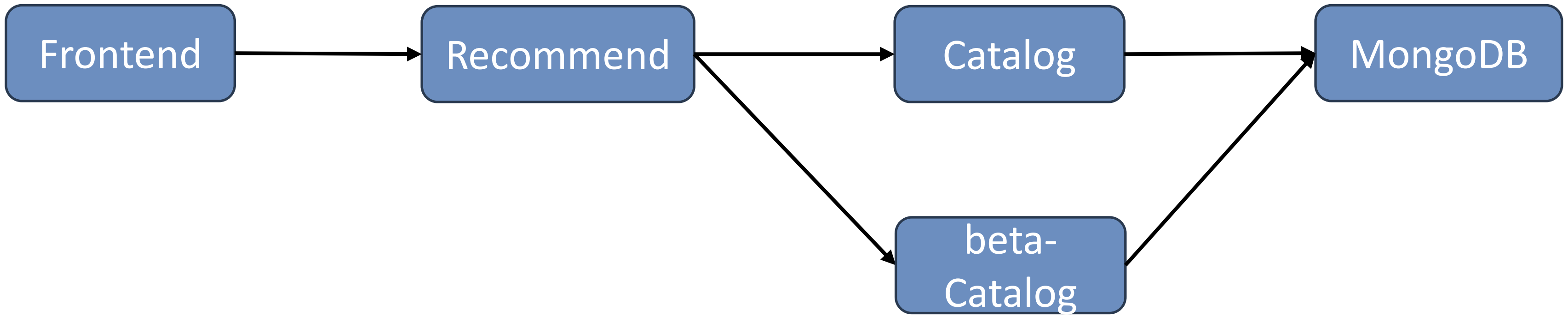


Implementing policies in the application code complicates ***development, deployment and configuration***

Service Meshes for Policy Enforcement

Enforce policies inside a *sidecar* container, running *alongside* app containers.

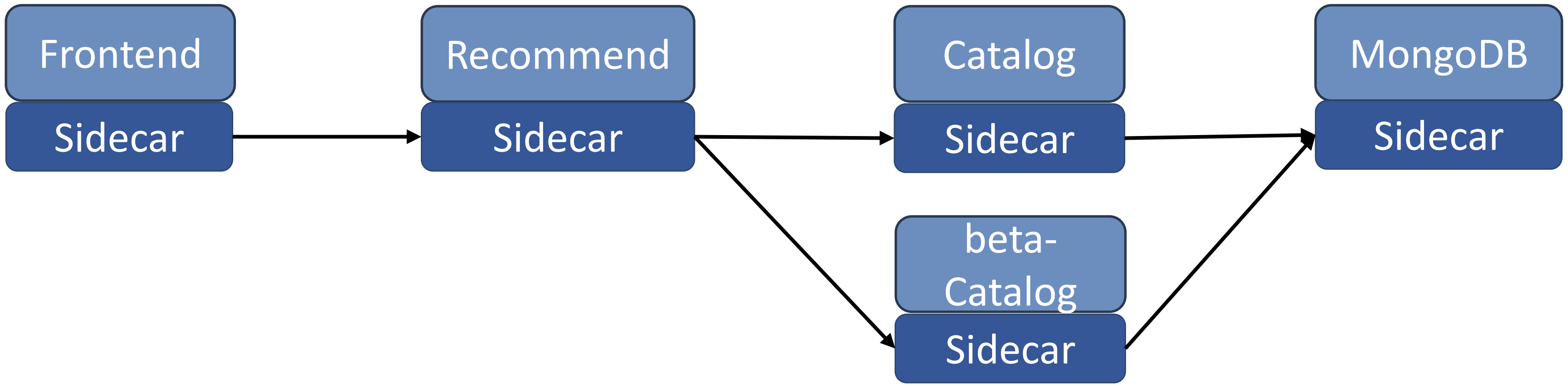
- Sidecars intercept all incoming and outgoing traffic from application containers.



Service Meshes for Policy Enforcement

Enforce policies inside a *sidecar* container, running *alongside* app containers.

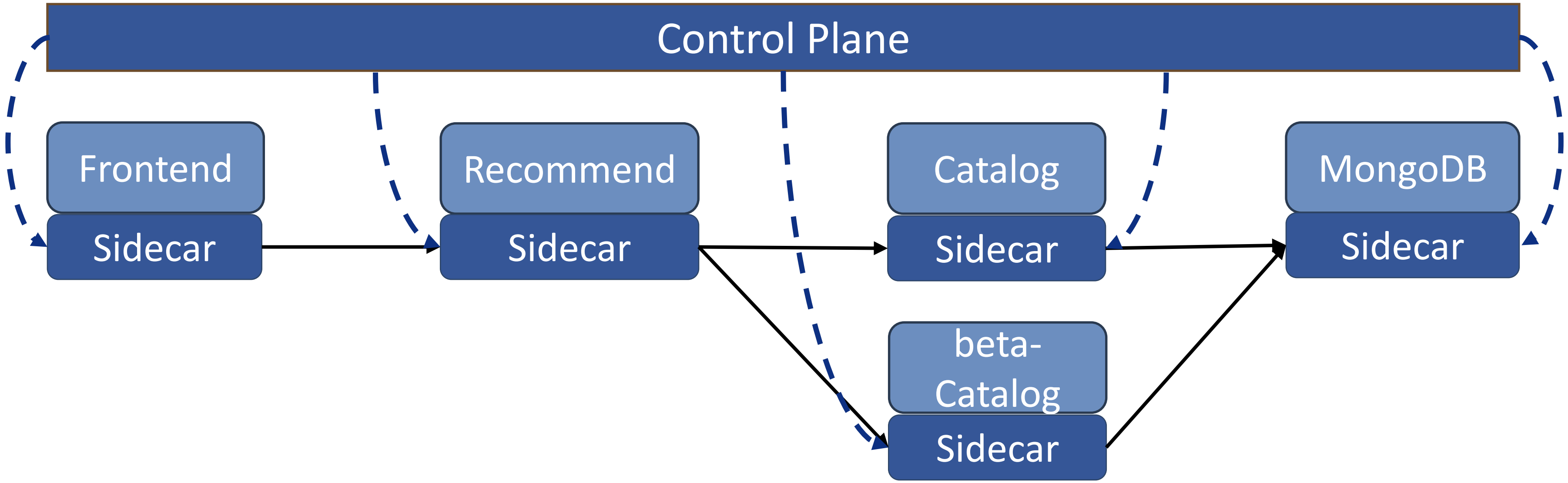
- Sidecars intercept all incoming and outgoing traffic from application containers.



Service Meshes for Policy Enforcement

Enforce policies inside a *sidecar* container, running *alongside* app containers.

- Sidecars intercept all incoming and outgoing traffic from application containers.
- Sidecars are configured by the service mesh control plane.



Variety of Service Mesh Offerings

Istio

Kong cilium

NGINX Service Mesh

Google Cloud Anthos linkerd

Several being used in production …

Variety of Service Mesh Offerings

Istio
Kong
cilium
NGINX Service Mesh
Google Cloud Anthos
linkerd

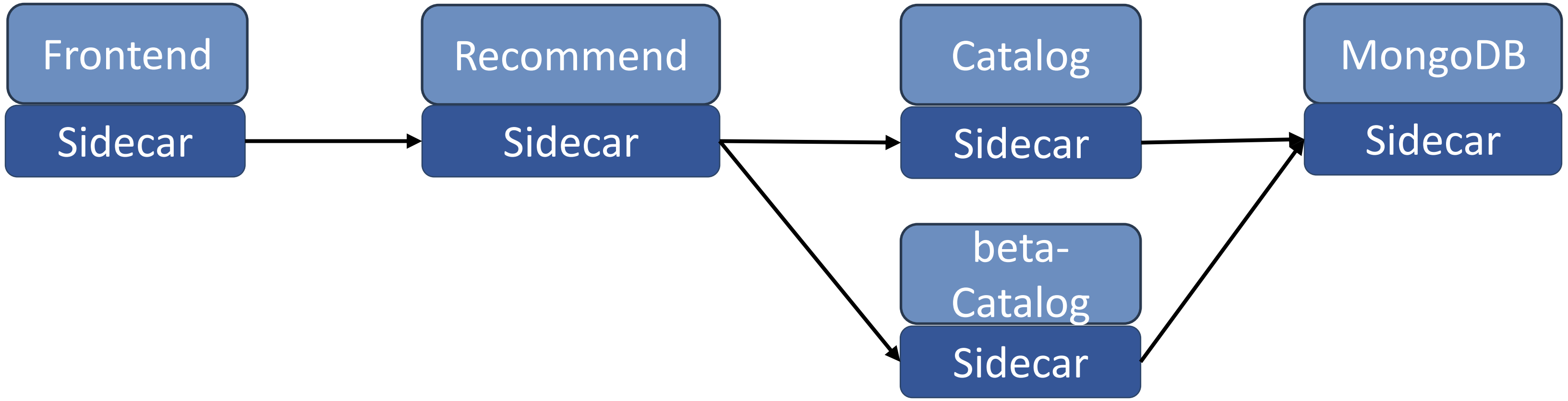
Several being used in production …

Aeraki Mesh
AWS App Mesh
HashiCorp Consul
EaseMesh
Flomesh Service Mesh
greymatter.io
KMESH
Kuma
Merbridge
Nasp
Sermant
Service Mesh Interface
SMP
slime
TSB
GLASNOSTIC
GLOO MESH
Open Service Mesh
OPENSERGO
træfik mesh

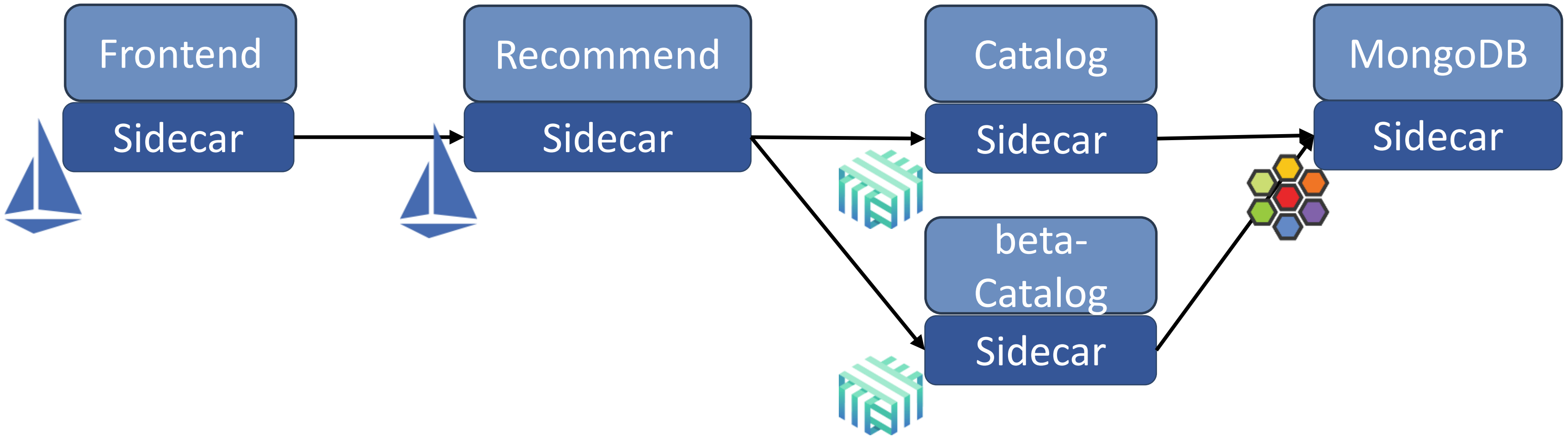
… and many more being actively developed!

The wide variety of service meshes allow different ***trade-offs between performance and ease of configurations.***

Service Meshes: Ideal Vision

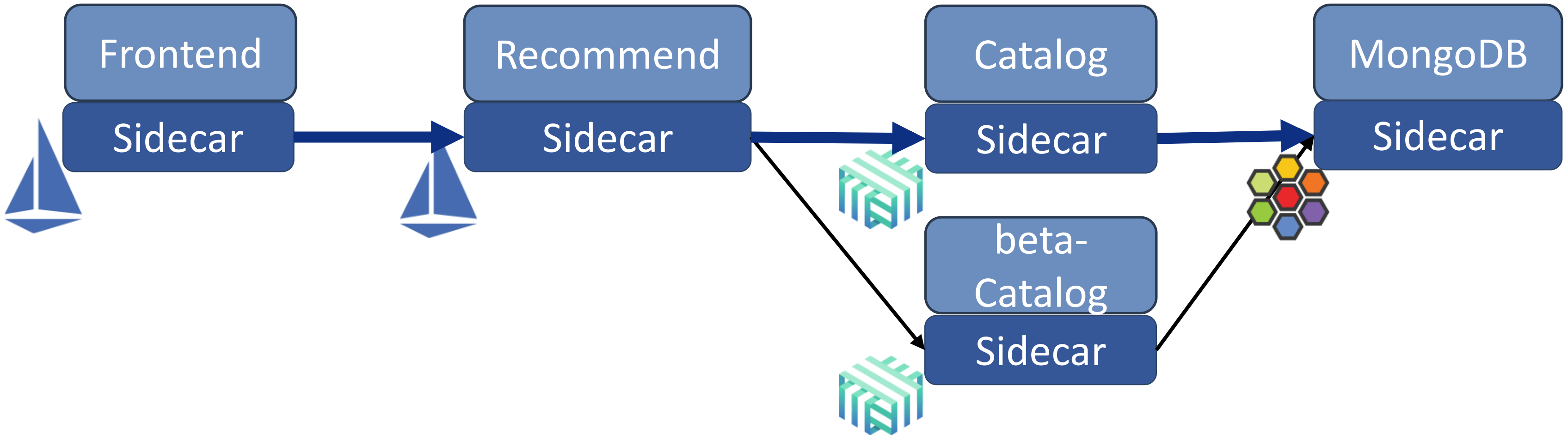


Service Meshes: Ideal Vision



*Exploit the trade-offs provided by **diverse data planes**.*

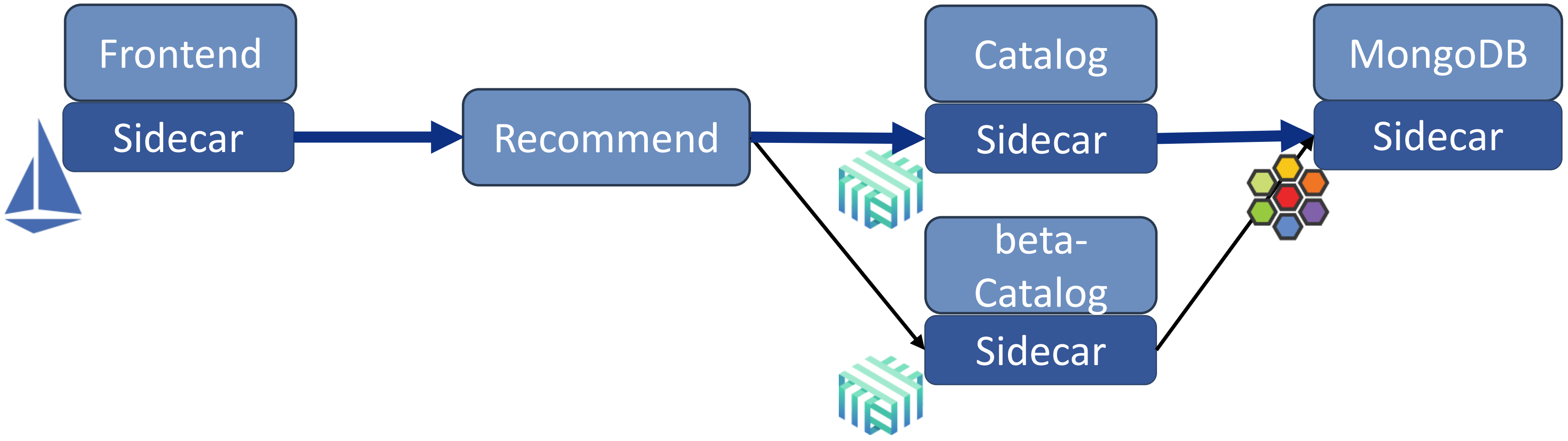
Service Meshes: Ideal Vision



*Exploit the trade-offs provided by **diverse data planes**.*

*Enable **rich policies** over microservice communication.*

Service Meshes: Ideal Vision

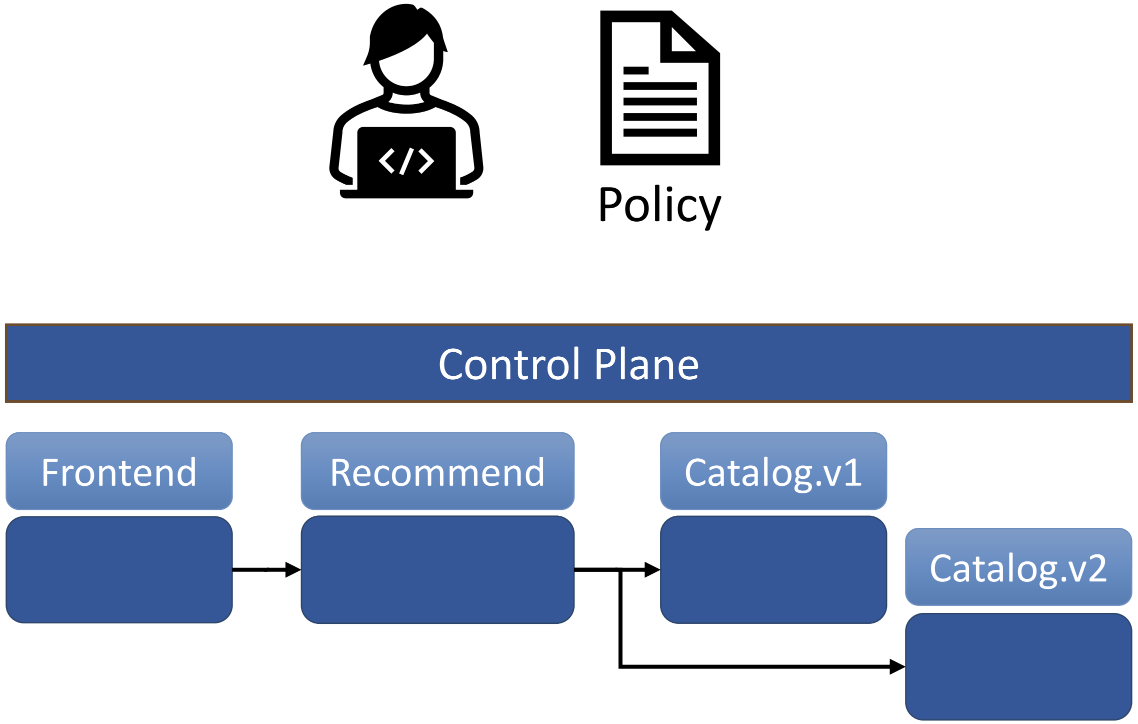


*Exploit the trade-offs provided by **diverse data planes**.*

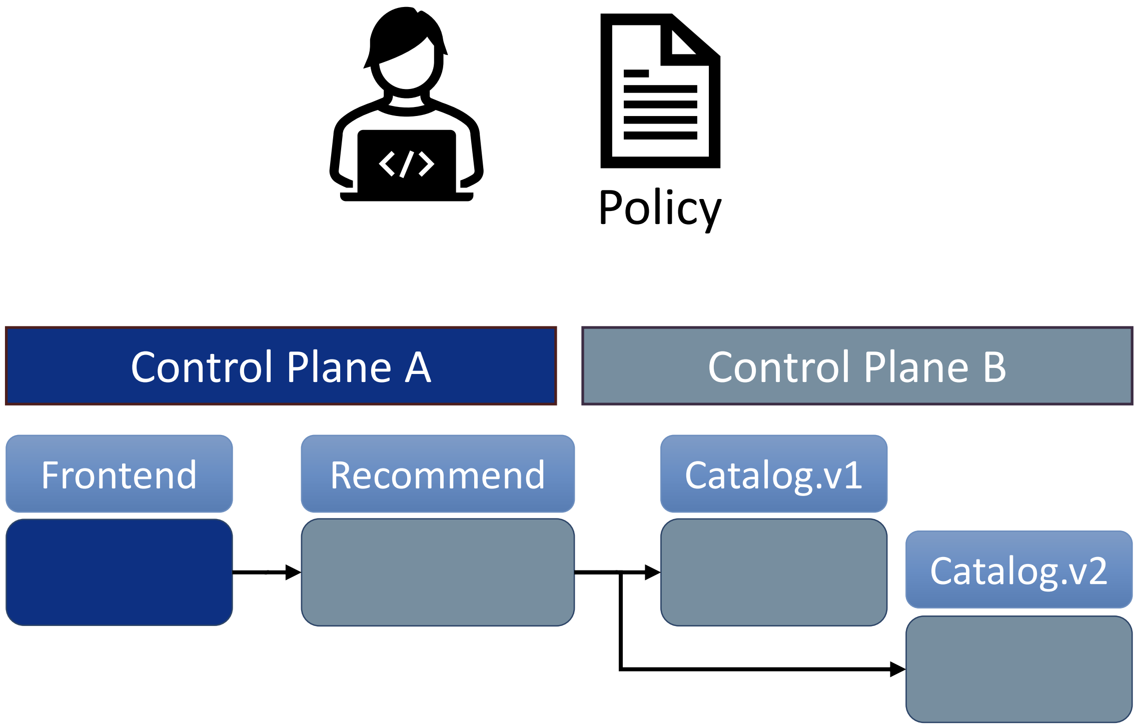
*Enable **rich policies** over microservice communication.*

*Enforce policies at **minimal performance overhead**.*

Service Meshes Today: Far from Ideal!



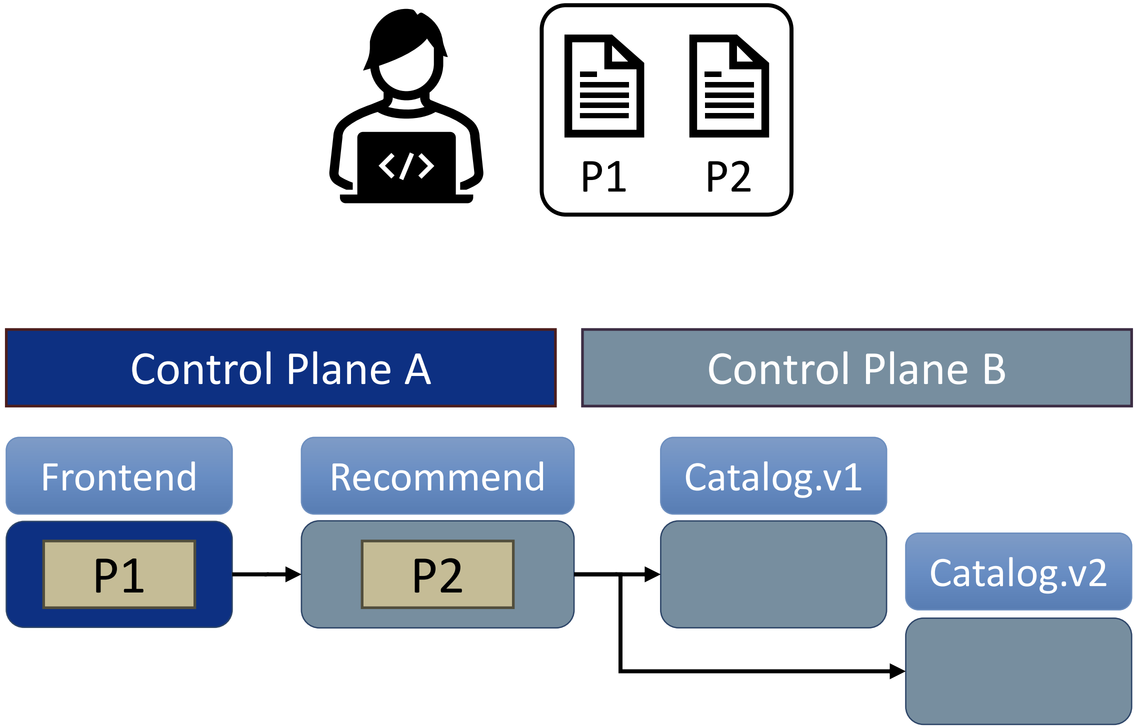
Service Meshes Today: Far from Ideal!



Use diverse dataplanes

! Tight coupling of control planes and dataplane implementations.

Service Meshes Today: Far from Ideal!



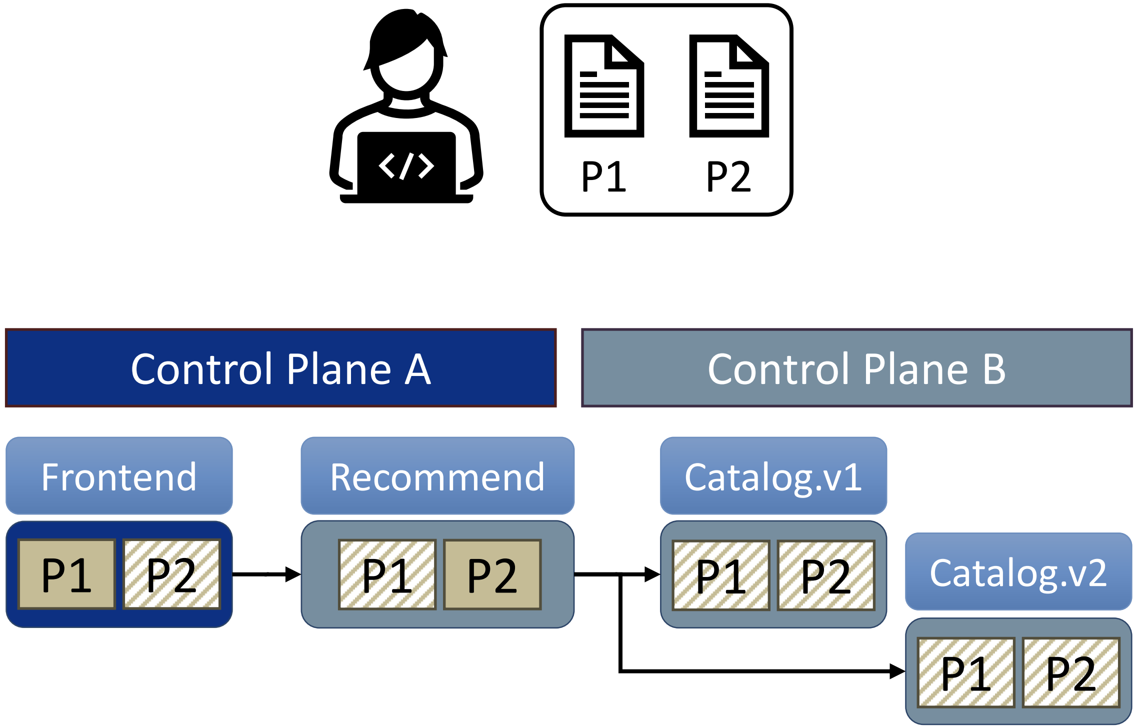
Use diverse dataplanes

! Tight coupling of control planes and dataplane implementations.

Specify rich policies

! Broken abstractions lead to tedious and error-prone policy specification.

Service Meshes Today: Far from Ideal!



Use diverse dataplanes

! Tight coupling of control planes and dataplane implementations.

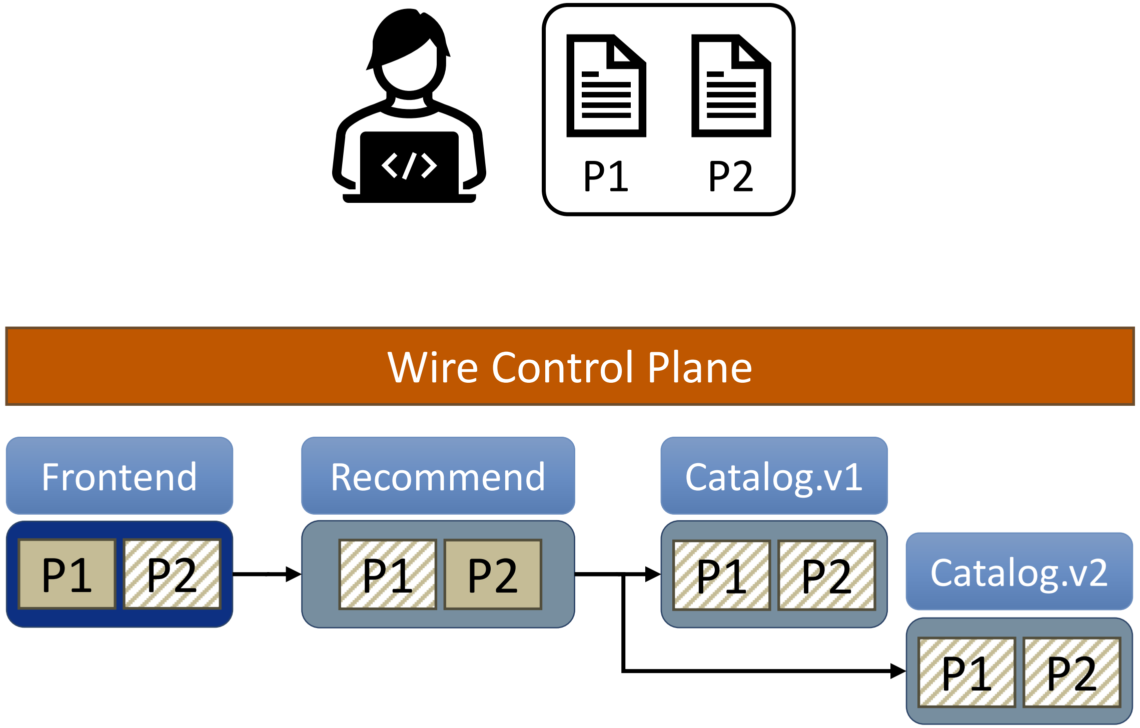
Specify rich policies

! Broken abstractions lead to tedious and error-prone policy specification.

Minimal overhead of mesh

! Control planes are application- and policy- unaware.

Our Proposal: *Copper and Wire*



Use diverse dataplanes

High-level abstraction for dataplane functionality, **ACTs**

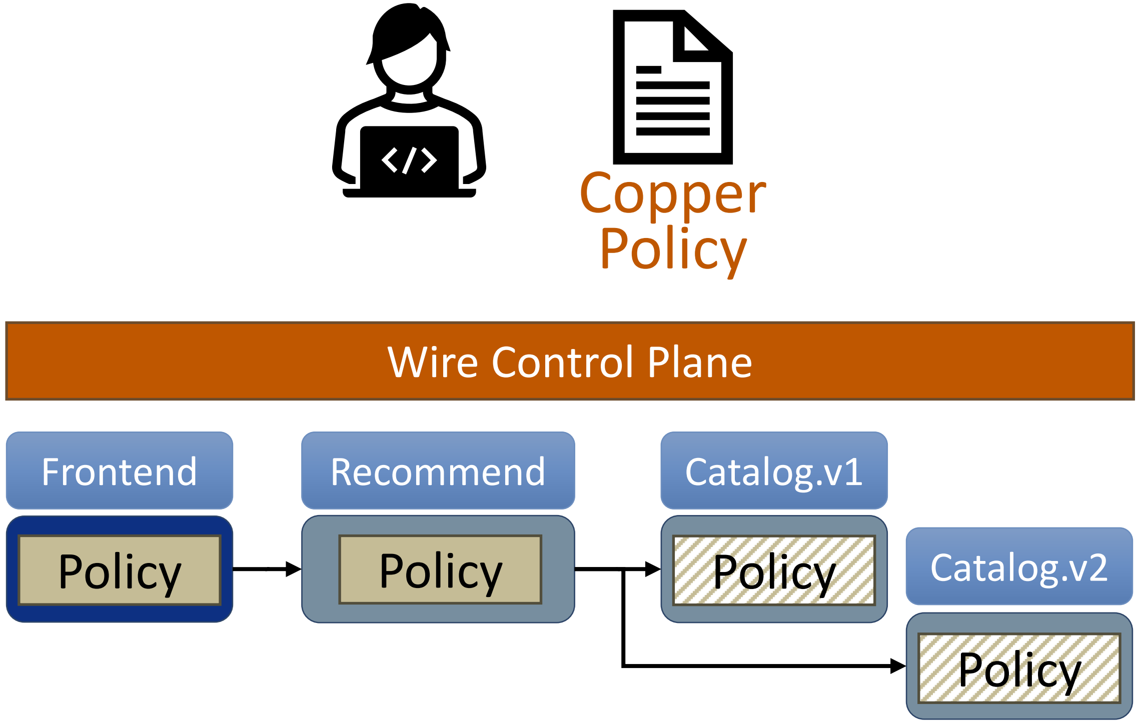
Specify rich policies

Broken abstractions lead to tedious and error-prone policy specification.

Minimal overhead of mesh

Control planes are application- and policy- unaware.

Our Proposal: *Copper and Wire*



Use diverse dataplanes

High-level abstraction for dataplane functionality, **ACTs**

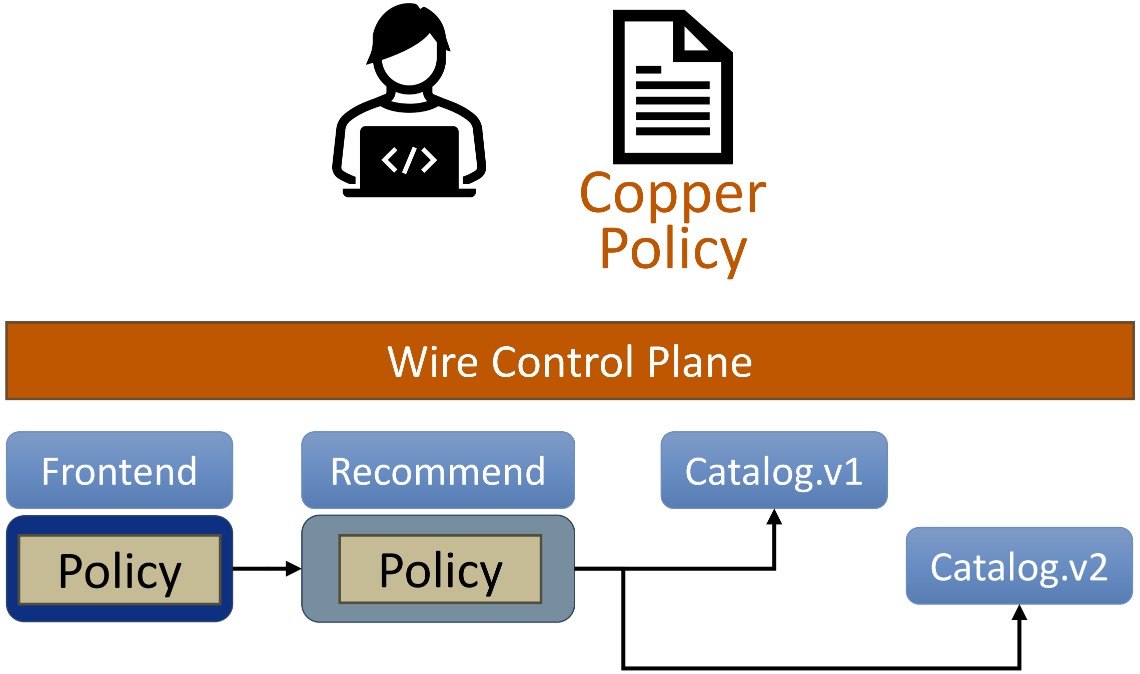
Specify rich policies

Specify policies over paths using **run-time contexts**

Minimal overhead of mesh

Control planes are application- and policy- unaware.

Our Proposal: *Copper and Wire*



Use diverse dataplanes

High-level abstraction for dataplane functionality, **ACTs**

Specify rich policies

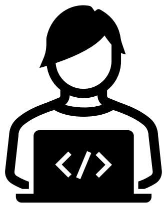
Specify policies over paths using **run-time contexts**

Minimal overhead of mesh

Eliminate redundant sidecars using clever optimizations.

Drawback: Dataplane Heterogeneity not Well-Supported

- To extract maximum performance, developers must use different dataplanes!

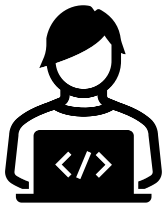


P1: ***Circuit breaking*** policy (at most 100 requests at a time)
P2: Drop requests with `free' header.

Dataplane	Features	Performance
Dataplane A	P1, P2	⌛
Dataplane B	P1	⚡⚡⚡

Drawback: Dataplane Heterogeneity not Well-Supported

- To extract maximum performance, developers must use different dataplanes.
- Problem: Same policy may require intricate configurations for each dataplane!



P1: ***Circuit breaking*** policy (at most 100 requests at a time)

P2: Drop requests with `free' header.

Dataplane A

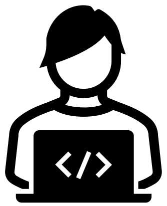
```
kind: DestinationRule
...
trafficPolicy:
 connectionPool:
  tcp:
   maxConnections: 1
  http:
   maxRequestsPerConnection: 100
```

Dataplane B

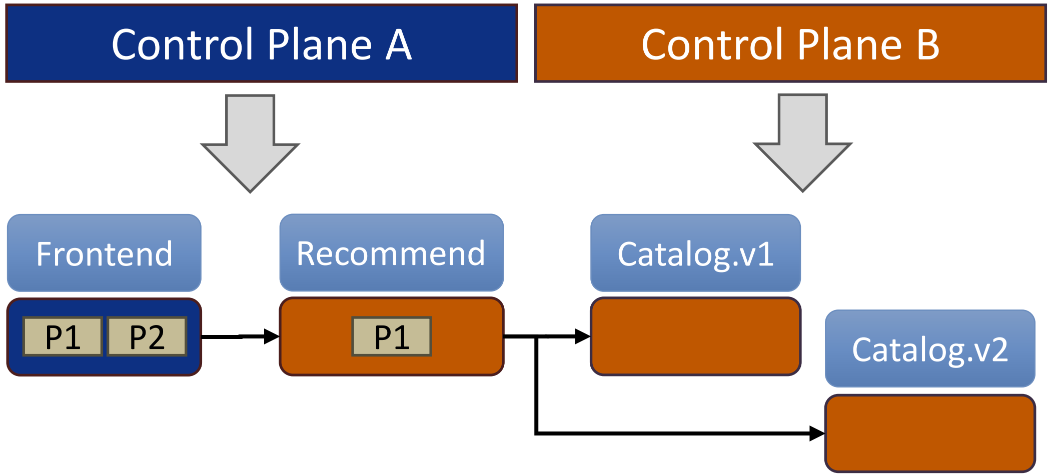
```
kind: CiliumClusterEnvoyConfig
...
circuit_breakers:
 thresholds:
 - priority: "DEFAULT"
   max_requests: 100
   max_pending_requests: 100
```

Drawback: Dataplane Heterogeneity not Well-Supported

- To extract maximum performance, developers must use different dataplanes.
- Problem: Same policy may require intricate configurations for each dataplane!
- Problem: Developers need to manually configure separate control planes!

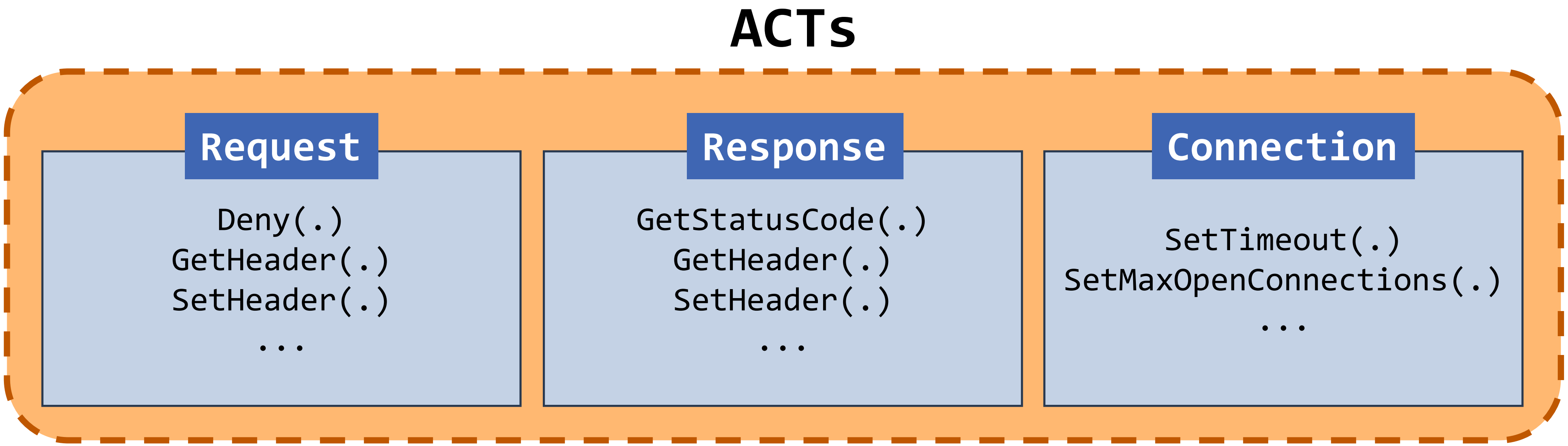


P1: ***Circuit breaking*** policy (at most 100 requests at a time)
P2: Drop requests with `free' header.



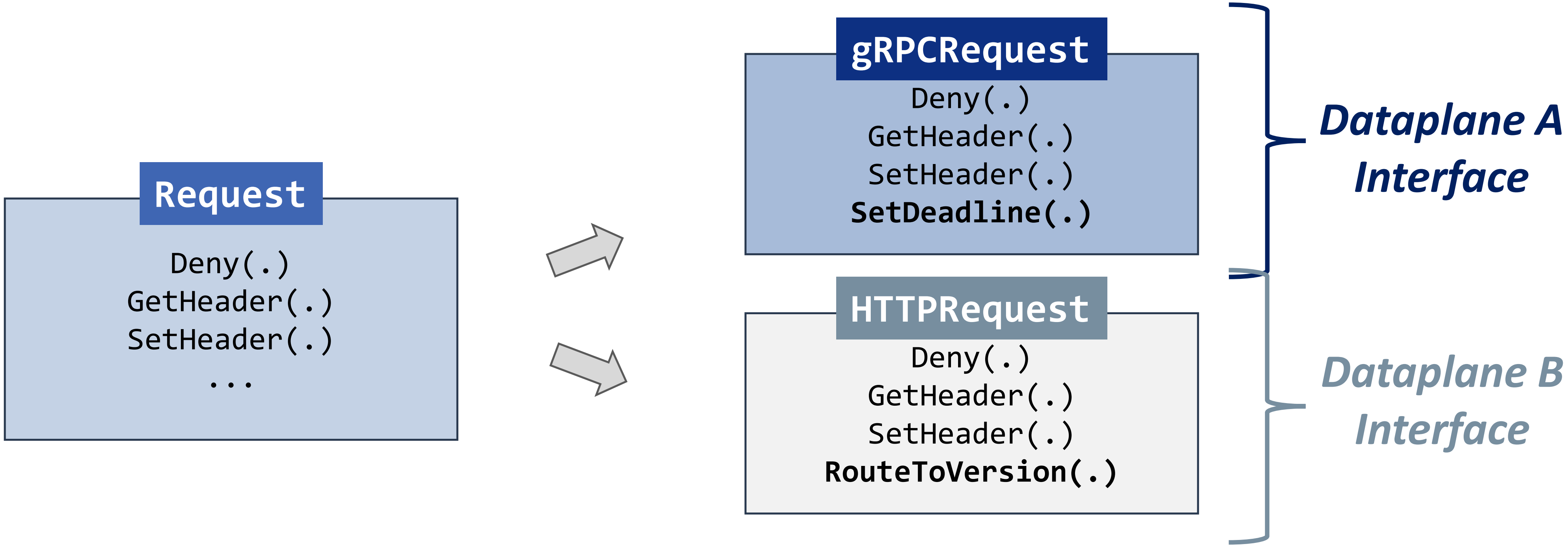
Idea: Abstract Communication Types (ACTs)

- Identify the common object used by ***all dataplanes and all policies.***
  - Elevate as a first-class citizen in programming (OpenFlow-inspired)
- Use standard polymorphism and OOP to represent dataplane functionality.



ACTs ⇒ Dataplane Interfaces

- ACTs can be derived to express dataplane functionality.
  - `Request` ⇒ `HTTPRequest`, `gRPCRequest`, etc.
  - `Connection` ⇒ `TCPConnection`, `HTTPConnection`, etc.



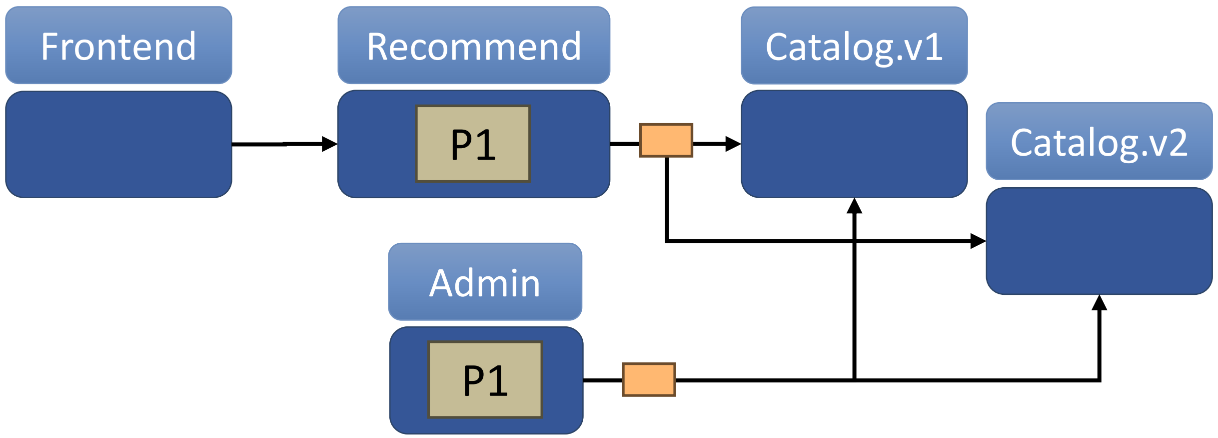
Drawback: Specifying Rich Policies is Hard

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

Policy P1

```
hosts:
- catalog
http:
- route:
  - destination:
      host: catalog
      subset: v2
    weight: 50
  - destination:
      host: catalog
      subset: v1
    weight: 50
```

Destination Service



But the given policy must only execute over requests from `Frontend`!

Drawback: Specifying Rich Policies is Hard

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

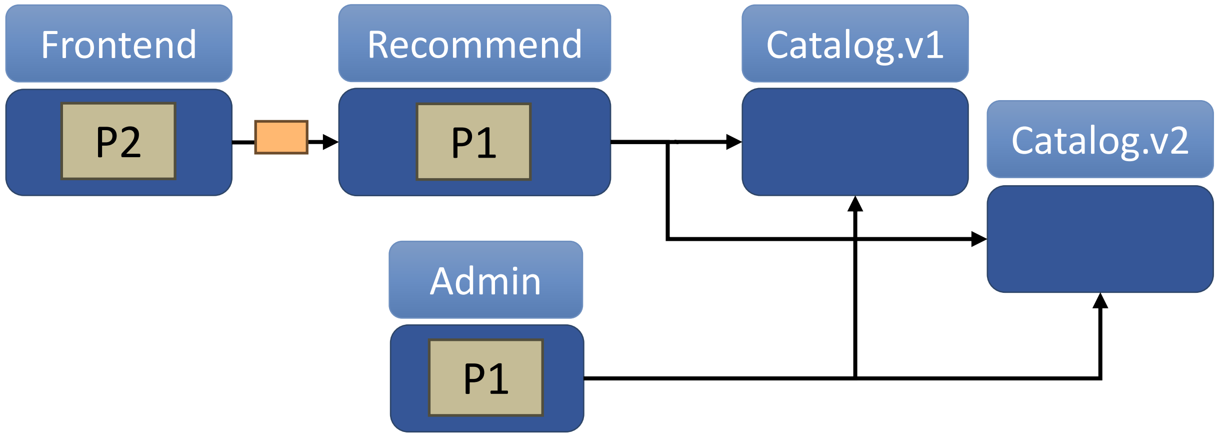
Policy P1

```
hosts:
- catalog
http:
- match:
  - headers:
      fromFE:
        exact: true
- route:
  - destination:
      host: catalog
      subset: v2
    weight: 50
  - destination:
      host: catalog
      subset: v1
    weight: 50
```

Policy P2

```
hosts:
- recommend
http:
- headers:
    request:
      add:
        fromFE: true
```

Add `fromFE` header to only the requests to Recommend service.



Drawback: Specifying Rich Policies is Hard

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

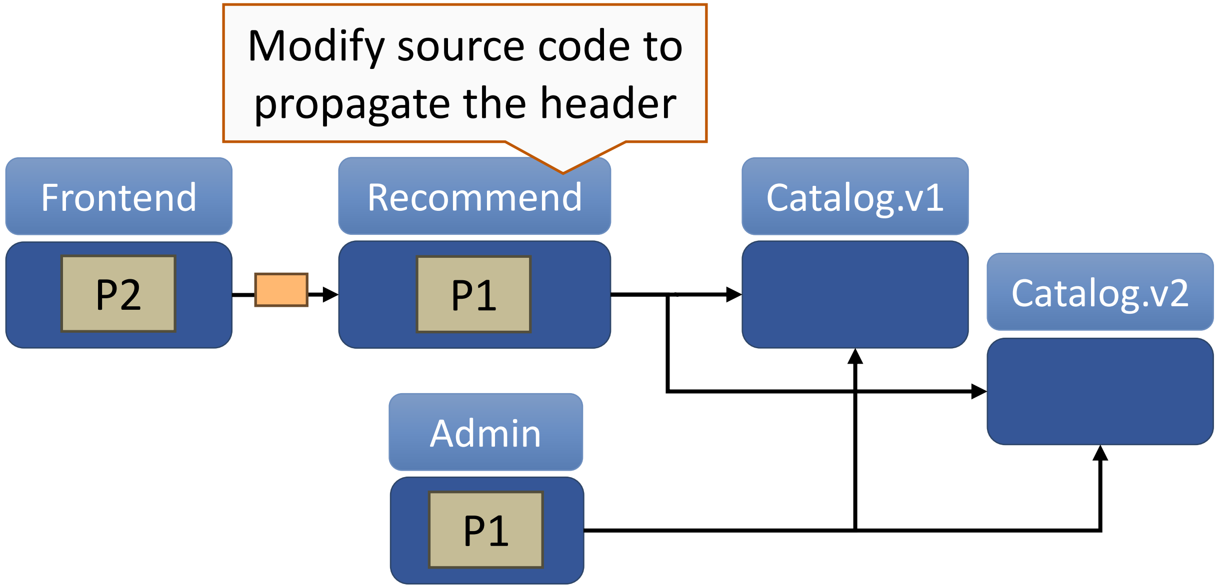
Policy P1

```
hosts:
- catalog
http:
- match:
  - headers:
      fromFE:
        exact: true
- route:
  - destination:
      host: catalog
      subset: v2
    weight: 50
  - destination:
      host: catalog
      subset: v1
    weight: 50
```

Policy P2

```
hosts:
- recommend
http:
- headers:
    request:
      add:
        fromFE: true
```

Add `fromFE` header to only the requests to Recommend service.



Drawback: Specifying Rich Policies is Hard

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

Policy P1

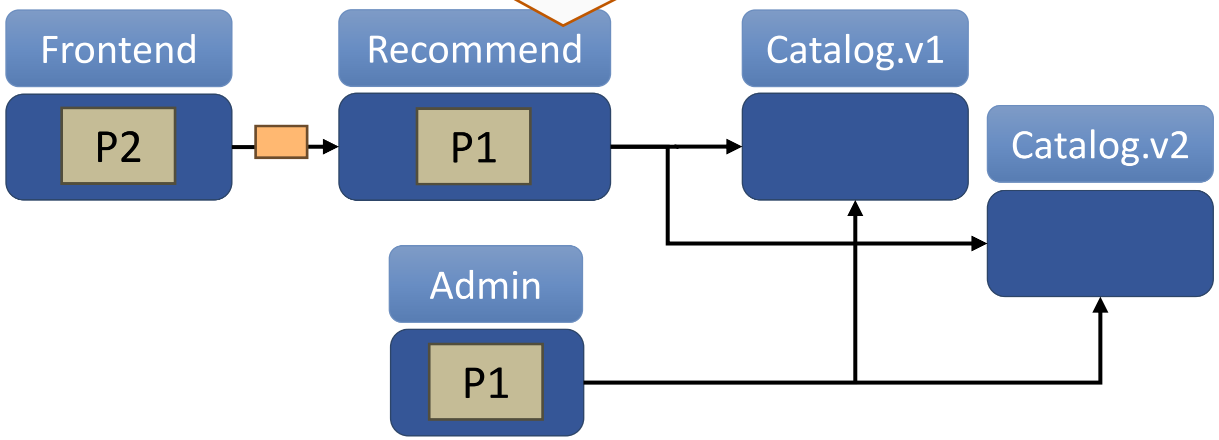
```
hosts:
- catalog
http:
- match:
  - headers:
      fromFE:
        exact: true
- route:
  - destination:
      host: catalog
      subset: v2
    weight: 50
  - destination:
      host: catalog
      subset: v1
    weight: 50
```

Policy P2

```
hosts:
- recommend
http:
- headers:
    request:
      add:
        fromFE: true
```

Add `fromFE` header to only the requests to Recommend service.

Modify source code to propagate the header

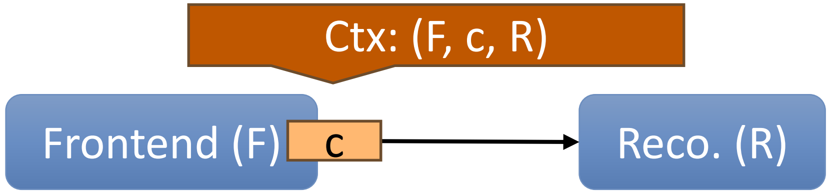


Complicates policy specification as developers need to manually "break-down" the policies!
Makes microservice modifications challenging as policies only work for specific application graphs!

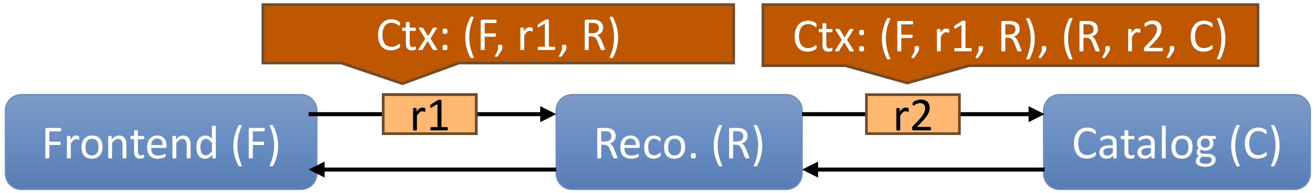
Run-time Contexts

- Copper policies are specified over concrete instantiations of ACTs, each associated with a ***run-time context***.
- The run-time context carries the history of events leading to an object.

Connection type objects



Request/Response type objects



Policy Expression over Context Patterns

Context patterns = regular expressions of context strings.

- Policy specification is independent of intermediate services.
- Multiple request paths can be expressed under a single policy.

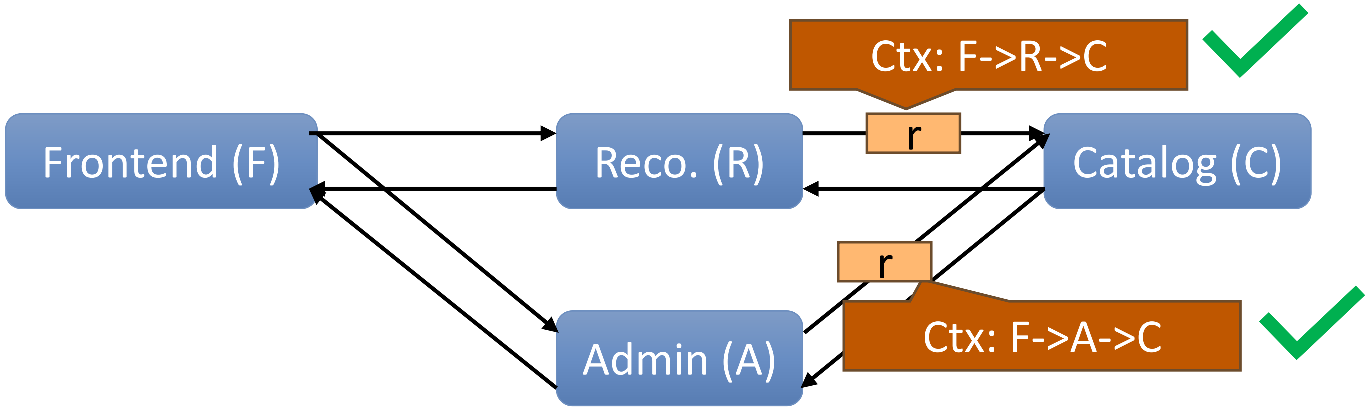
*Apply policy over context pattern: “Frontend.*Catalog”*

Policy Expression over Context Patterns

Context patterns = regular expressions of context strings.

- Policy specification is independent of intermediate services.
- Multiple request paths can be expressed under a single policy.

*Apply policy over context pattern: "Frontend.*Catalog"*



Copper Policy Programs

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

Copper Policy Program

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  using (FloatState sampler)
  context ("Frontend.*Catalog")
) {
  GetRandomSample(sampler);
  if (IsLessThan(sampler, 0.5)) {
    RouteToVersion(req, 'Catalog', 'v1');
  } else {
    RouteToVersion(req, 'Catalog', 'v2');
 }
}
```

Copper Policy Programs

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

Copper Policy Program

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  using (FloatState sampler)
  context ("Frontend.*Catalog")
) {
  GetRandomSample(sampler);
  if (IsLessThan(sampler, 0.5)) {
    RouteToVersion(req, 'Catalog', 'v1');
  } else {
    RouteToVersion(req, 'Catalog', 'v2');
 }
}
```

Dataplane Interface

```
import common.cui;
state FloatState{
  action GetRandomSample(self),
  action IsLessThan(self, float value),
}
act RPCRequest: Request{
  action Deny(self),
  action GetHeader(self, string header),
  action SetHeader(self, string header,
                   string value),
  action RouteToVersion(self, string service,
                   string label),
}
```

Copper Policy Programs

Policy: Distribute requests from `Frontend` to the two versions of `Catalog` in 50:50 ratio

Specify context pattern

Copper Policy Program

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  using (FloatState sampler)
  context ("Frontend.*Catalog")
) {
  GetRandomSample(sampler);
  if (IsLessThan(sampler, 0.5)) {
    RouteToVersion(req, 'Catalog', 'v1');
  } else {
    RouteToVersion(req, 'Catalog', 'v2');
 }
}
```

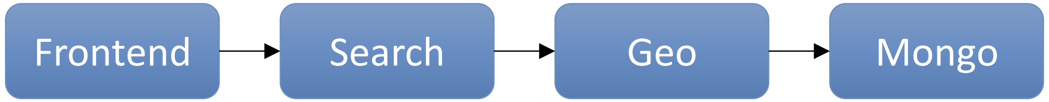
Dataplane Interface

```
import common.cui;
state FloatState{
  action GetRandomSample(self),
  action IsLessThan(self, float value),
}
act RPCRequest: Request{
  action Deny(self),
  action GetHeader(self, string header),
  action SetHeader(self, string header,
                   string value),
  action RouteToVersion(self, string service,
                        string label),
}
```

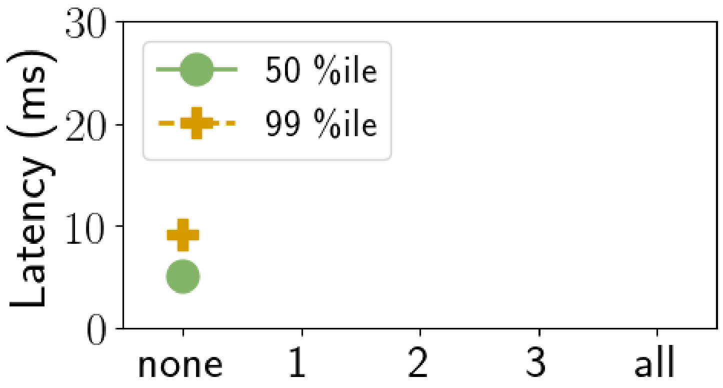
Specify policy logic, using conditionals and supported states.

Drawback: Sidecars Impose Overheads

- L7 processing imposes latency overheads



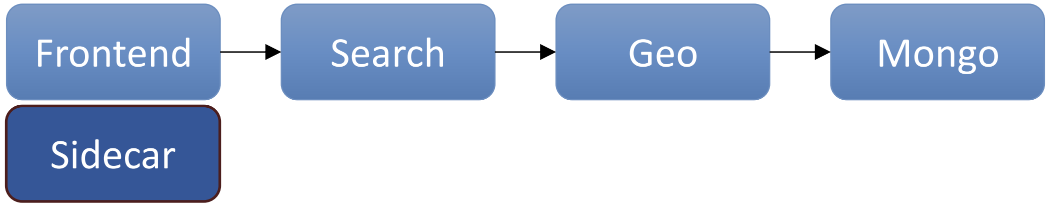
(Microservice chain from HotelReservation[1] benchmark)



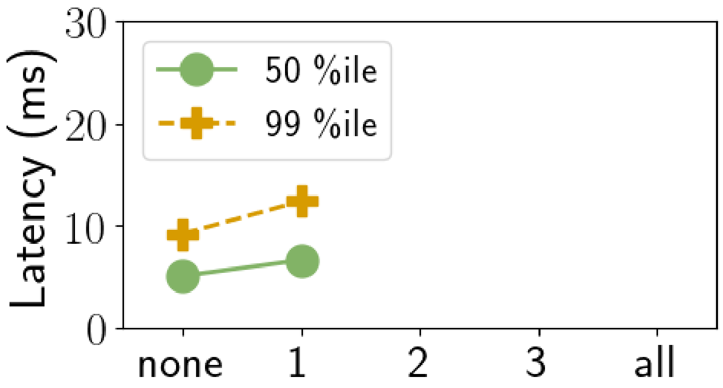
[1] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et. al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.

Drawback: Sidecars Impose Overheads

- L7 processing imposes latency overheads



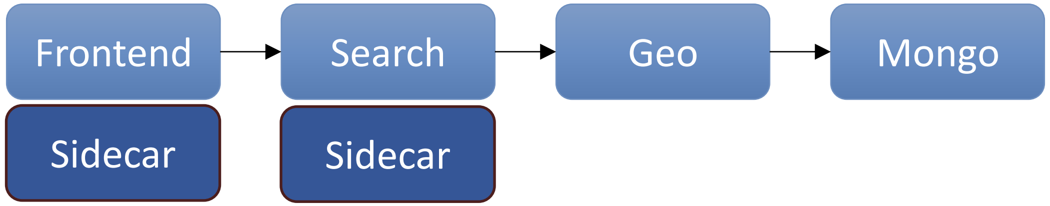
(Microservice chain from HotelReservation[1] benchmark)



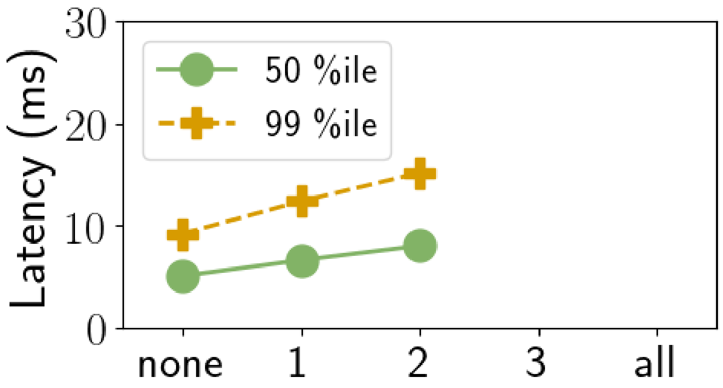
[1] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et. al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.

Drawback: Sidecars Impose Overheads

- L7 processing imposes latency overheads



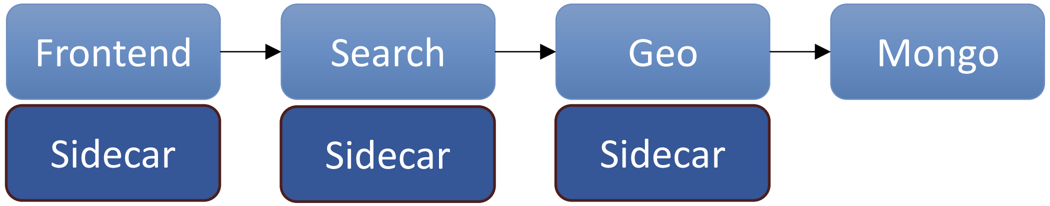
(Microservice chain from HotelReservation[1] benchmark)



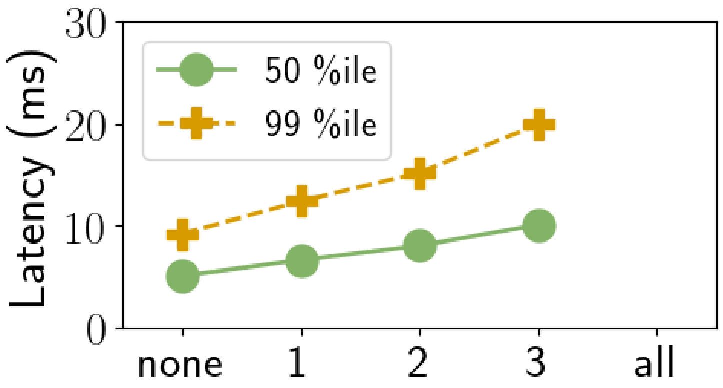
[1] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et. al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.

Drawback: Sidecars Impose Overheads

- L7 processing imposes latency overheads



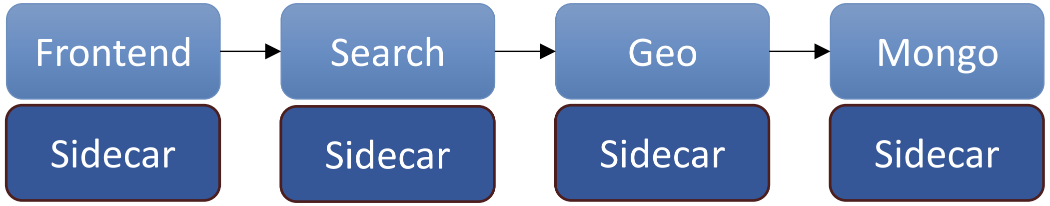
(Microservice chain from HotelReservation[1] benchmark)



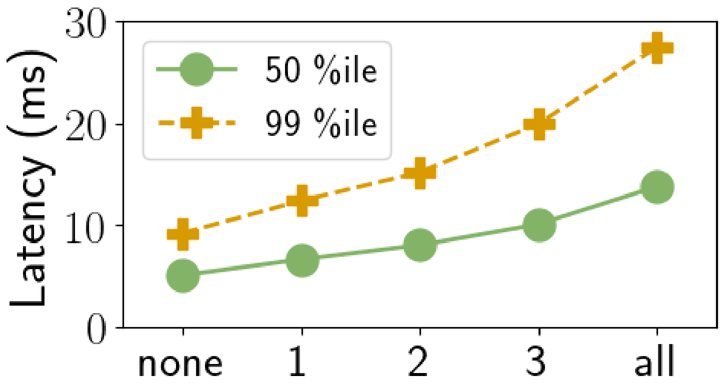
[1] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et. al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.

Drawback: Sidecars Impose Overheads

- L7 processing imposes latency overheads



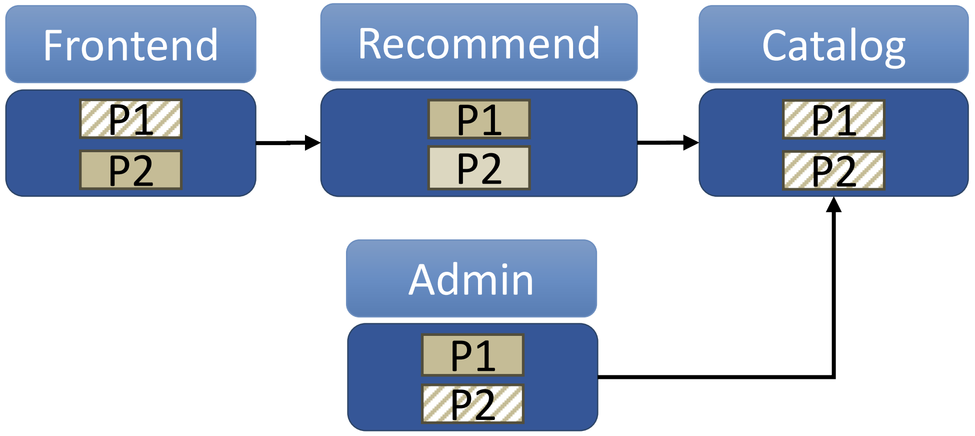
(Microservice chain from HotelReservation[1] benchmark)



[1] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et. al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.

Pruning Redundant Sidecars

Prune redundant sidecars using:



Example policies shown above

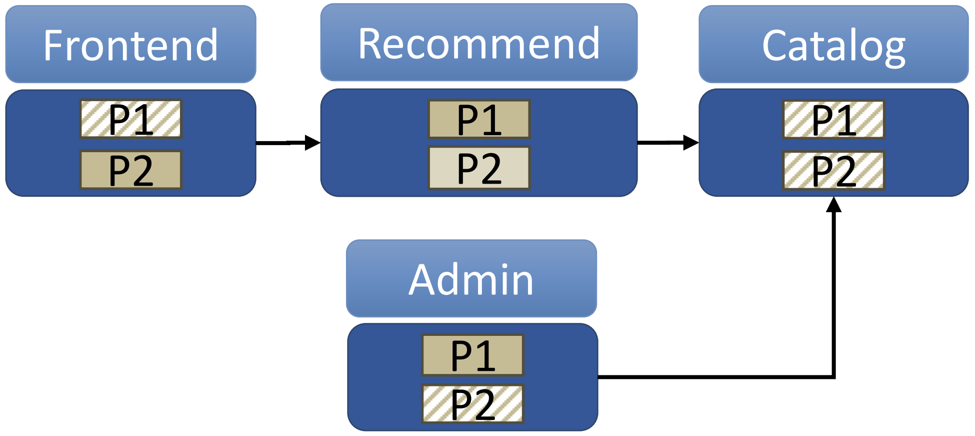
P1: ***Set a deadline*** for requests to Catalog

P2: ***Attach a `region' header*** to requests from Frontend to Recommend

Pruning Redundant Sidecars

Prune redundant sidecars using:

- Application graphs: service communication graph



Example policies shown above

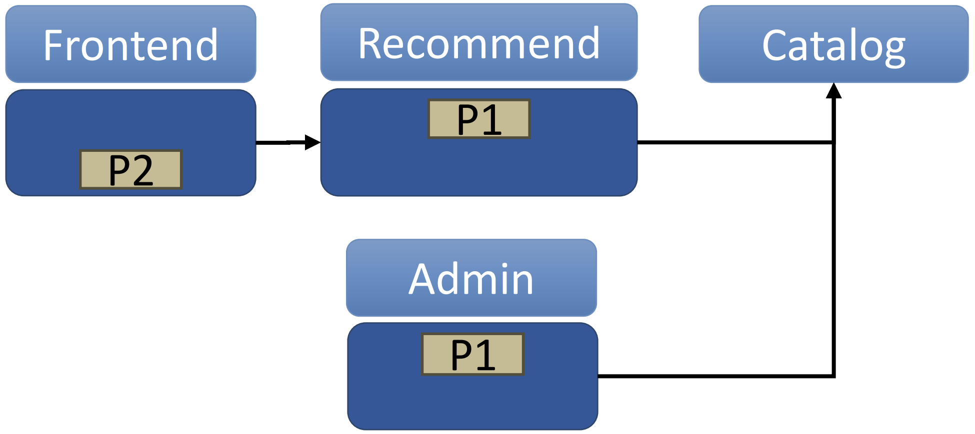
P1: ***Set a deadline*** for requests to Catalog

P2: ***Attach a `region' header*** to requests from Frontend to Recommend

Pruning Redundant Sidecars

Prune redundant sidecars using:

- Application graphs: service communication graph



Example policies shown above

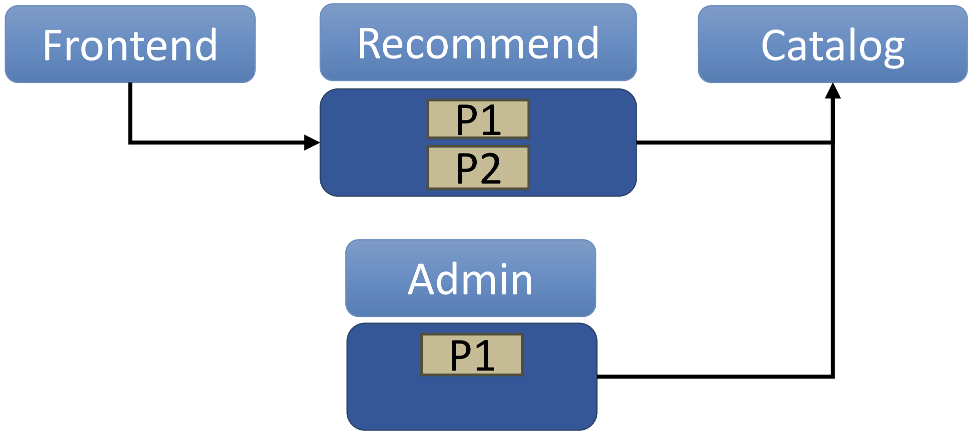
P1: ***Set a deadline*** for requests to Catalog

P2: ***Attach a `region' header*** to requests from Frontend to Recommend

Pruning Redundant Sidecars

Prune redundant sidecars using:

- Application graphs: service communication graph
- Policy semantics: where a policy can be correctly executed
    - For example, P2 can be enforced at the sidecar of either `Frontend` or `Recommend`



Example policies shown above

P1: ***Set a deadline*** for requests to Catalog

P2: ***Attach a `region' header*** to requests from Frontend to Recommend

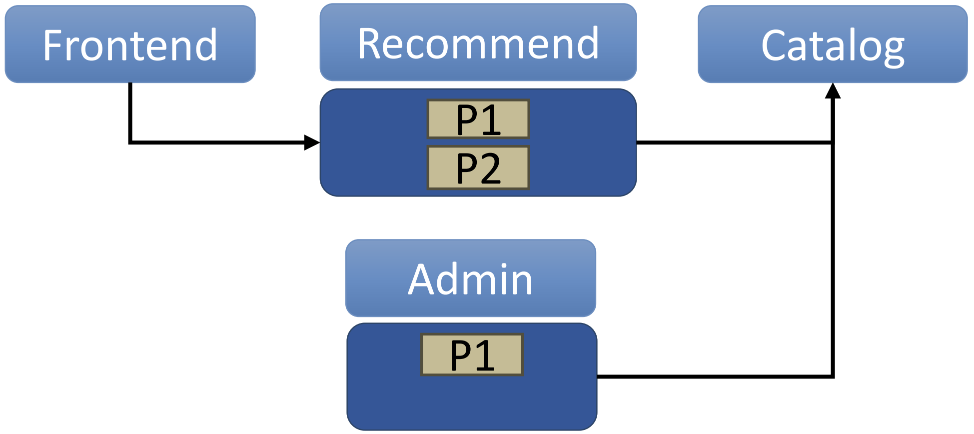
Pruning Redundant Sidecars

Prune redundant sidecars using:

- Application graphs: service communication graph
- Policy semantics: where a policy can be correctly executed
    - For example, P2 can be enforced at the sidecar of either `Frontend` or `Recommend`

```
[Egress]
action RouteToVersion(self, string service, string label)
```

Use ***Action Annotations*** in dataplane interfaces to extract policy execution semantics.



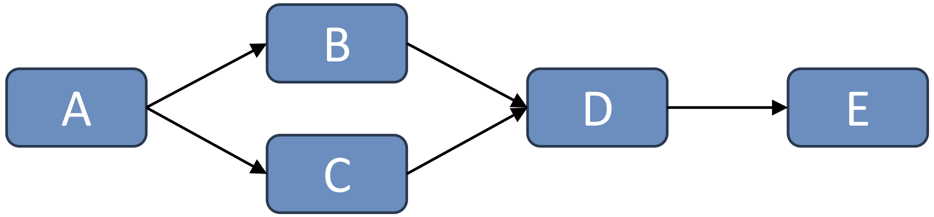
Example policies shown above

P1: ***Set a deadline*** for requests to Catalog

P2: ***Attach a `region' header*** to requests from Frontend to Recommend

Dataplane Optimization by Wire: Overview

Context patterns from user policies



*"A.*DE" can only be checked at D or E!*

- A policy can correctly execute at the terminal pair of the context pattern.

ACTs and Action Annotations from Interfaces

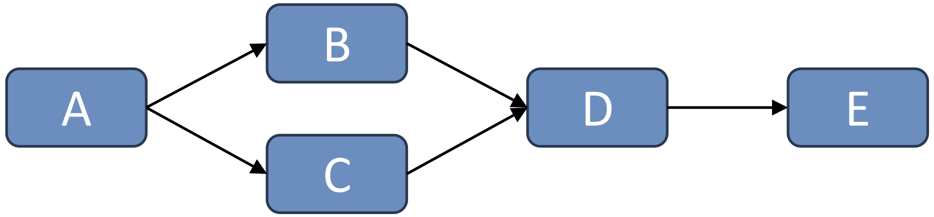
```
// Dataplane A interface
action GetHeader(self, string header)

// Dataplane B interface
action GetHeader(self, string header)
action SetDeadline(self, float deadline)
```

- A dataplane can only support the functions listed in its interface.

Dataplane Optimization by Wire: Overview

Context patterns from user policies



*"A.*DE" can only be checked at D or E!*

- A policy can correctly execute at the terminal pair of the context pattern.

ACTs and Action Annotations from Interfaces

```
// Dataplane A interface
action GetHeader(self, string header)

// Dataplane B interface
action GetHeader(self, string header)
action SetDeadline(self, float deadline)
```

- A dataplane can only support the functions listed in its interface.

Model these as constraints to an SMT solver!

Evaluation Questions

- Does Copper help enable simple and expressive mesh policies relative to today's approaches?
- How beneficial is Wire for real-world applications in lowering dataplane overhead?
- Does Wire help in enabling the effective use of multiple dataplanes compared to today's best approaches?
- What is the scalability of the Wire control plane?
- What are the overheads of using the eBPF add-on?

Evaluation Questions

In this talk

- Does Copper help enable simple and expressive mesh policies relative to today’s approaches?
- How beneficial is Wire for real-world applications in lowering dataplane overhead?
- Does Wire help in enabling the effective use of multiple dataplanes compared to today’s best approaches?
- What is the scalability of the Wire control plane?
- What are the overheads of using the eBPF add-on?

Does *Copper* Simplify Policy Expression?

Against Istio, Copper policies are significantly smaller and easier to write.

	Policy Description	App	Policy Lines of Code	
			Istio (App source changes)	Copper
Header Manipulation	**P1:** Set 'display' header 'true' for all requests to `catalog` originating from `frontend`.	Online Boutique	54 (8 lines in 2 services)	8 (**6.75×**)
	P1: Set the 'critical' header to 'true' for all requests to `geo` and `rate` originating from `frontend`.	Hotel Reservation	37 (4 lines in 1 service)	8 (**4.63×**)
	P1: Set 'write' header 'true' for all requests to `post-storage` originating from `compose-post`.	Social Network	54 (8 lines in 2 services)	8 (**6.75×**)
Traffic Management	**P2:** Route to v2 of a service if request is from `checkout`; v1 if from `frontend`	Online Boutique	101 (4 lines in 1 service)	36 (**2.8×**)
	P2: Route to v2 of a service if request is from `search`; v1 if from `frontend`	Hotel Reservation	59 (4 lines in 1 service)	18 (**3.28×**)
	P2: Route to v2 of a service if request is from `compose-post`; v1 if from `frontend`	Social Network	80 (12 lines in 3 services)	27 (**2.96×**)
Access Control	**P3:** Restrict access to database services	Online Boutique	24	9 (**2.6×**)
	P3: Restrict access to database services	Hotel Reservation	99	57 (**1.7×**)
	P3: Restrict access to database services	Social Network	99	60 (**1.65×**)
Rate Limiting	**P4:** Rate limit requests from `frontend` to `catalog`	Online Boutique	92 (8 lines in 2 services)	16 (**5.75×**)

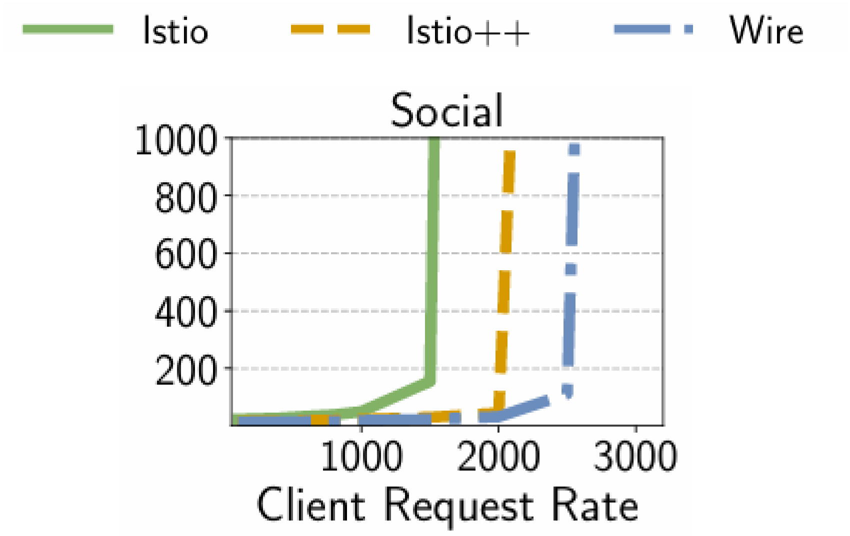
Can *Wire* Lower Dataplane Overheads?

- Comparison controllers:
  - **Istio**: Default control plane
  - **Istio++**: Default control plane + knowledge of application graph to prune sidecars. *[The best developers can get today via significant manual effort.]*
  - ***Wire***: Uses application graph + policy semantics to optimize the data plane.
- Testbed: 80-core Cloudlab cluster, consisting of 4 nodes each with 20-core Xeon CPU@2.40GHz and 64GB RAM

Can *Wire* Lower Dataplane Overheads?

Enforced policy: Header manipulation rules for a set of contexts

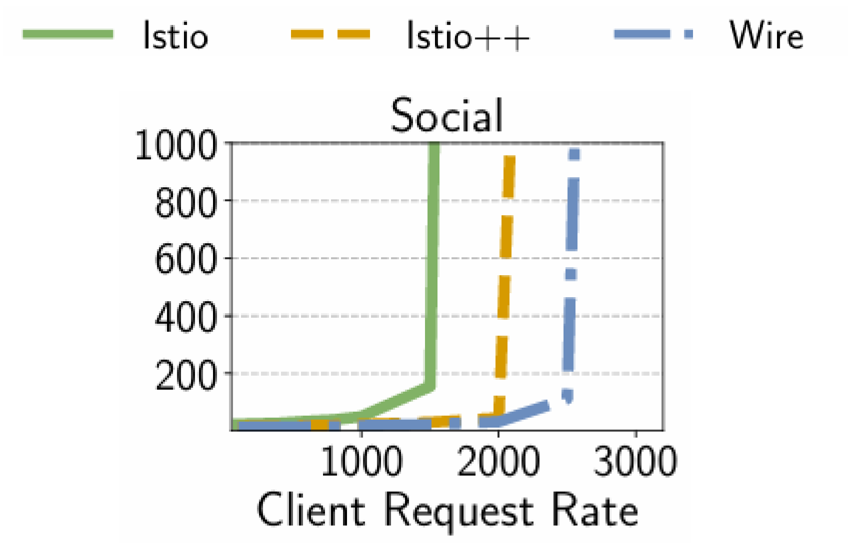
By systematically reducing sidecars, Wire's configured dataplane can sustain higher throughput.

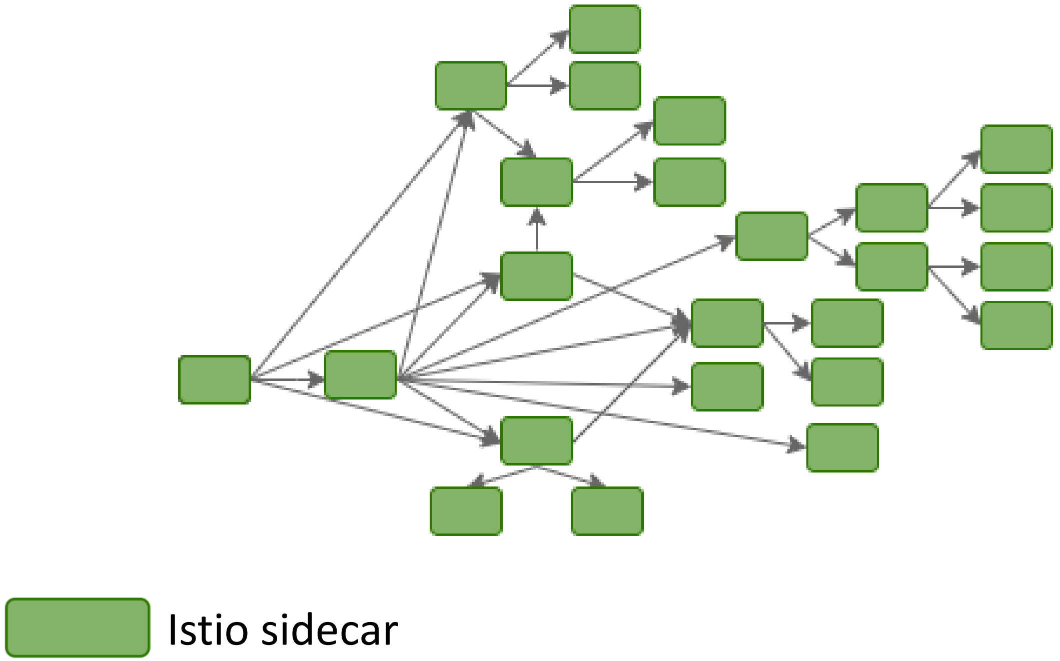


Can *Wire* Lower Dataplane Overheads?

Enforced policy: Header manipulation rules for a set of contexts

By systematically reducing sidecars, Wire's configured dataplane can sustain higher throughput.

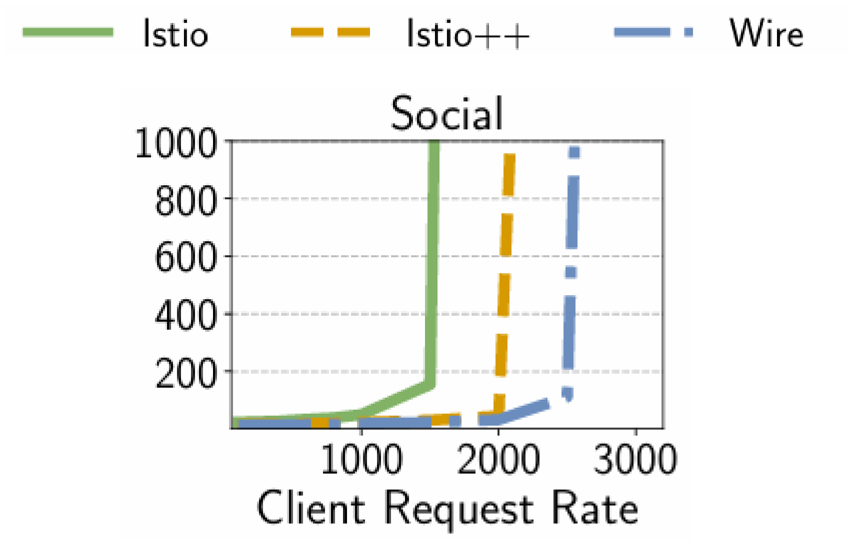


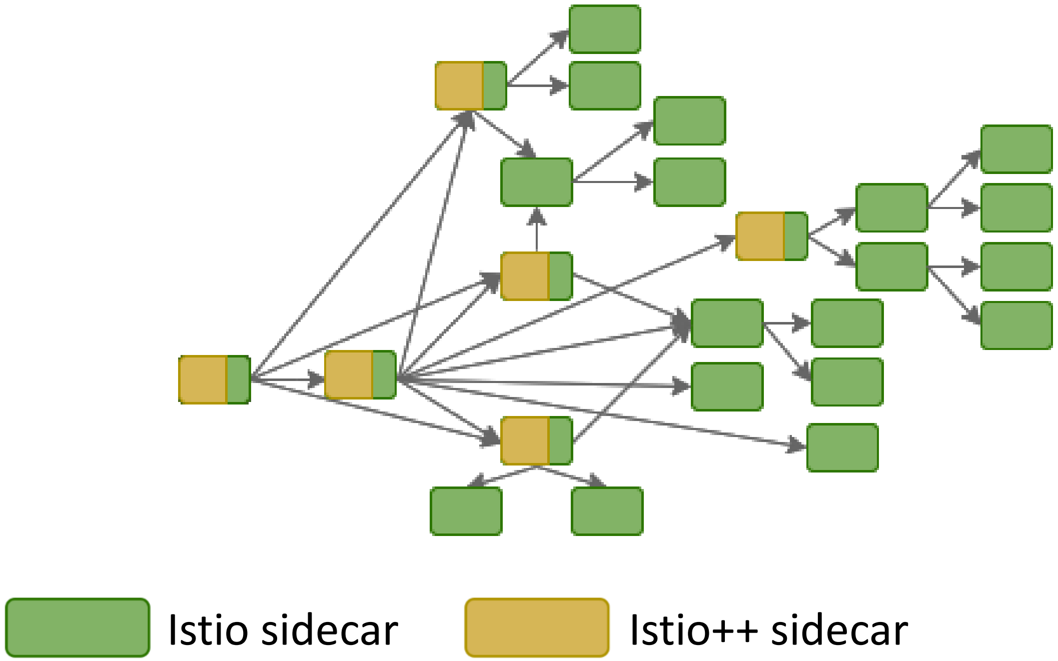


Can *Wire* Lower Dataplane Overheads?

Enforced policy: Header manipulation rules for a set of contexts

By systematically reducing sidecars, Wire's configured dataplane can sustain higher throughput.

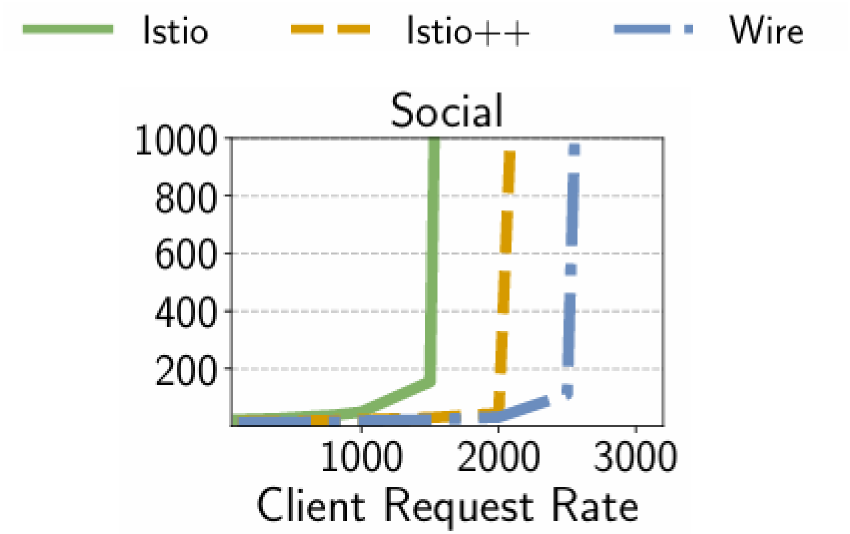


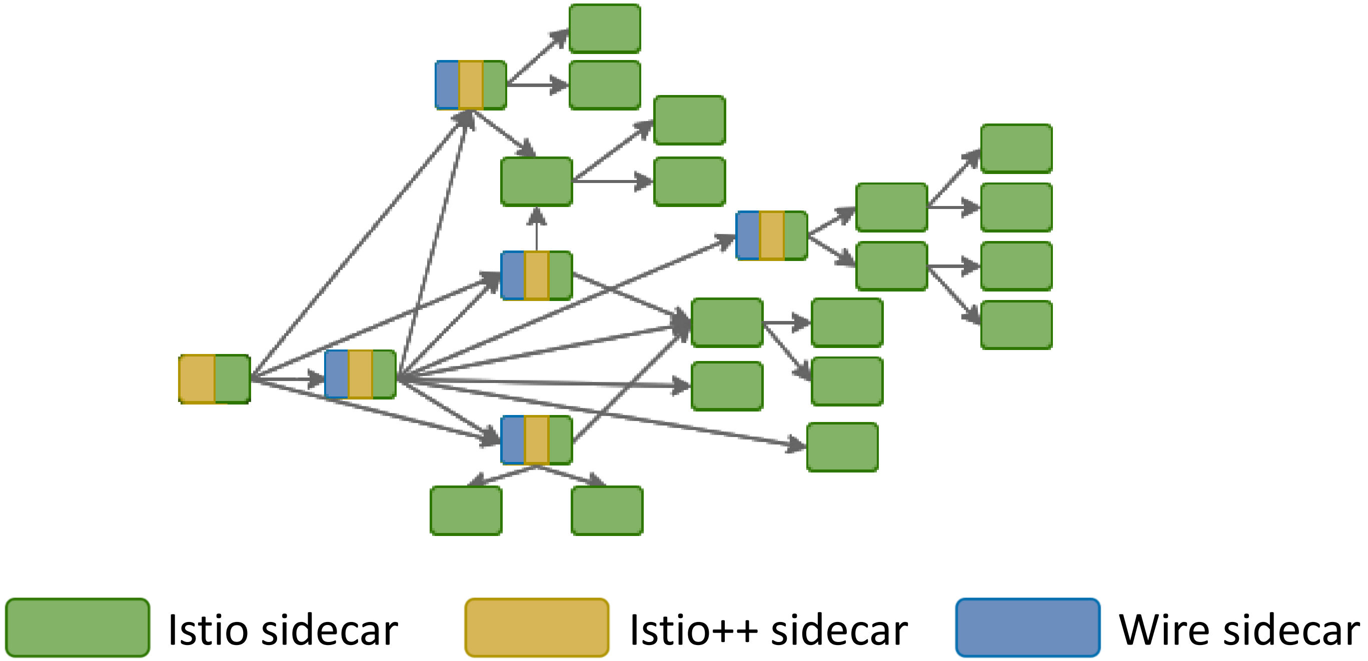


Can *Wire* Lower Dataplane Overheads?

Enforced policy: Header manipulation rules for a set of contexts

By systematically reducing sidecars, Wire’s configured dataplane can sustain higher throughput.





Copper Wire: Summary

interface.cui

```
act RPCRequest: Request{
  ....
}
```

```
act HTTPRequest: Request{
  ....
}
```

A **new abstraction** for dataplanes to express their functionalities.

policy.cup

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  context (“A.*D")
) {
  ....
}
```

Policy specification over **contexts** is easy and intuitive.

Copper Wire: Summary

interface.cui

```
act RPCRequest: Request{
  ....
}
```

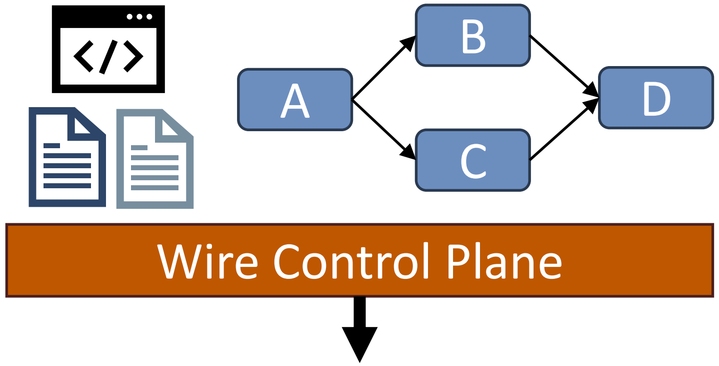
```
act HTTPRequest: Request{
  ....
}
```

A **new abstraction** for dataplanes to express their functionalities.

policy.cup

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  context (“A.*D")
) {
  ....
}
```

Policy specification over **contexts** is easy and intuitive.



Optimal Dataplane Sidecars

Using **application graph, policy contexts**, and **policy semantics,** Wire minimizes dataplane overheads.

Copper Wire: Summary

interface.cui

```
act RPCRequest: Request{
  ....
}
```

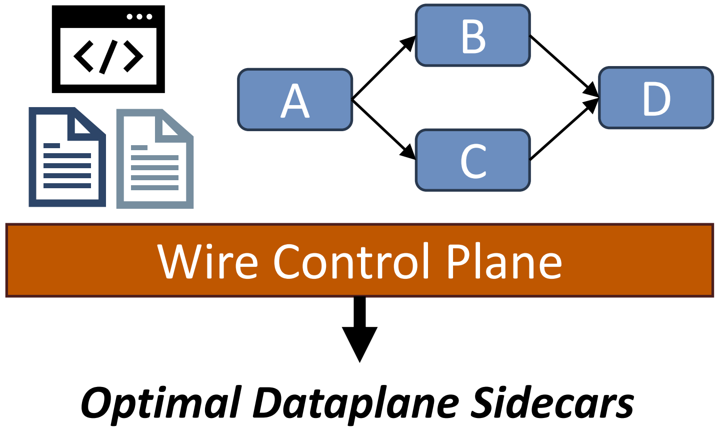
```
act HTTPRequest: Request{
  ....
}
```

A **new abstraction** for dataplanes to express their functionalities.

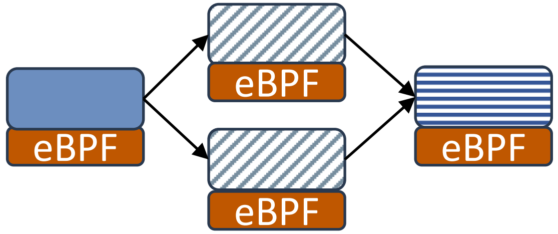
policy.cup

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  context (“A.*D")
) {
  ....
}
```

Policy specification over **contexts** is easy and intuitive.



Using **application graph, policy contexts**, and **policy semantics,** Wire minimizes dataplane overheads.



eBPF add-ons perform context propagation, avoiding heavy sidecars.

Copper Wire: Summary

interface.cui

```
act RPCRequest: Request{
  ....
}
```

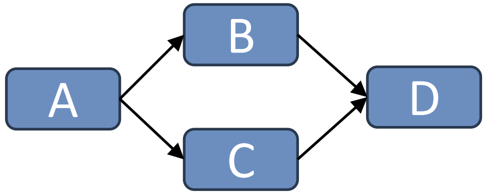
```
act HTTPRequest: Request{
  ....
}
```

A **new abstraction** for dataplanes to express their functionalities.

policy.cup

```
import "interface.cui"
policy distribute (
  act (RPCRequest req)
  context (“A.*D")
) {
  ....
}
```

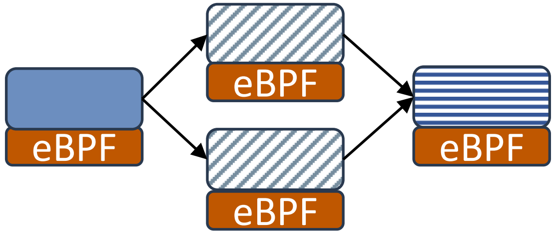
Policy specification over **contexts** is easy and intuitive.



Wire Control Plane

Optimal Dataplane Sidecars

Using **application graph, policy contexts**, and **policy semantics,** Wire minimizes dataplane overheads.



eBPF add-ons perform context propagation, avoiding heavy sidecars.

With Copper Wire, we redesign the entire service mesh stack leading to **simpler policy specification and better performance**

Thank You!

Questions?

Divyanshu Saxena (dsaxena@cs.utexas.edu)

https://divyanshusaxena.github.io/