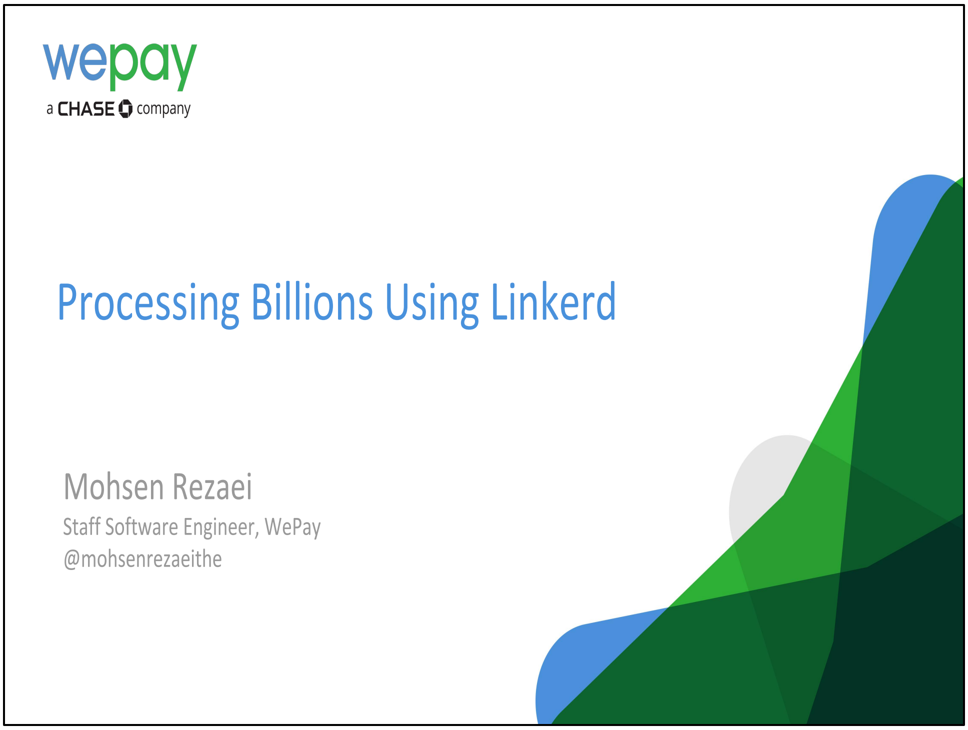


wepay

a CHASE company

Processing Billions Using Linkerd

Mohsen Rezaei

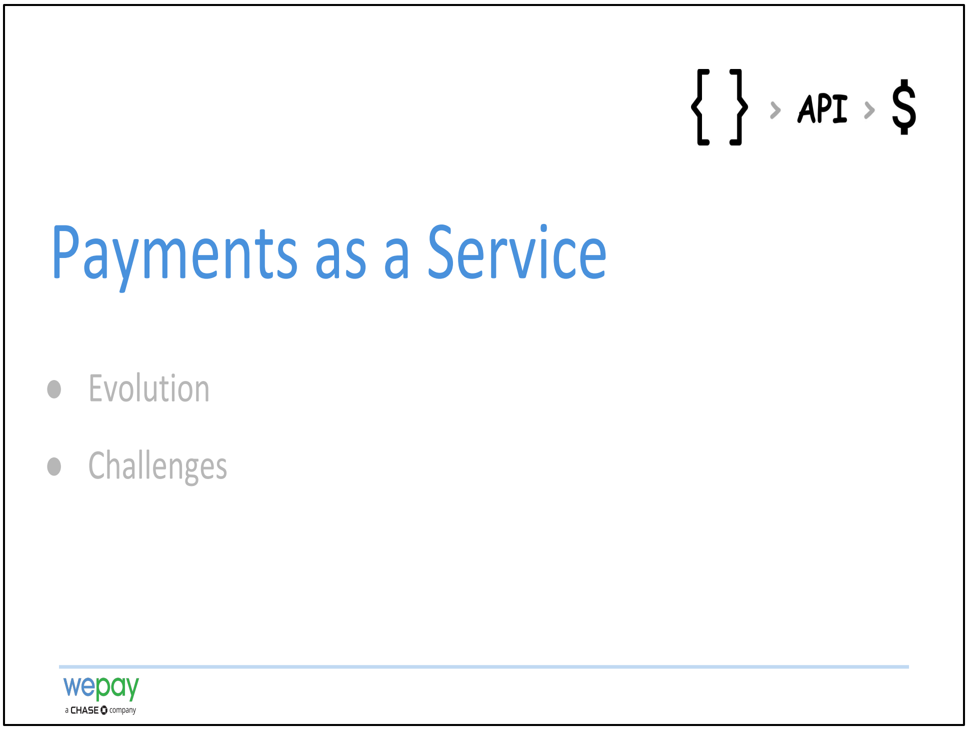
Staff Software Engineer, WePay

@mohsenrezaeithe

In 2018, WePay’s infrastructure was **integrated with Linkerd** and fully **migrated all the traffic** to use Linkerd for all requests.

Agenda

1. Payments as a Service
2. Modern Service Graph
3. Day 2+



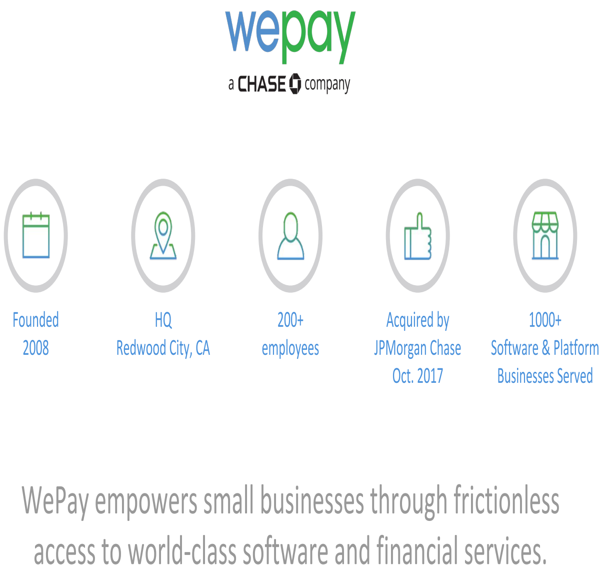
{ } › API › $

Payments as a Service

- Evolution
- Challenges

The big picture:

- How did we arrive at service mesh for our traditional infrastructure?
- Ties into the challenges that service mesh solved.



wepay
a CHASE company

Founded
2008

HQ
Redwood City, CA

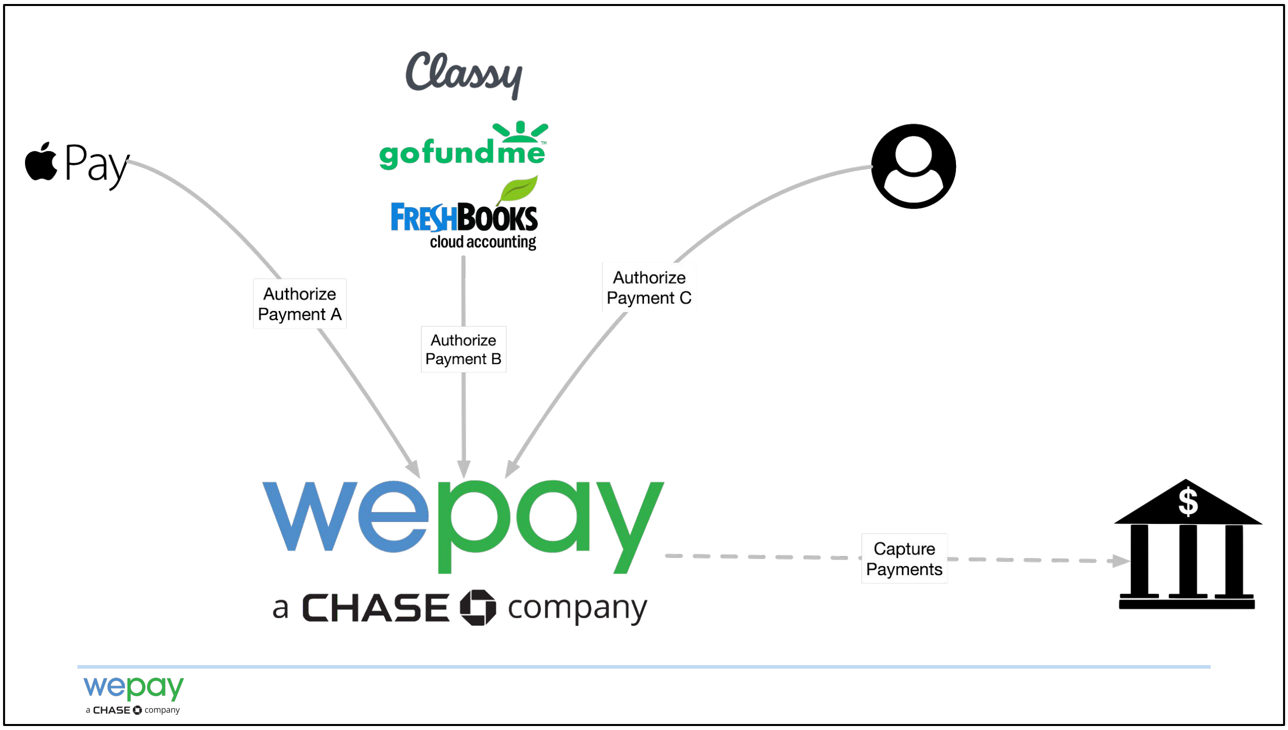
200+
employees

Acquired by
JPMorgan Chase
Oct. 2017

1000+
Software & Platform
Businesses Served

WePay empowers small businesses through frictionless access to world-class software and financial services.

<u>Business focus:</u> We provide **software solutions to marketplaces** and other platforms to **facilitate payments**.



Authorize Payment A

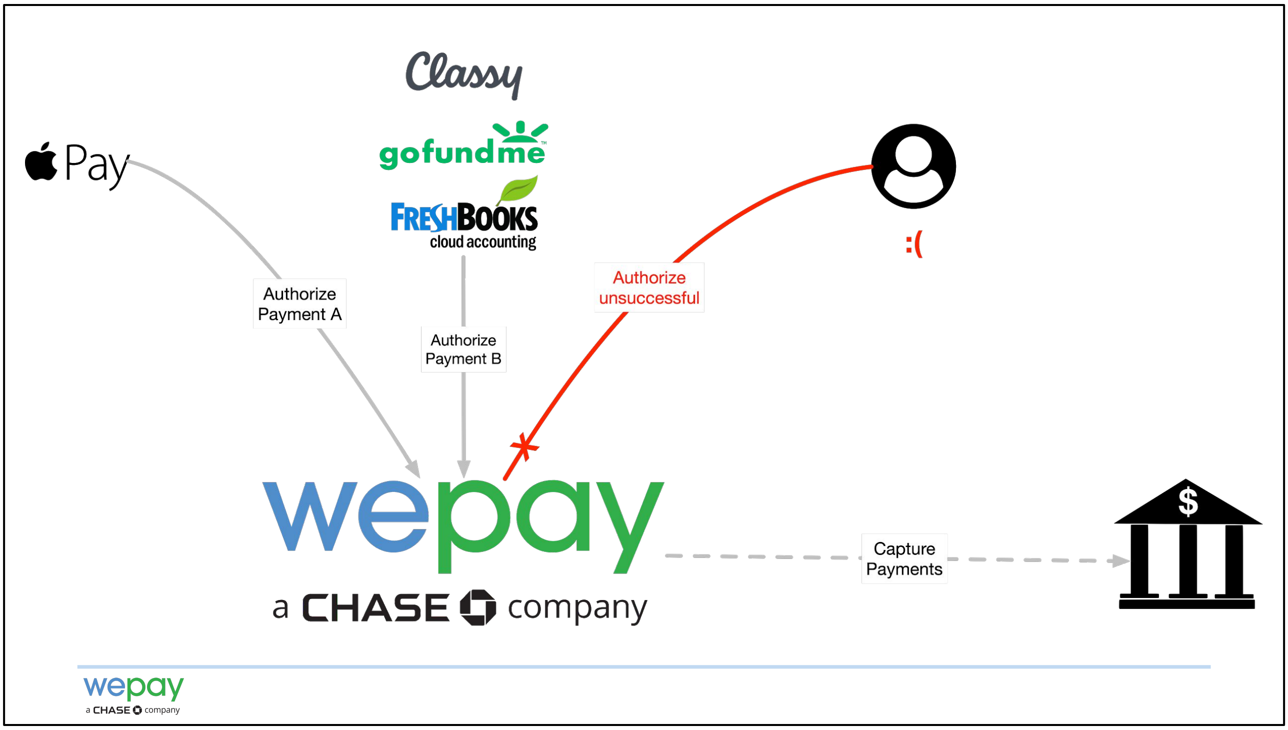
Authorize Payment B

Authorize Payment C

Capture Payments

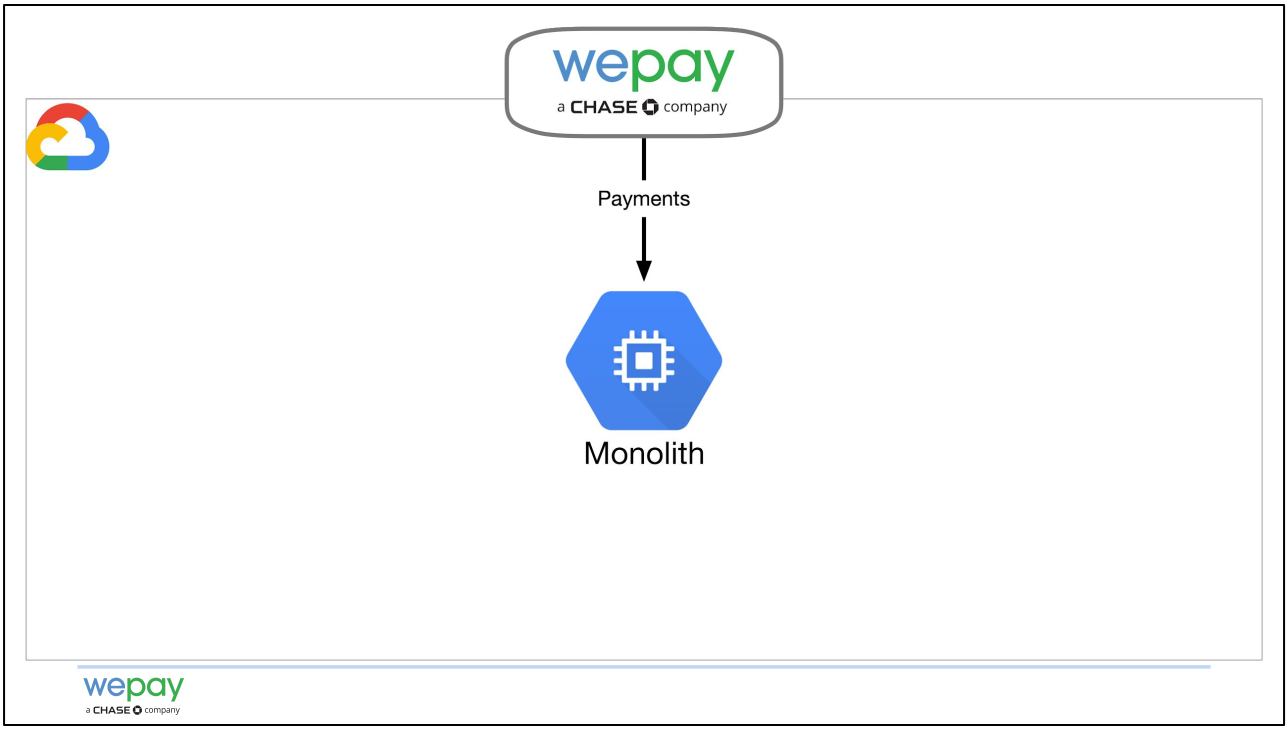
WePay provides **public payment APIs** that allow payment partners to **synchronously authorize charge of payments** they receive from their users.

Payments that are successfully authorized, are **captured in the background** where the actual money movement happens.

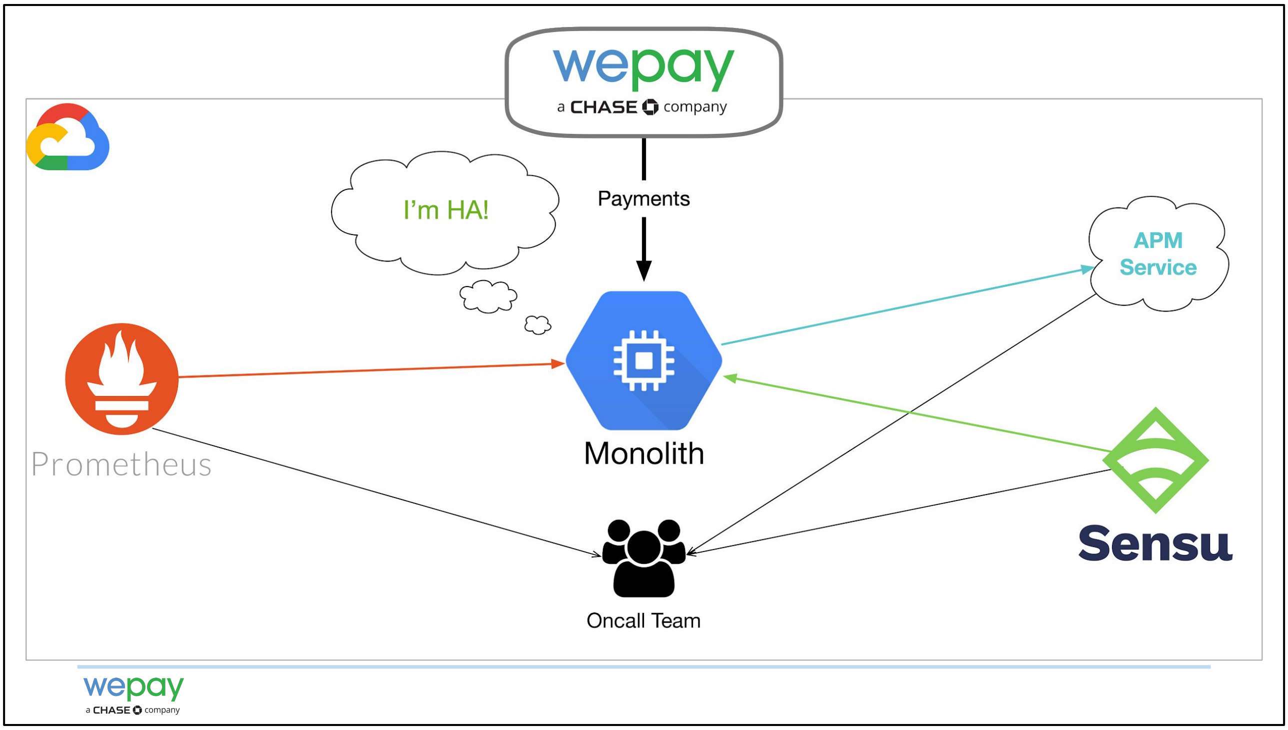


As a **highly available payment service**, the goal is to provide a **very high success rate for the valid payments** being sent to our APIs.

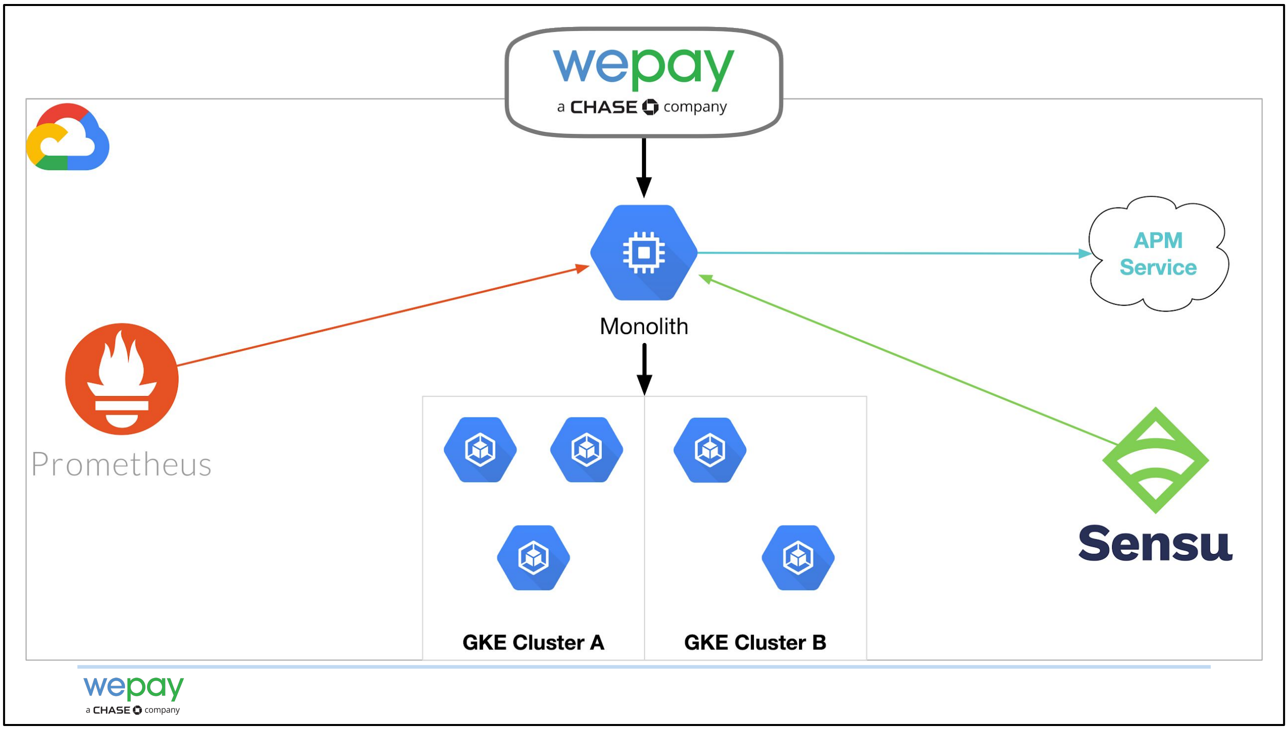
infrastructure or internal server **issues internally could cause payment processing failures** seen by API customers. This is **not ideal**!



A couple of years ago, these **APIs were backed by a single monolithic application**, running in Google Cloud.

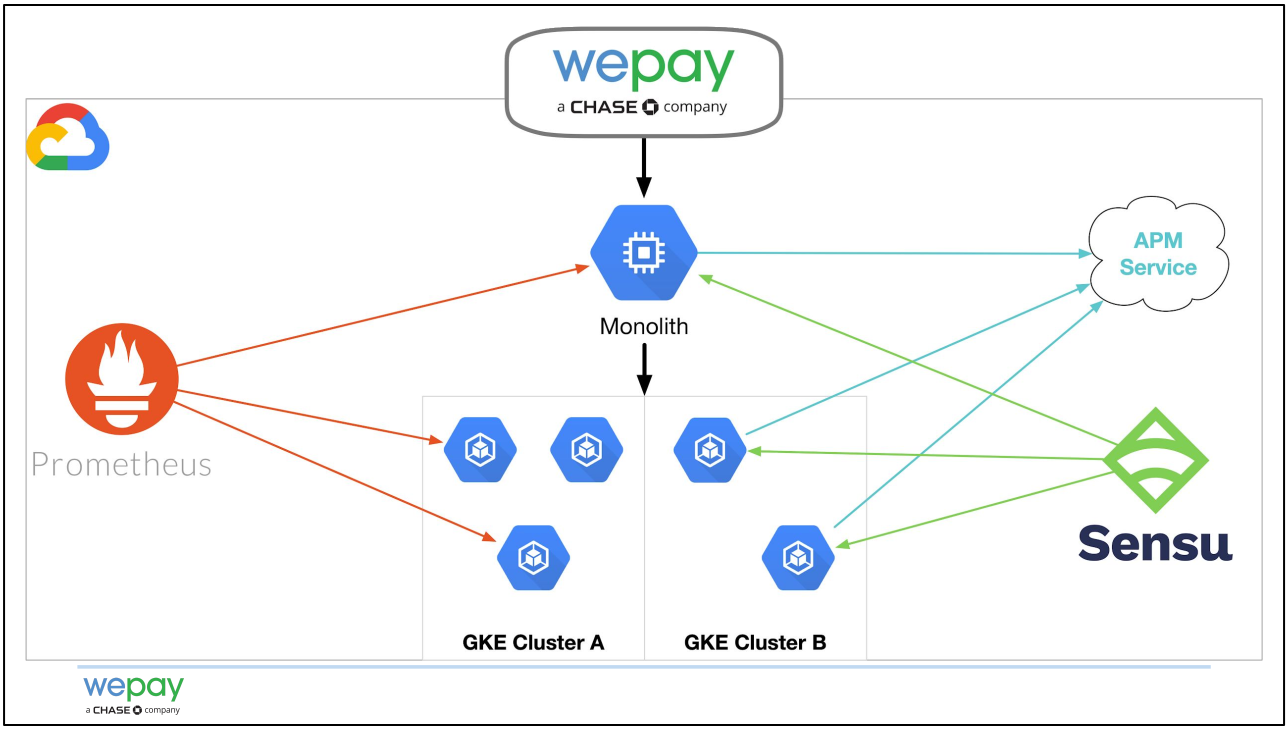


To maintain an overall **highly available product**, various monitoring services were used to **monitor the monolithic service's activities**.

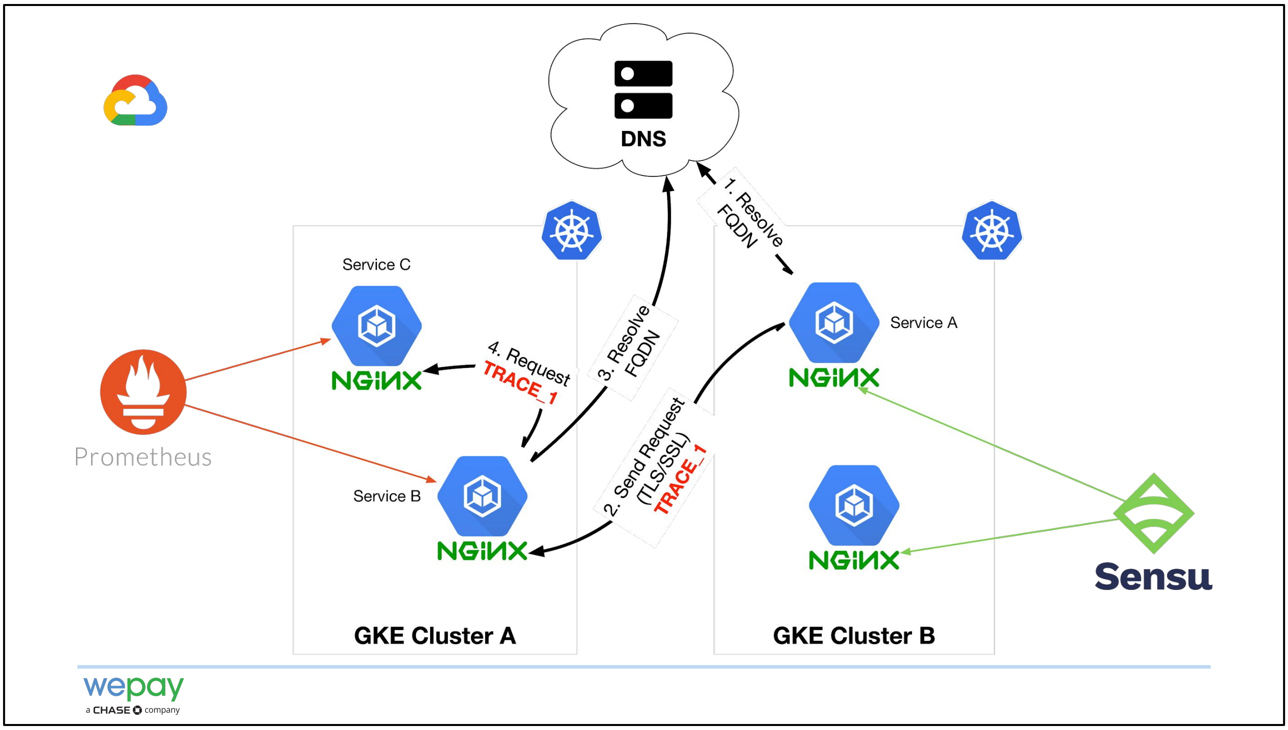


For **easier development**:

- **Monolith got refactored** into smaller microservices
- Introduced Google Kubernetes Engine **(GKE) to the environment to host all new microservices**
- **Groups of services** were setup into different network subdomains and GKE clusters

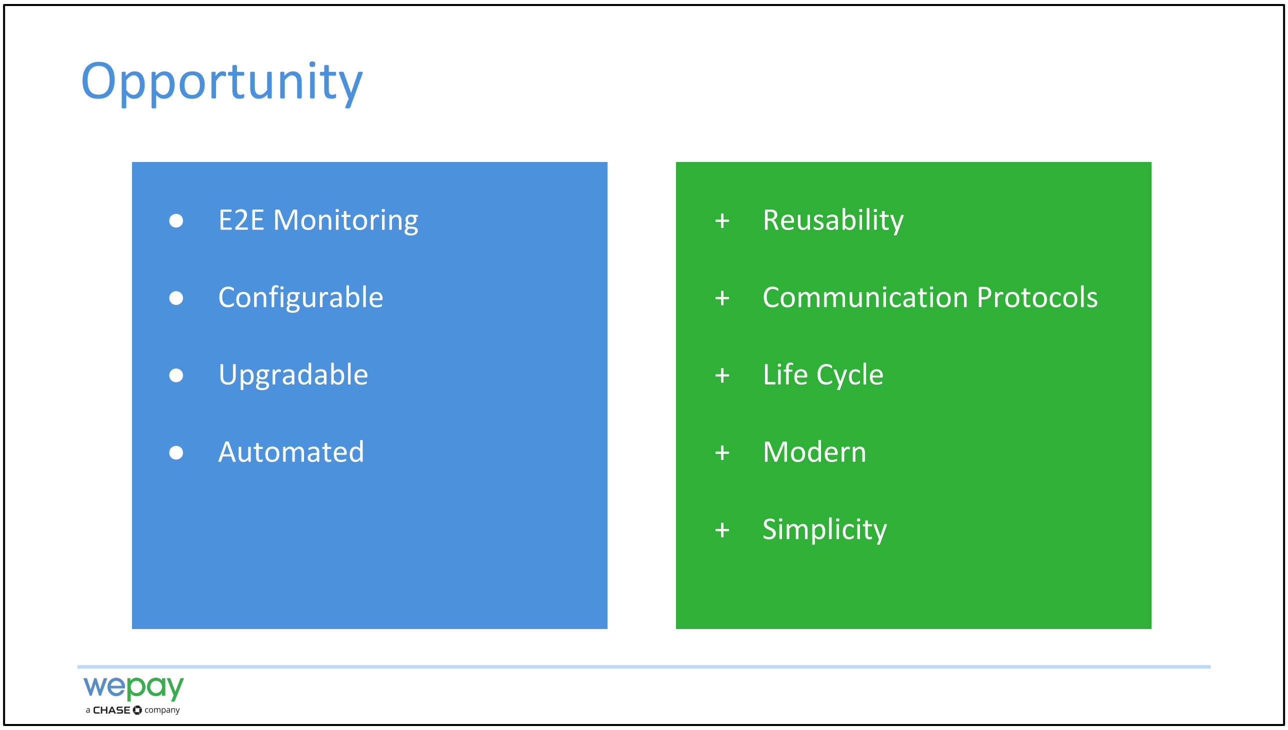


...and applied the **same monitoring best practices** to the microservices



Details of microservices scope:

- **NGINX terminates SSL** to ensure secure s2s communications
- **Services use a FQDN resolver** (internal or external) to find their downstream services
- **Services are responsible to generate necessary metrics and tracing** information to monitor and debug the graph
- **Prometheus gathers the generated metrics** for aggregation and visualization
- **Sensu is configured to test the same microservice entry points** used by other services



Opportunity

- E2E Monitoring
- Configurable
- Upgradable
- Automated

+ Reusability

+ Communication Protocols

+ Life Cycle

+ Modern

+ Simplicity

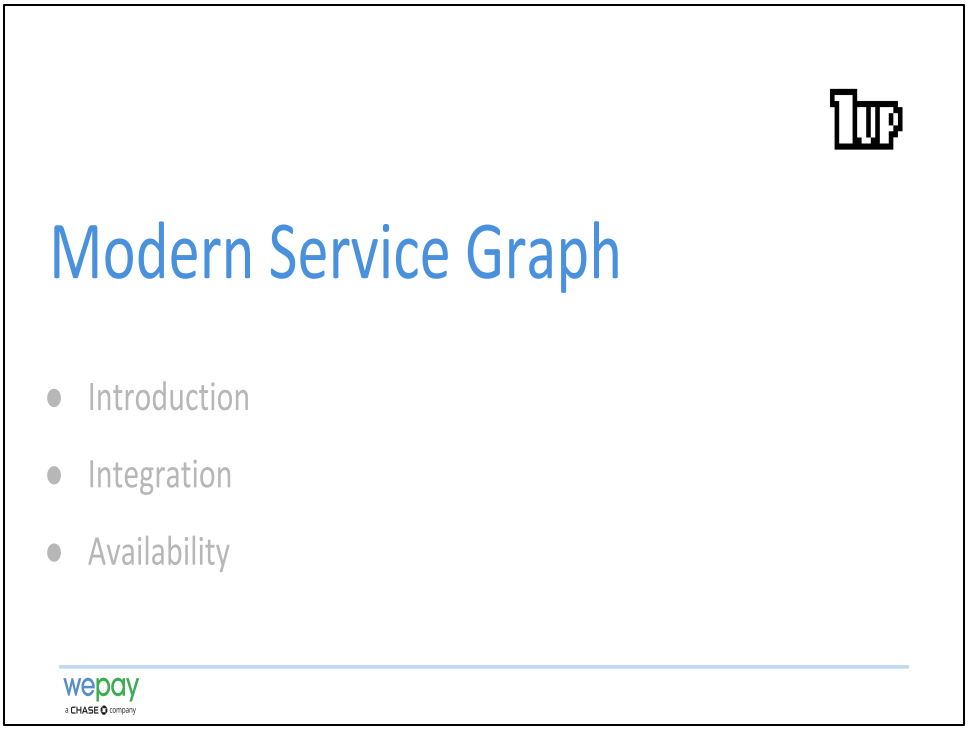
Successfully **developed the traditional infrastructure** with things on the left:

- **Every piece is monitored** end-to-end with appropriate alerting.
- All pieces in the **infra is configurable** either through a centrally distributed configuration or service specific ones.
- Every piece **can be upgraded independently** with no effect on other pieces in the environment.
- Things like **deployments, health checking, etc, are automated** and handled by tools or pipelines.

The setup is **complex and too closely integrated with services** running on it, which doesn't encourage big improvements.

These challenges opened an opportunity to **improve our traditional infrastructure**. Inline with codable infrastructure (IaC), infrastructure needed to change to have the things on the right:

- Separation of concerns
- Adding/modifying/removing support for different protocols
- Easier and more maintainable life cycle
- Modern operations, e.g. more useful software load balancing features
- Zero config, zero code integration

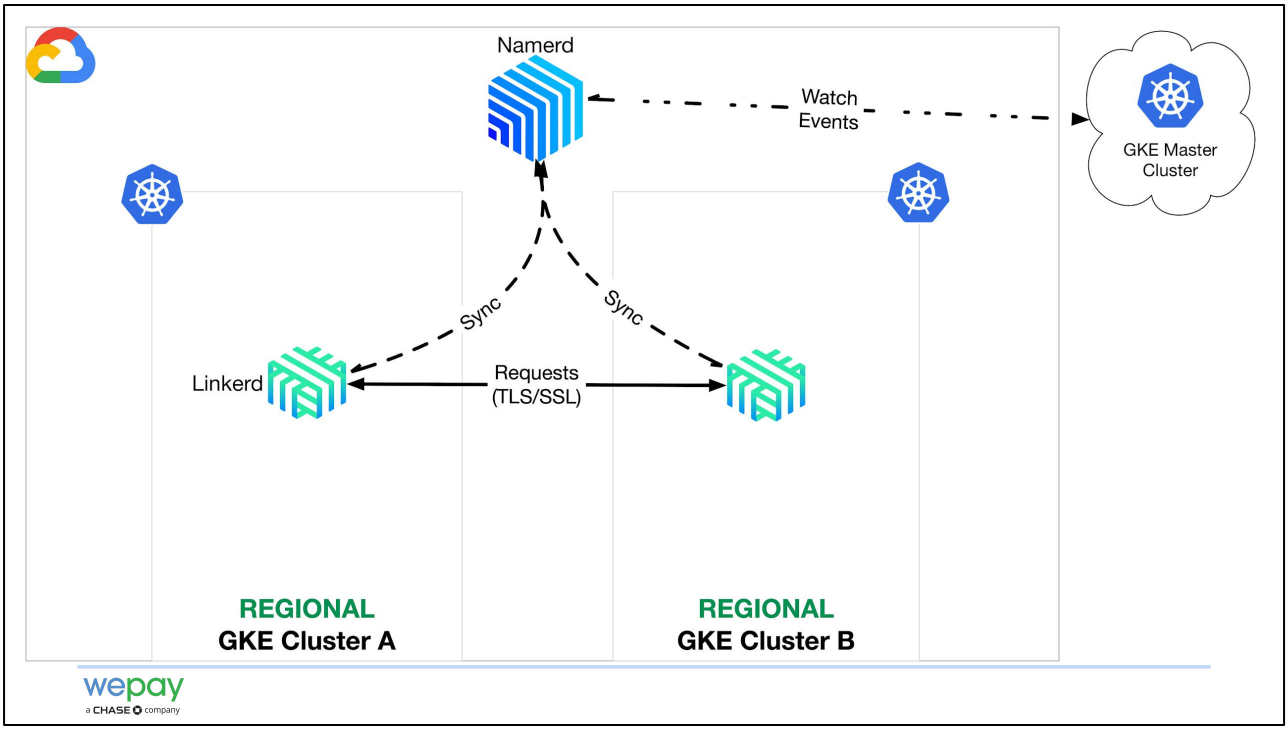


Modern Service Graph

- Introduction
- Integration
- Availability

Main goals of integrating with service mesh: **Organization** and **modernization.**

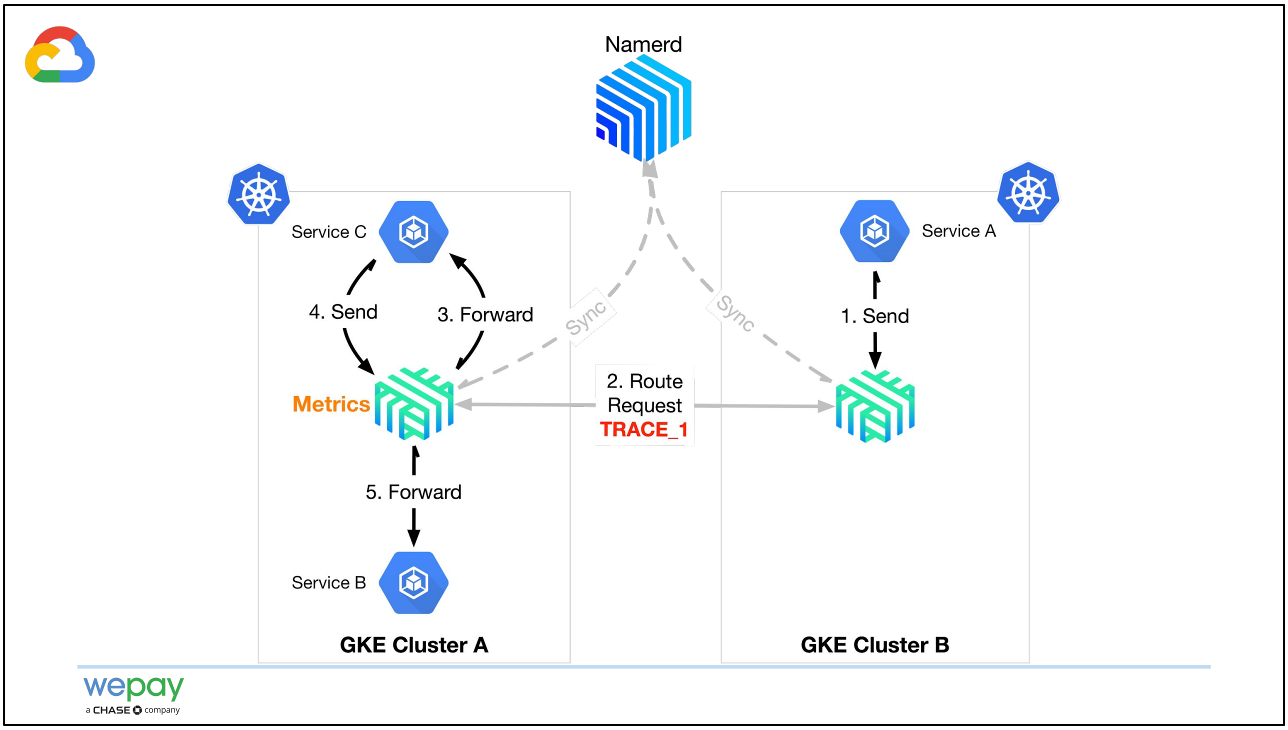
Going to infrastructure 2.0 with service mesh, involved **three major steps**.



Service mesh generally involves a data and a control plane, **Linkerd proxies** and **Namerd**, respectively:

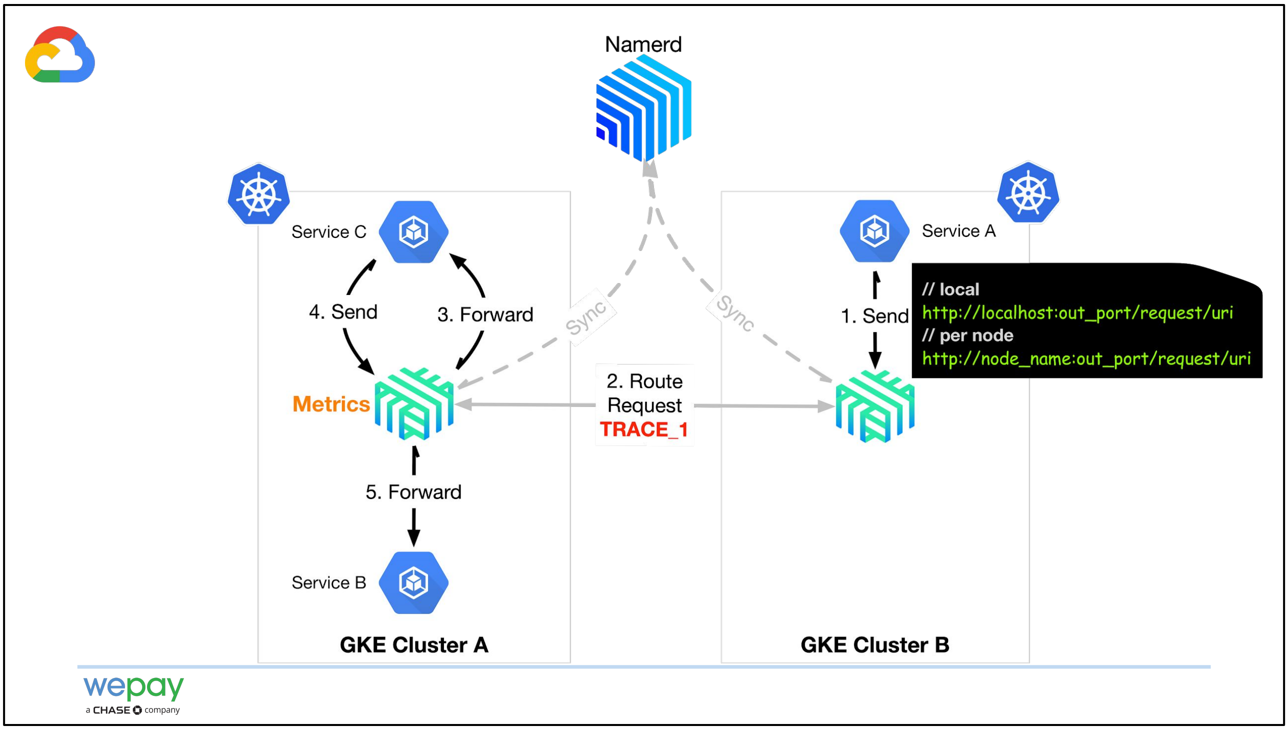
- Proxies carry data around, **deliver requests**, etc.
- Namerd gives **service discovery**.

GKE **regional clusters provide HA discovery** for the service mesh infrastructure, by providing a LB for the horizontally scaled GKE masters.



Layering the service mesh infrastructure with the microservices layout:

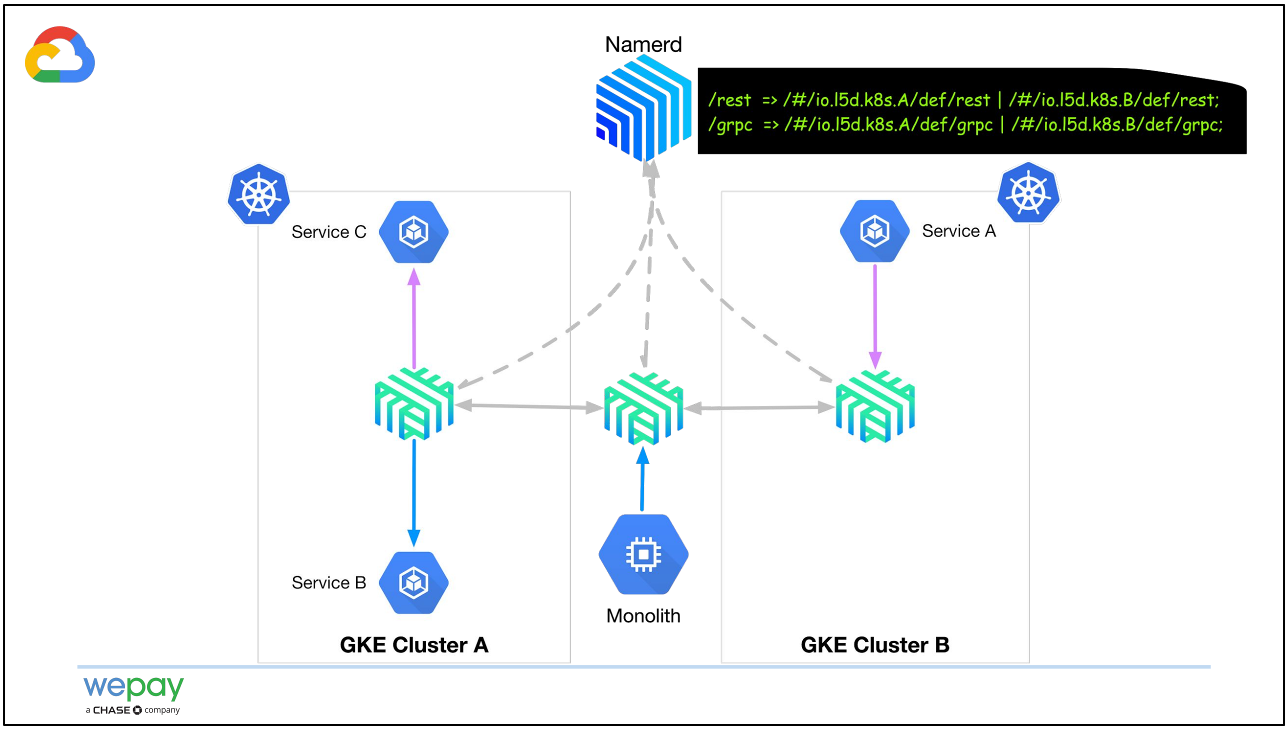
- Microservices don’t need to resolve names for sending requests (**Namerd provides discovery** to the Linkerd proxies)
- As the requests go through Linkerd, **trace information are generated by Linkerd** and can be gathered for visualization
- All **Linkerds generate metrics** for system and request activities.



```
// local
http://localhost:out_port/request/uri
// per node
http://node_name:out_port/request/uri
```

Services integrate with their proxies based on a sidecar (local) setup or a DaemonSet (per node).

Challenge: Injecting services with appropriate proxies for DaemonSet model at runtime.



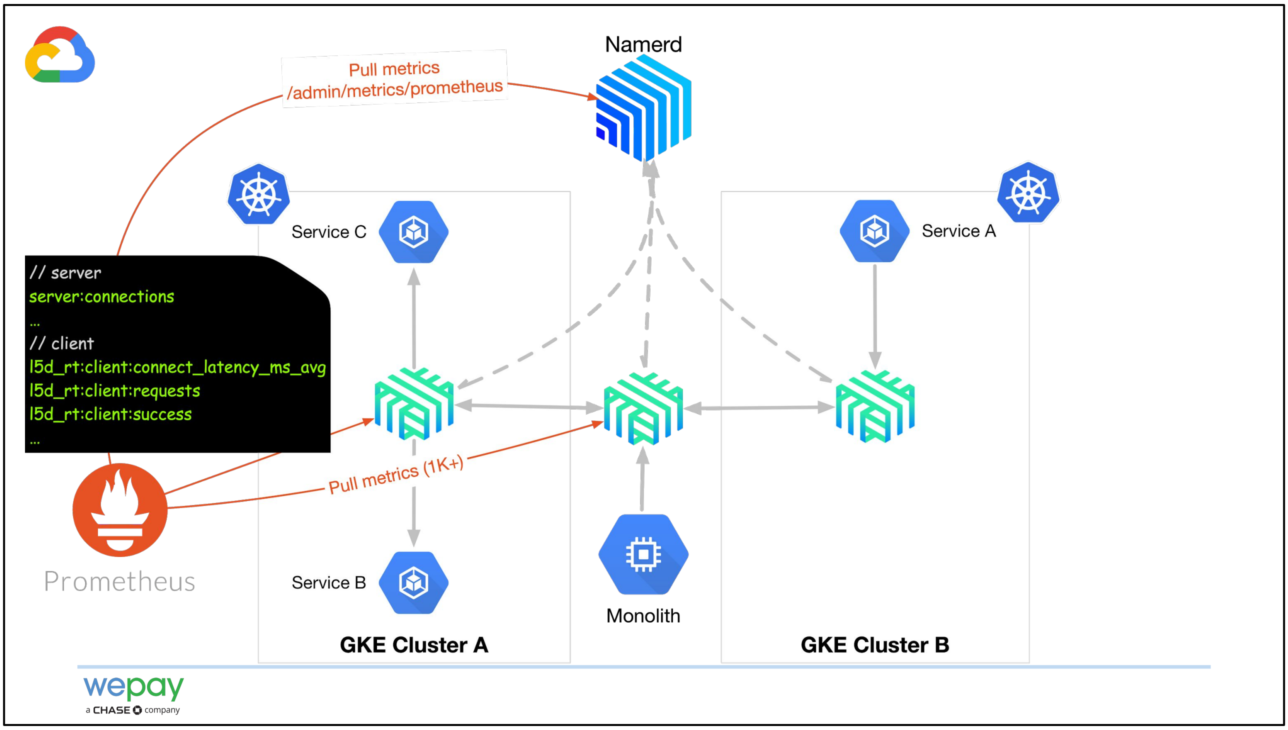
```
/rest  => /#/io.l5d.k8s.A/def/rest | /#/io.l5d.k8s.B/def/rest;
/grpc  => /#/io.l5d.k8s.A/def/grpc | /#/io.l5d.k8s.B/def/grpc;
```

Since we have **both K8s and non-K8s services** in the environment, using **Linkerd 1** to provide service mesh inside and outside of K8s with the **same discovery scope** within a single environment.

Scenarios:

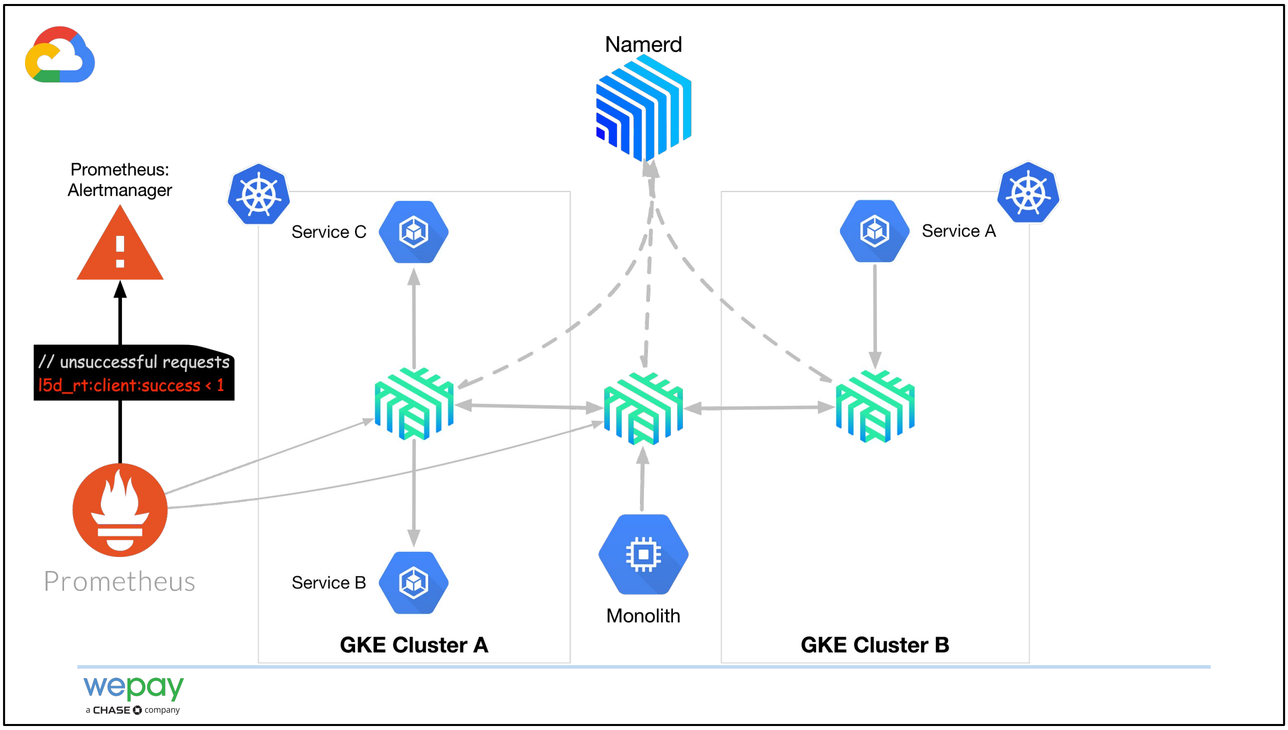
- Monolith sends request to Service B
- Service A sends a request to Service C

In both scenarios the **recipient** of the request is discovered using the **same discovery scope** in Namerd.



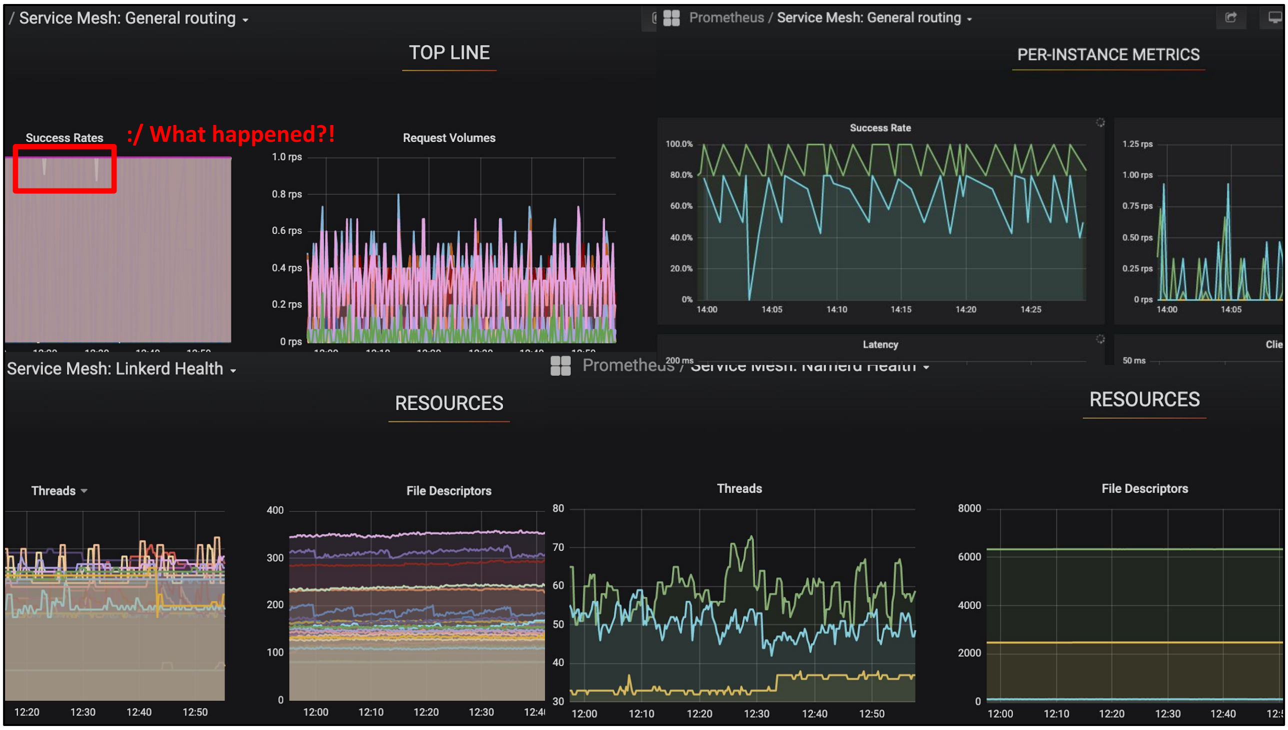
```
// server
server:connections
...
// client
l5d_rt:client:connect_latency_ms_avg
l5d_rt:client:requests
l5d_rt:client:success
...
```

Linkerd and Namerd instances **generate over 1K metric points** related to server and client that is **gathered by Prometheus for visualization and debugging**.



```
// unsuccessful requests
l5d_rt:client:success < 1
```

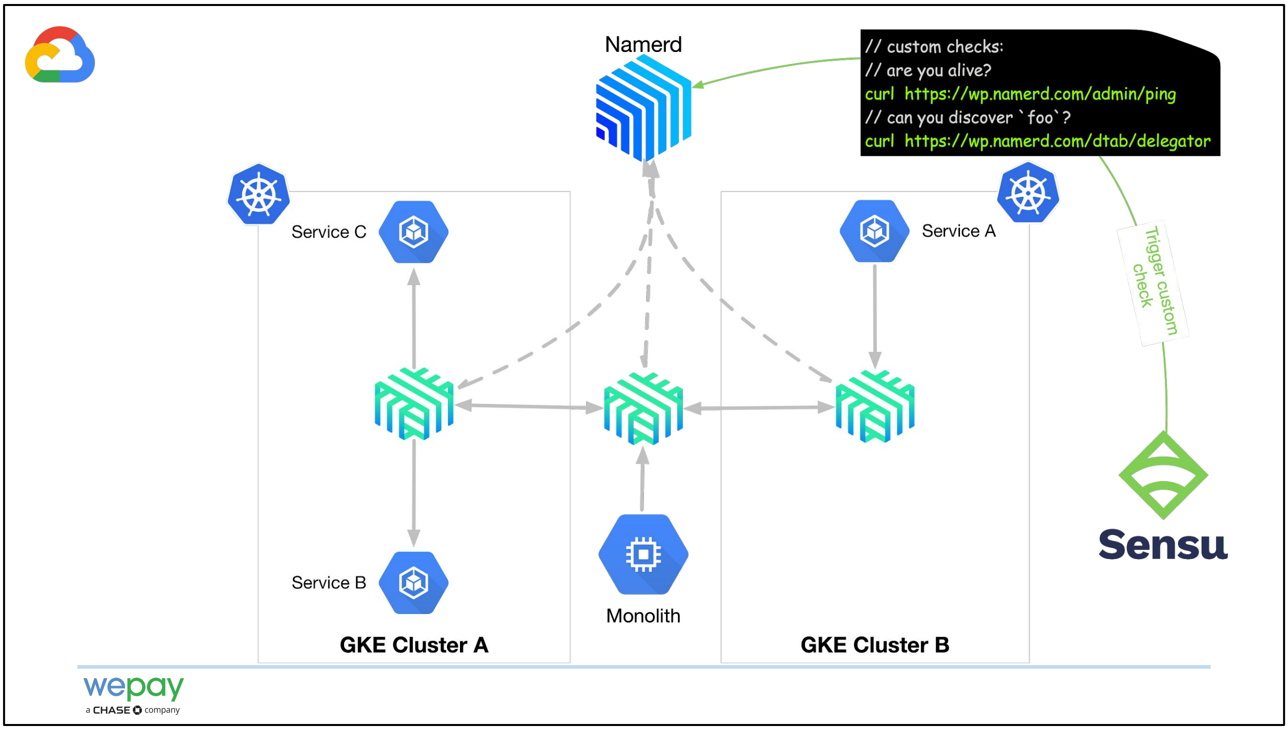
Aggregated metrics are used for **alerting on important events** in the environment.



:/ What happened?!

Wide variety of **visualizations can be achieved from the metrics available** from Namerd and Linkerd.

Helps with debugging live issues and **correlating events with their corresponding metrics** from data and/or control plane.

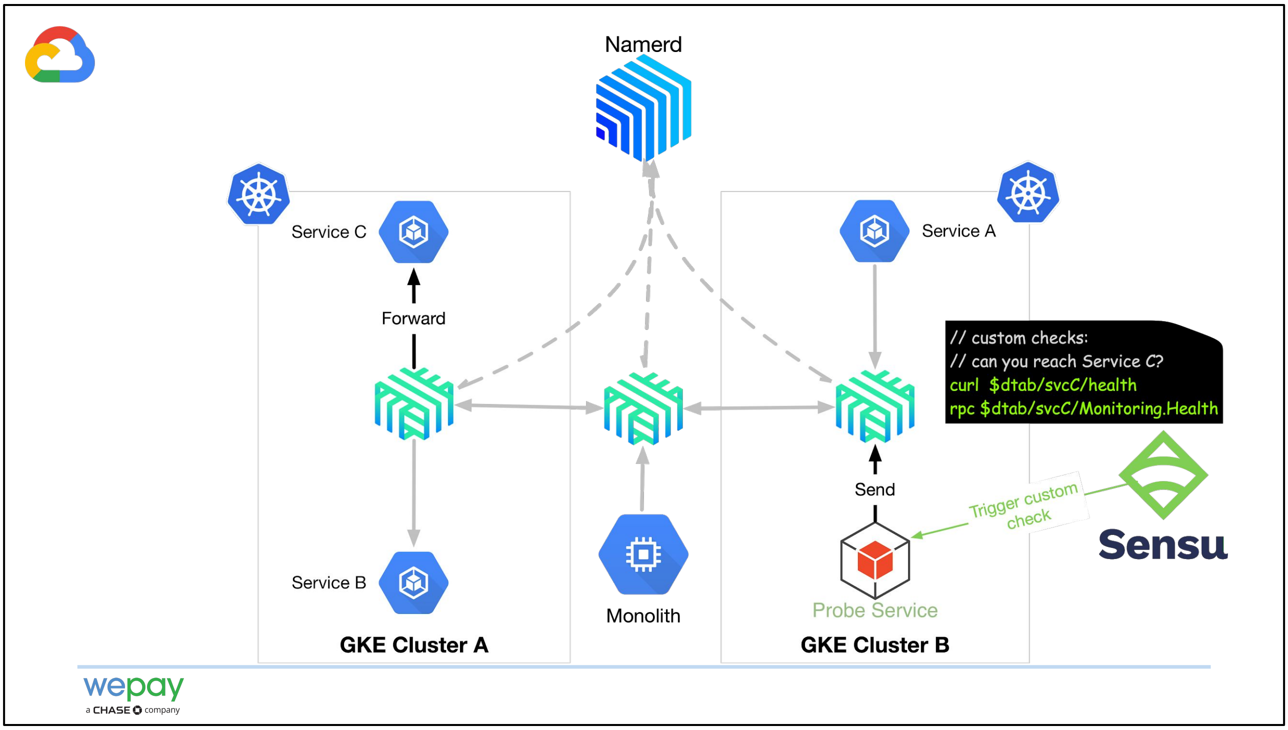


```
// custom checks:
// are you alive?
curl  https://wp.namerd.com/admin/ping
// can you discover `foo`?
curl  https://wp.namerd.com/dtab/delegator
```

High Availability...

Challenge: Ensuring that all services are discoverable within the scope, and can accept requests from other services.

A **service registry drives the expectations** for dynamically defining what discovery checks are run on Sensu.

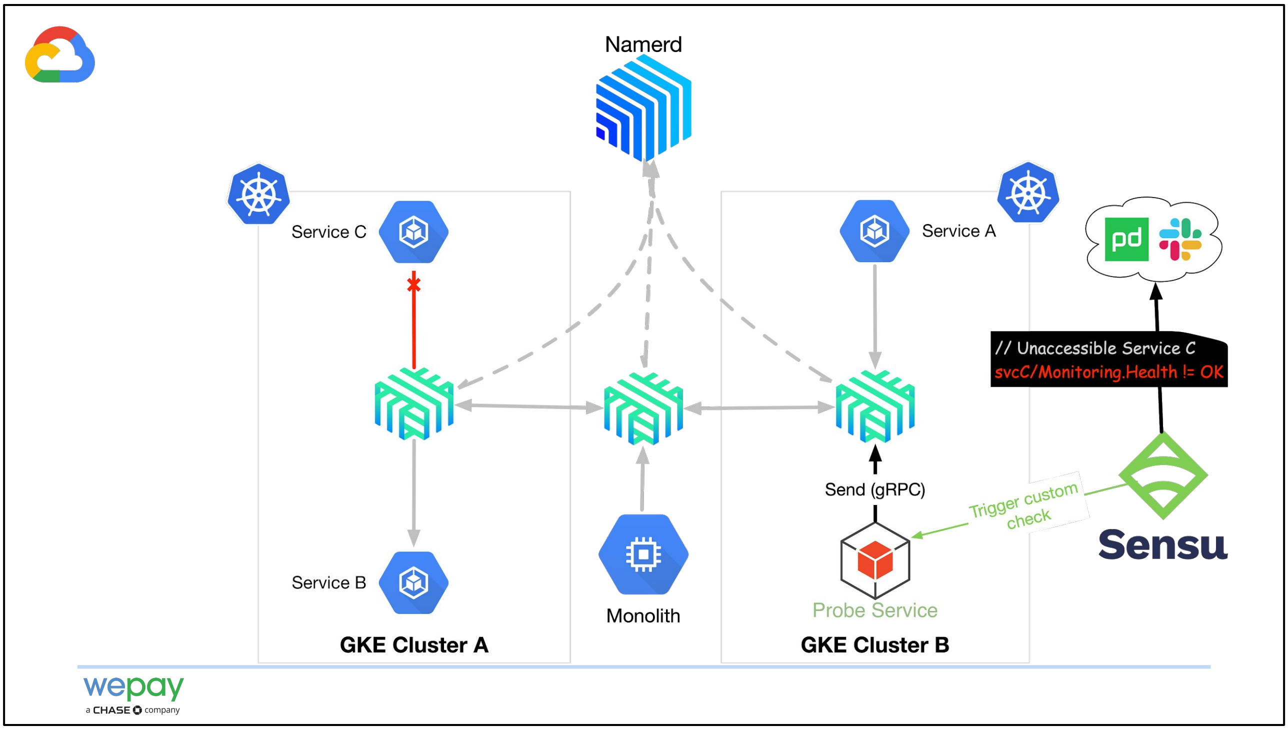


```
// custom checks:
// can you reach Service C?
curl  $dtab/svcC/health
rpc $dtab/svcC/Monitoring.Health
```

Challenge: Checking that all expected services are routable within the scope of service mesh in the environment.

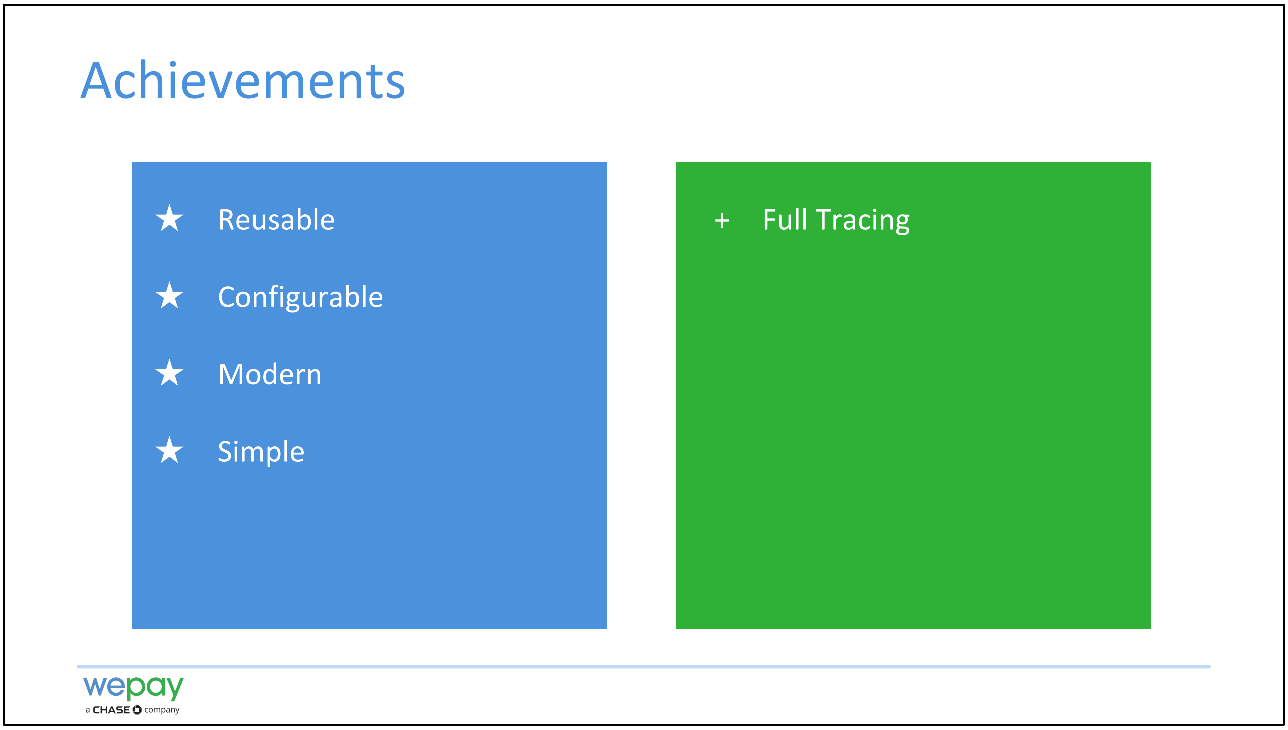
An internal probing service checks health:

- Handles **both RESTful and gRPC** health checking
- Handles both **mesh and non-mesh health** checking (used for comparing both behaviors at migration)
- Gives the **same perspective as other services** in the service mesh scope



```
// Unaccessible Service C
svcC/Monitoring.Health != OK
```

Custom checks trigger alerts based on their own thresholds and alerting criteria.



Achievements

★ Reusable

★ Configurable

★ Modern

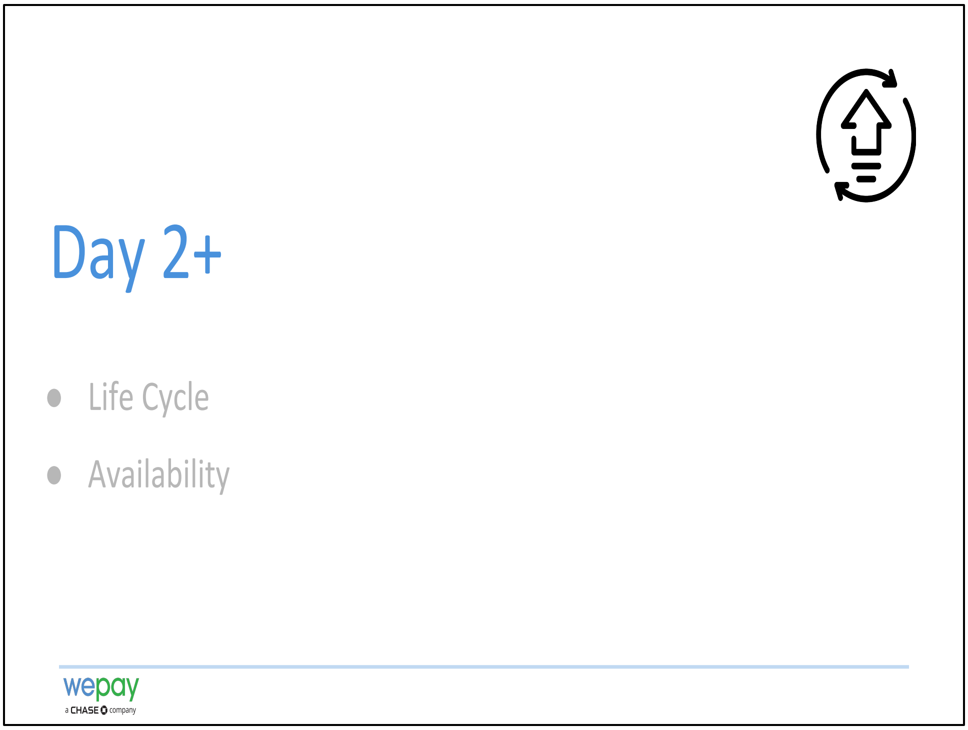
★ Simple

+ Full Tracing

By integrating the infrastructure with a service mesh, the infrastructure has become **simpler and easier to maintain with more modern features**.

Opportunities for improvement:

- Providing 100% live tracing is too expensive for proxies.
- Currently services like Instana help offload tracing from service mesh proxies



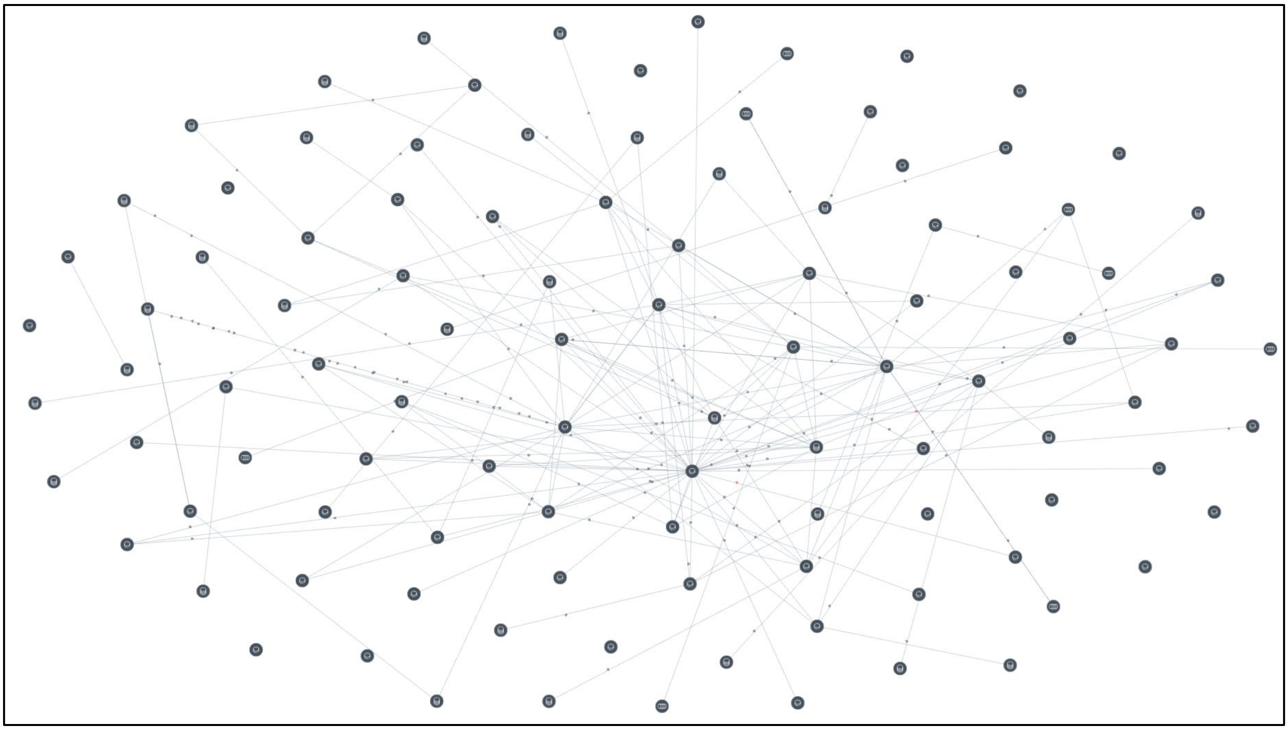
Day 2+

- Life Cycle
- Availability

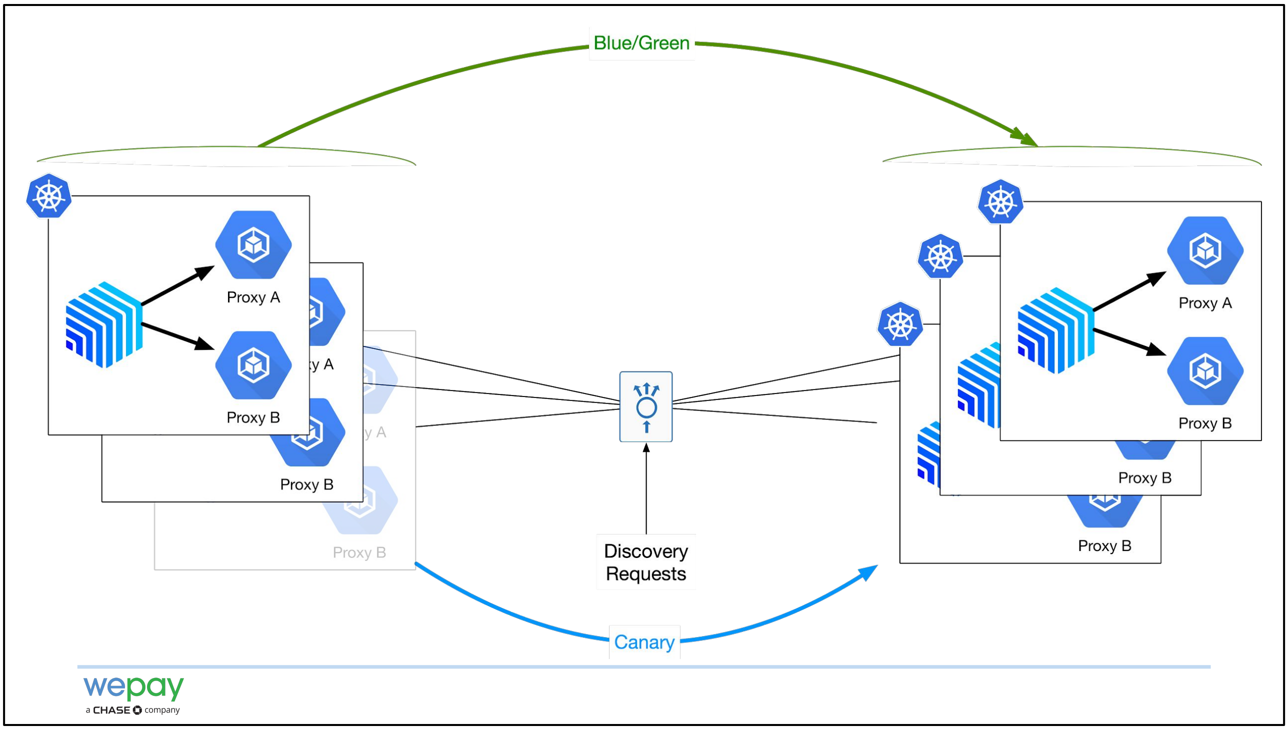
After setting up a highly available infrastructure and data plane, the **focus is on maintaining** all the pieces after the initial setup:

- Changing service mesh configurations
- Upgrading service mesh services

Ensuring all pieces can be maintained **without affecting live traffic and independent of one another**.

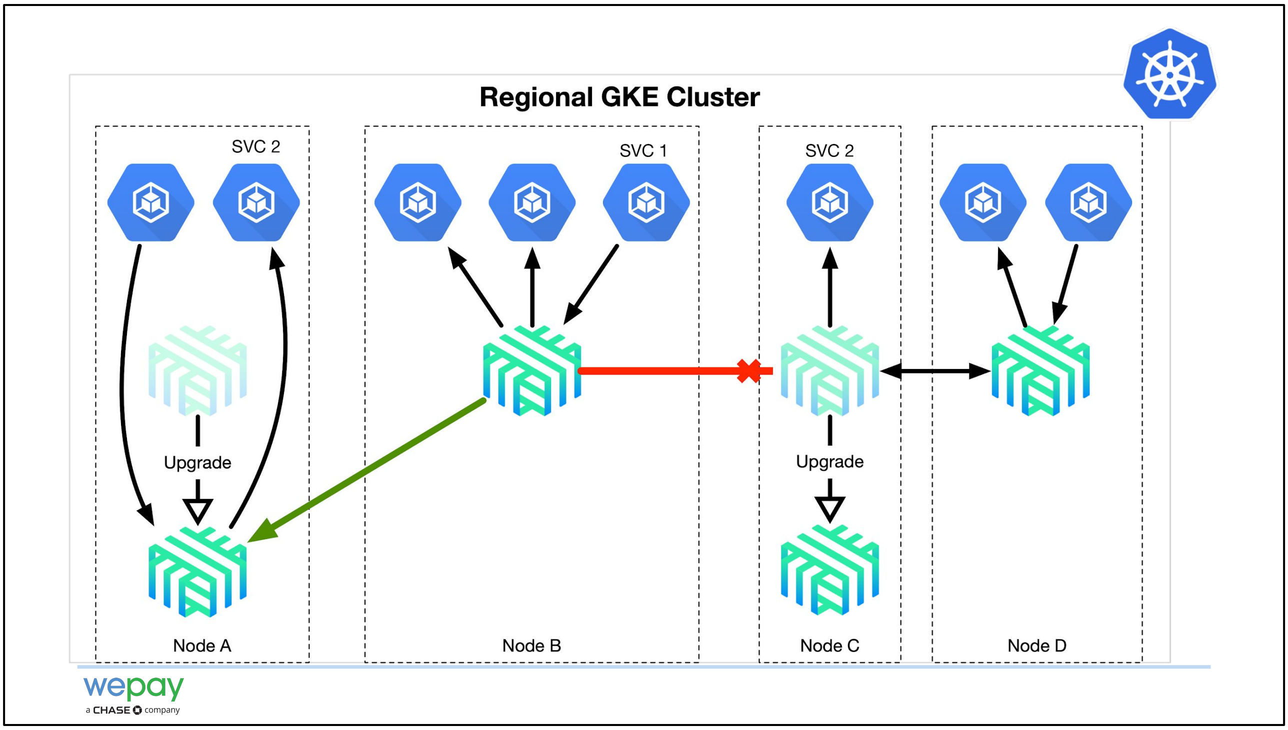


Challenge: Upgrading the service mesh infrastructure as live requests are going through the system.



Namerd **upgrades are easier** by building the service and its proxies into **an independent, horizontally scalable, and isolated pods**.

Any suitable release strategy like canary, rolling update, or blue/green **can be used to upgrade** the service and it’s configurations. This **will not affect the live traffic**, and **rollbacks are easy** in case of service or compatibility problems.



<u>Challenges:</u>

- Not interrupting live traffic
- Rolling out breaking changes
- Rolling out backward incompatible changes
- Rolling out config changes

DaemonSet (per node) setup is a more interesting scenario from upgrading perspective. In this setup, the Linkerd instances **independent of any of the services**' life cycle that use it for routing requests.

Scenario:

- Node A and C Linkerds are upgraded in a rolling update fashion
- SVC 1 sends a request to SVC 2, and since Node C's Linkerd is not available it is routed to Node A
- Node A's Linkerd forwards the request successfully to SVC 2

wecode.wepay.com

Thank you!

Q&A