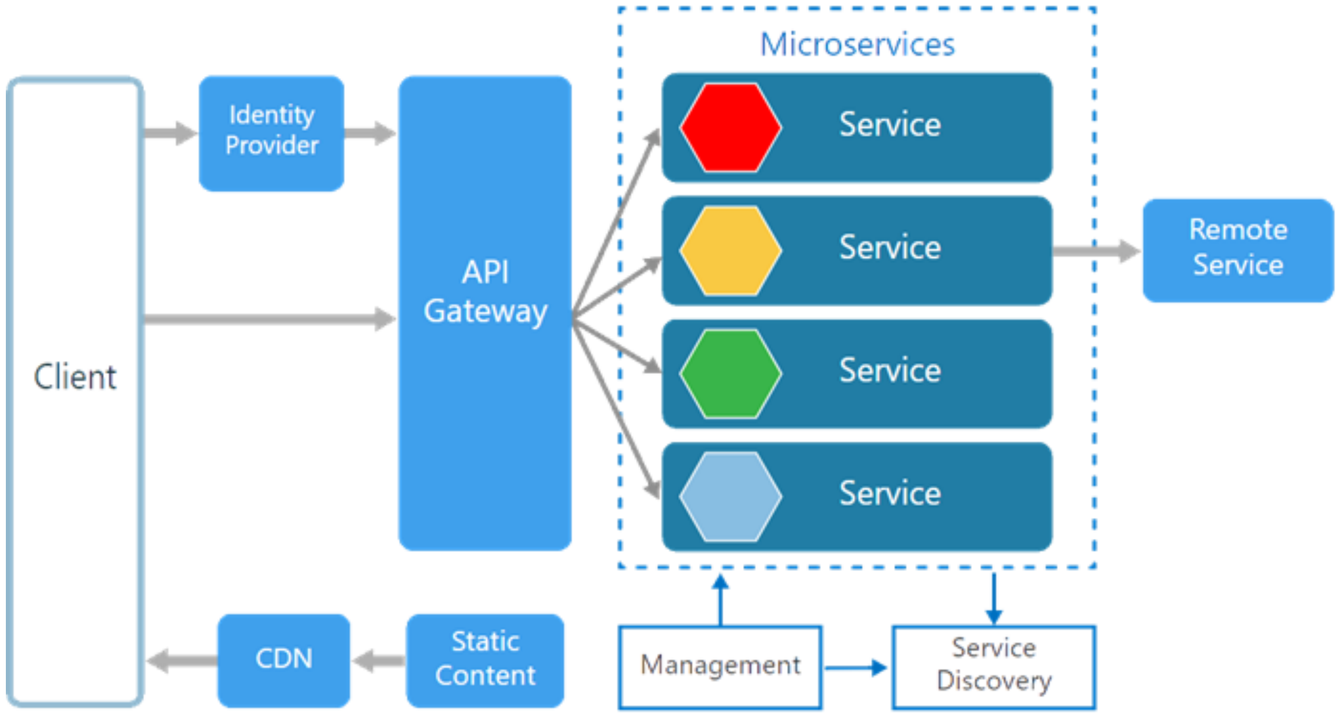
TOPDev



Microservice - 500K CCU

About me

Tran Xuan Viet (Viet Tran)

Solution Architect at **200lab**

- Former Solution Architect at Sendo.
- Former CTO at Skylab
- Former Software Engineer at Foody.

viettranx@gmail.com

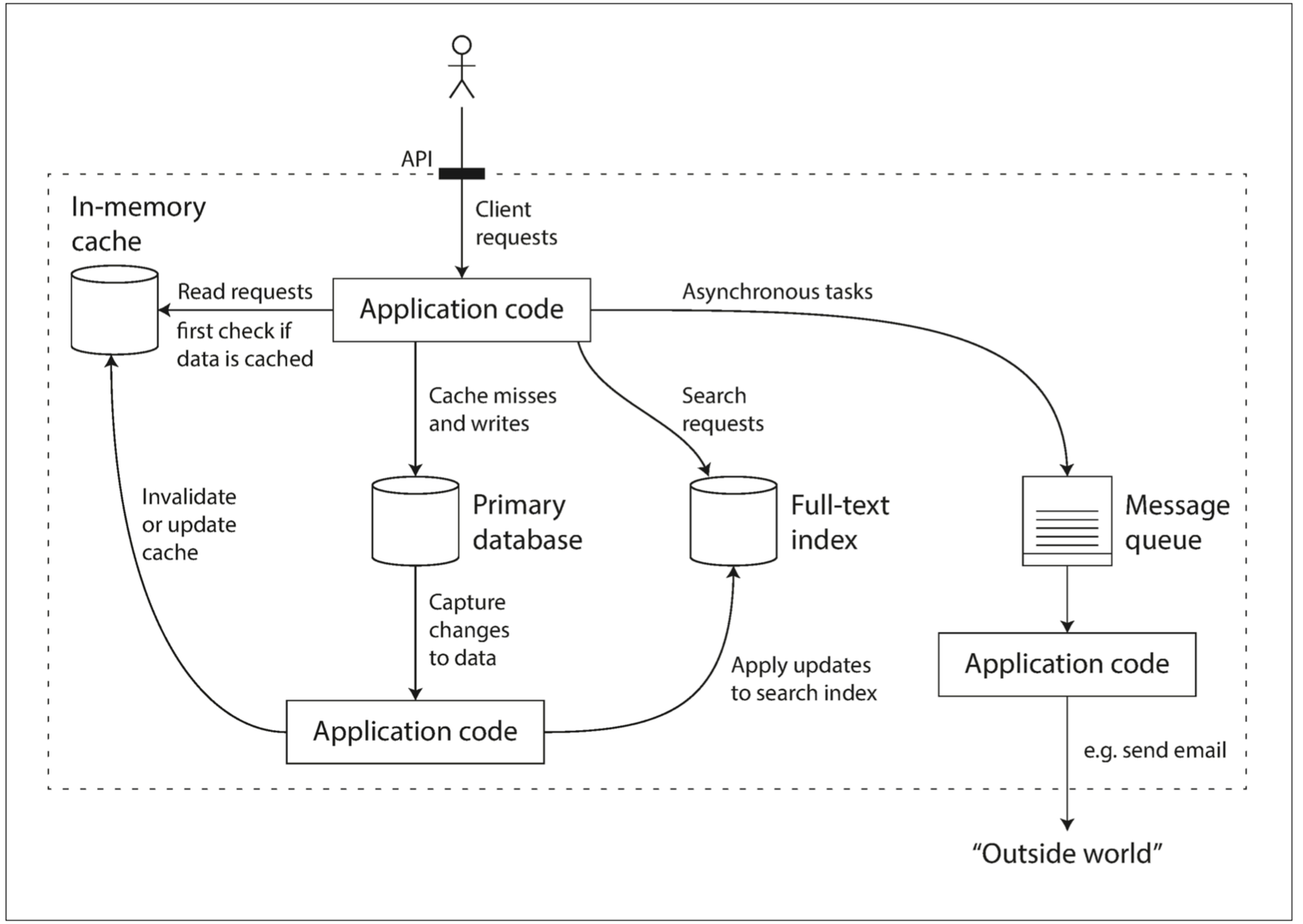
What is CCU?

- CCU = Concurrent User
- Total connected users at the same
- CCU is not Request Per Seconds

Agenda

- A simple high load system
- Monitoring & Tracing
- Monolith app to microservice (Sendo)
- Microservice with gRPC and Protobuf (Sendo)
- Common problems and solutions

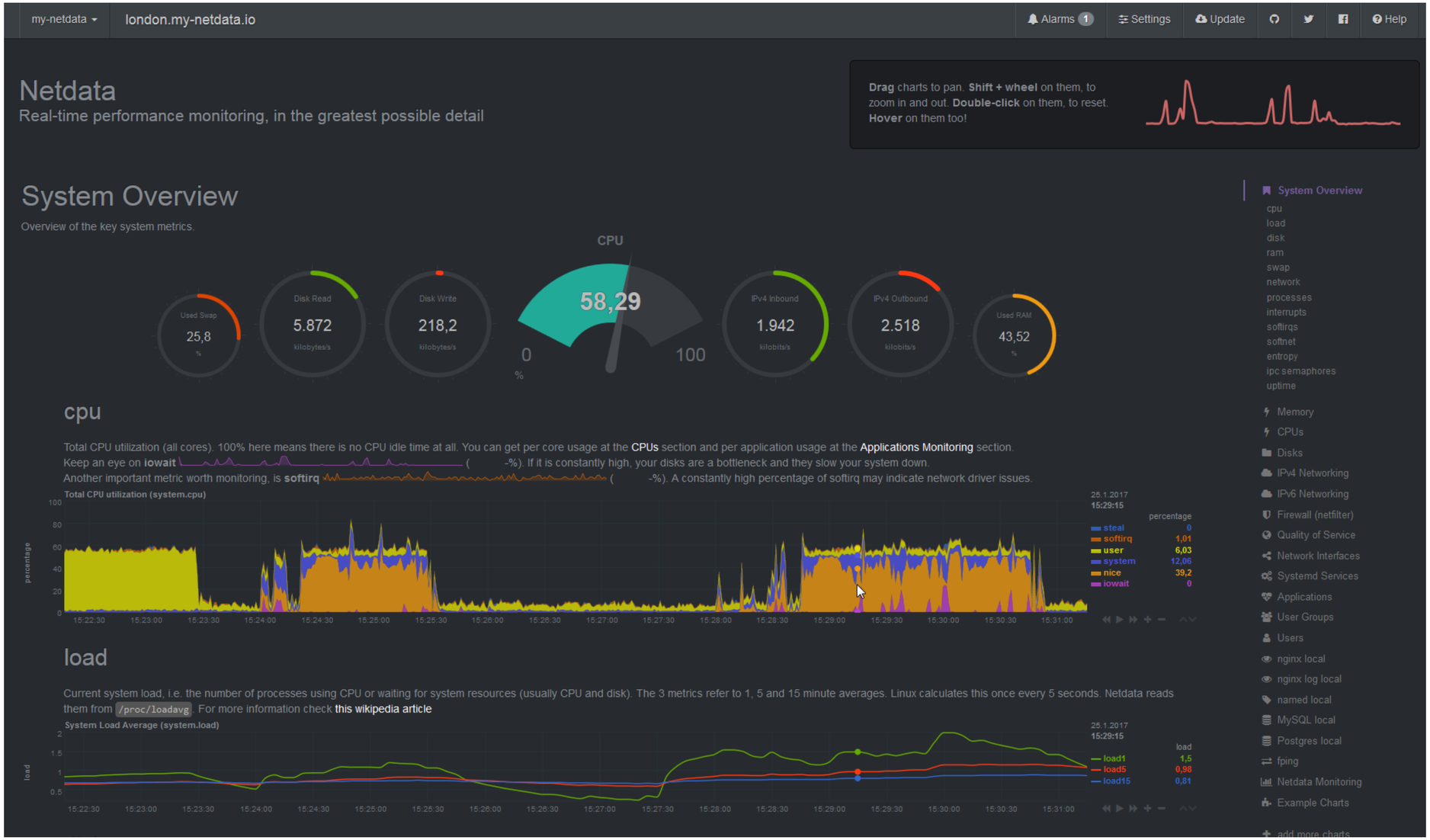
A simple high load system



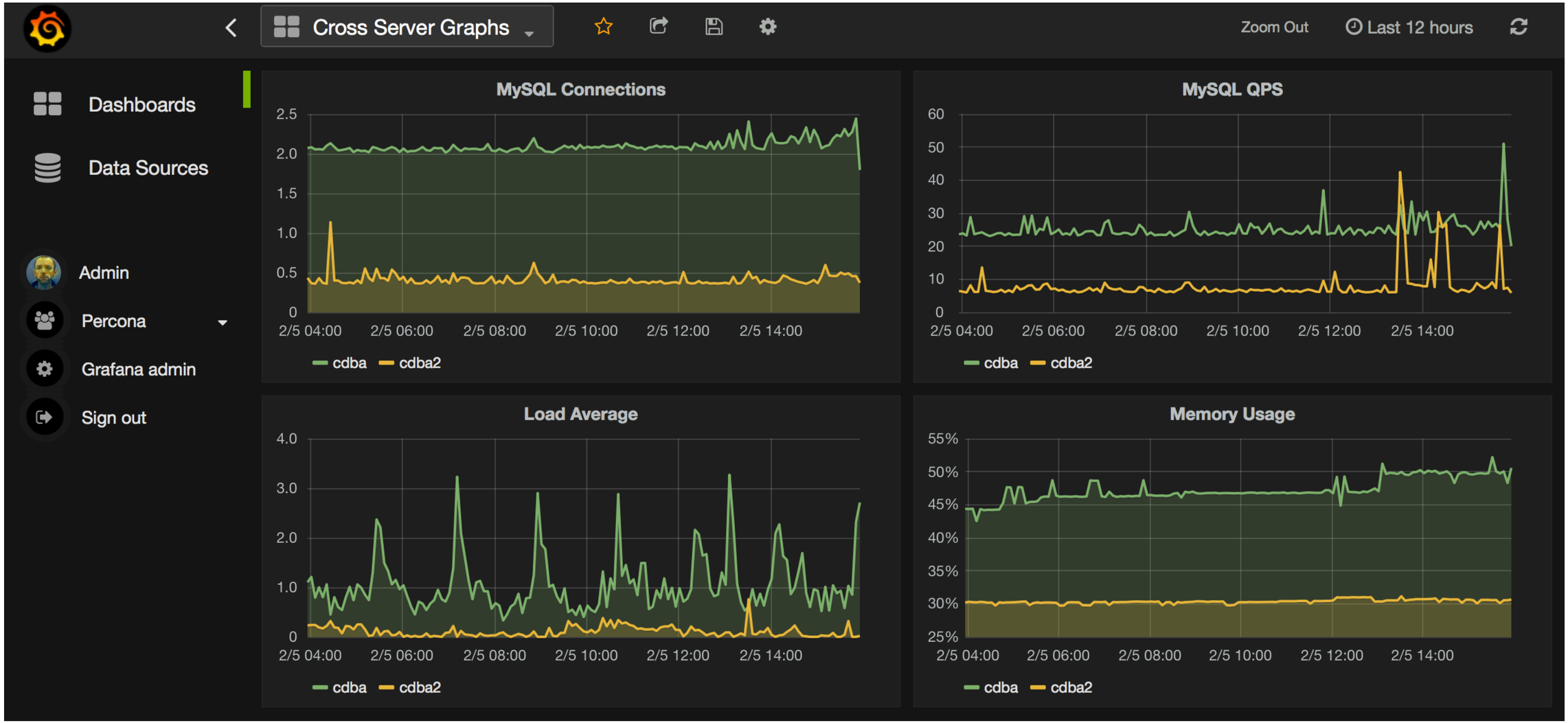
How to build a system can serve 500K CCU?

First at all, we need to know what makes our system slow

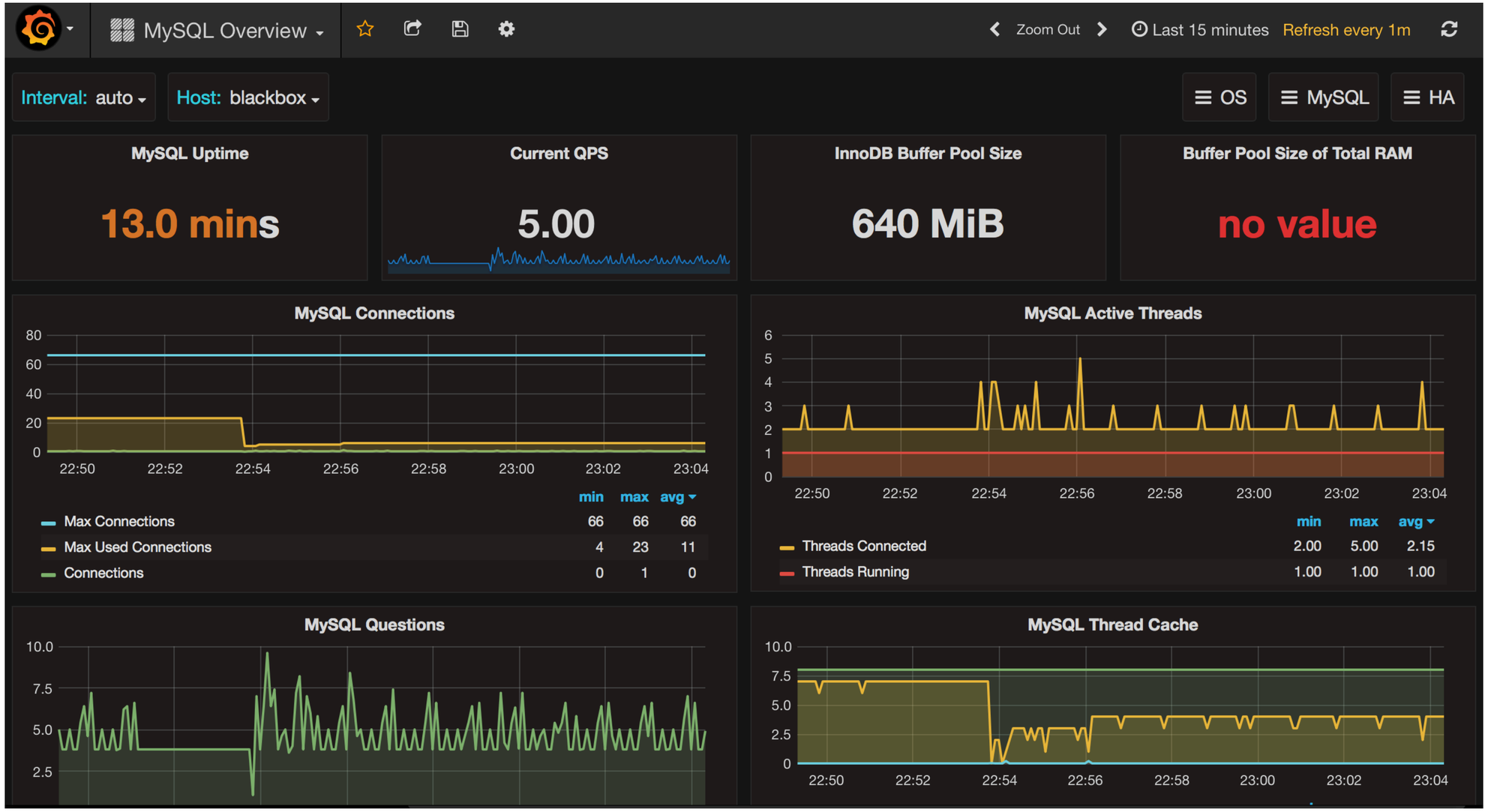
Monitoring & Logging



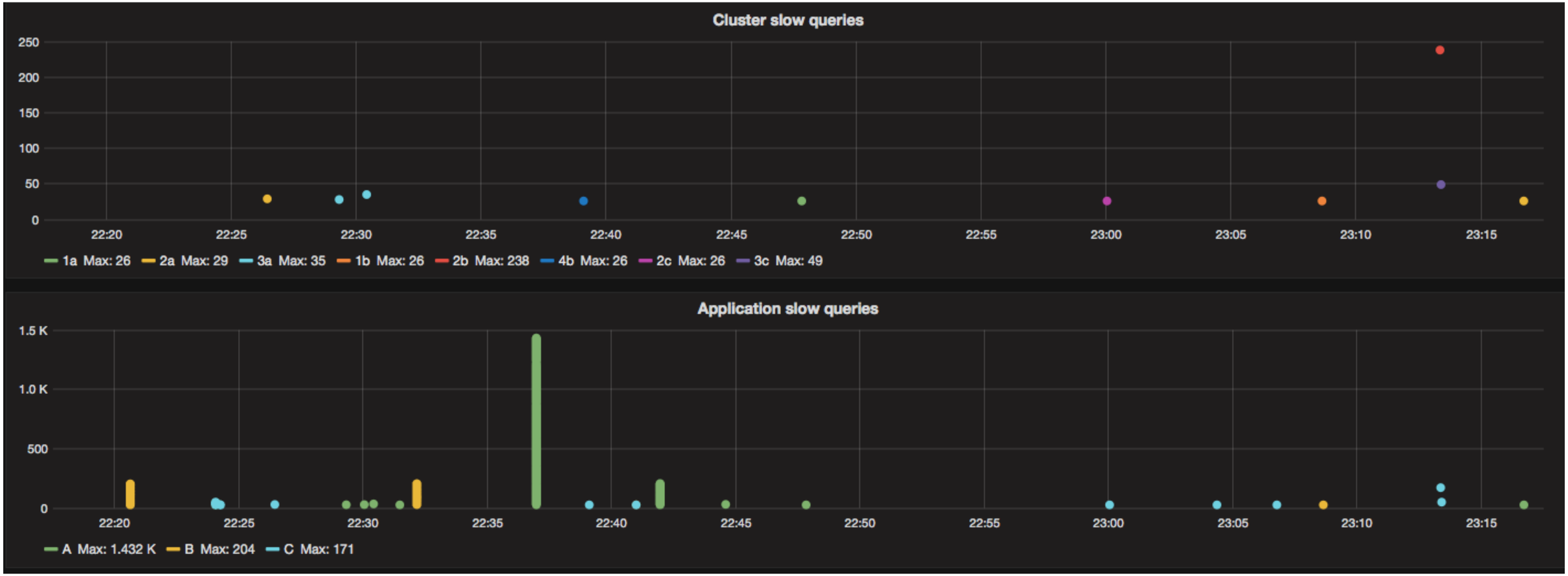
MySQL Monitoring



MySQL Monitoring (cont.)

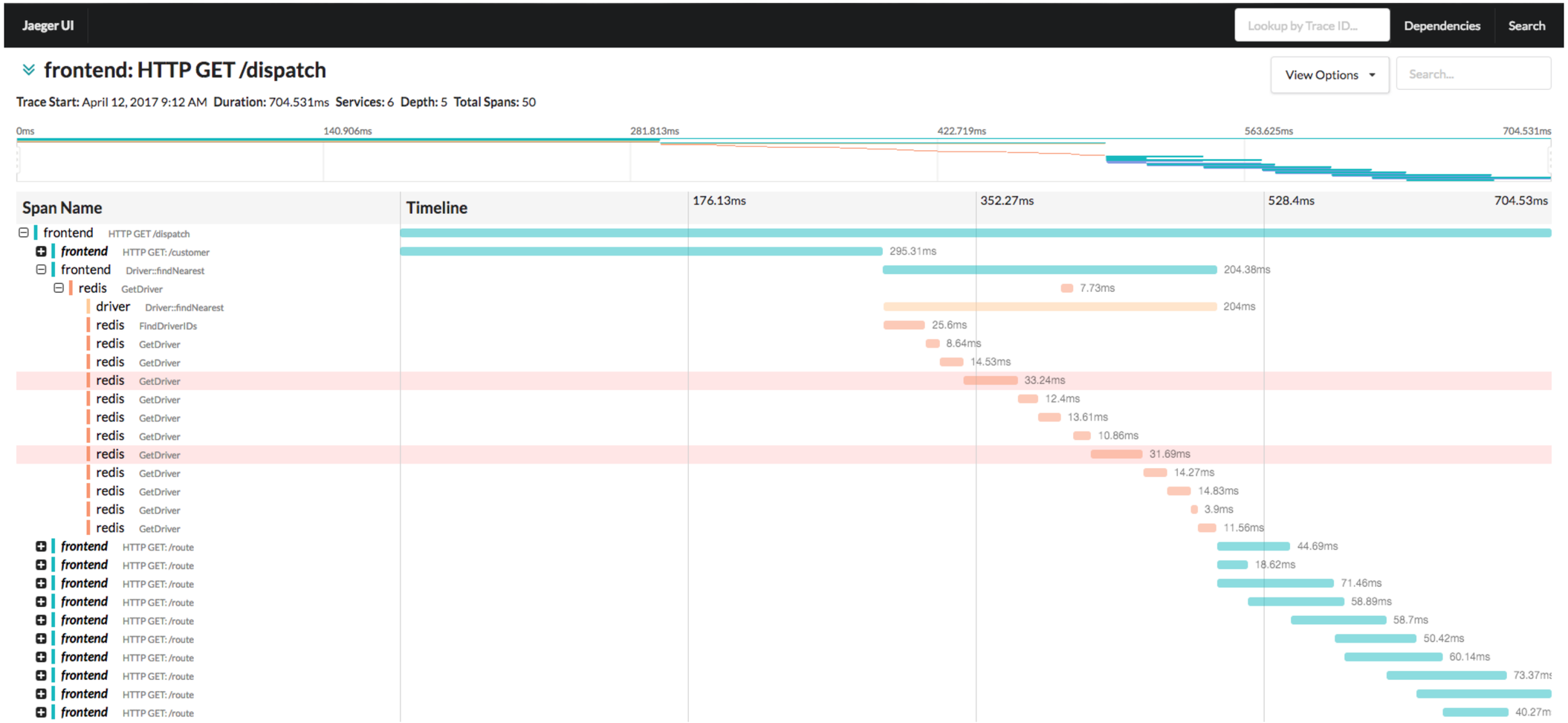


MySQL Slow Query



And many monitor metrics for databases & services

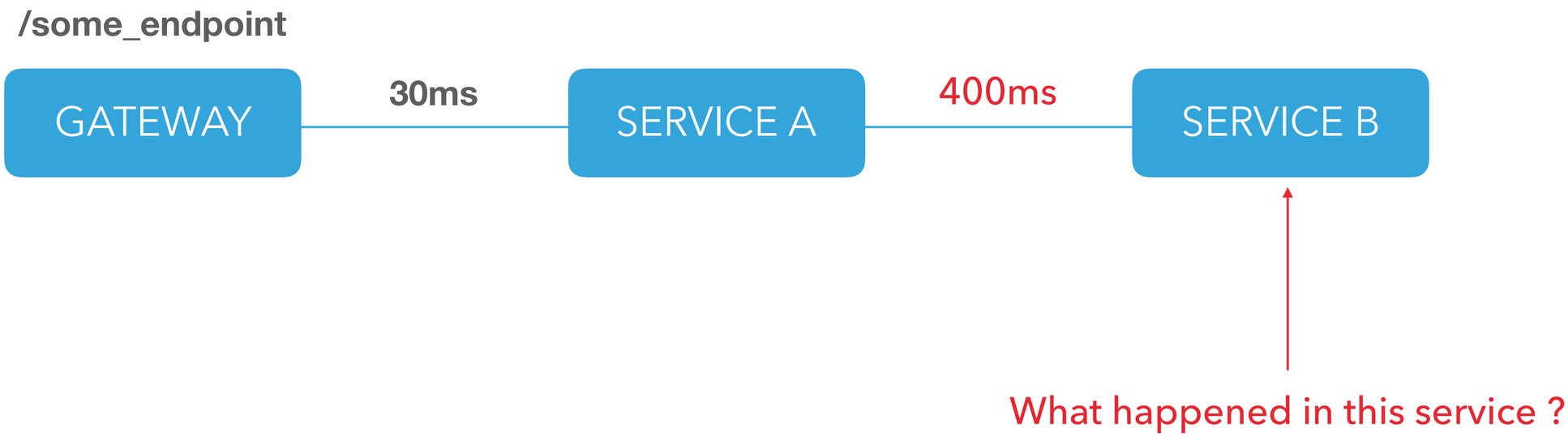
Tracing (Jaeger)



Distributed Tracing

with OpenCensus & Jaeger

Current tracing



We're tracing on each service, but the details in each !!!

Tracing on each service manually

SERVICE B

```
func doSomething() {
    funcTimer := timer.Start("doSomething")
    defer funcTimer.Stop()
    logicATimer := timer.Start("logicA")
    logicA()
    logicATimer.Stop()
    logicBTimer := timer.Start("logicA")
    logicB()
    logicBTimer.Stop()
}
```

Tracing on each service manually (cont.)

SERVICE B

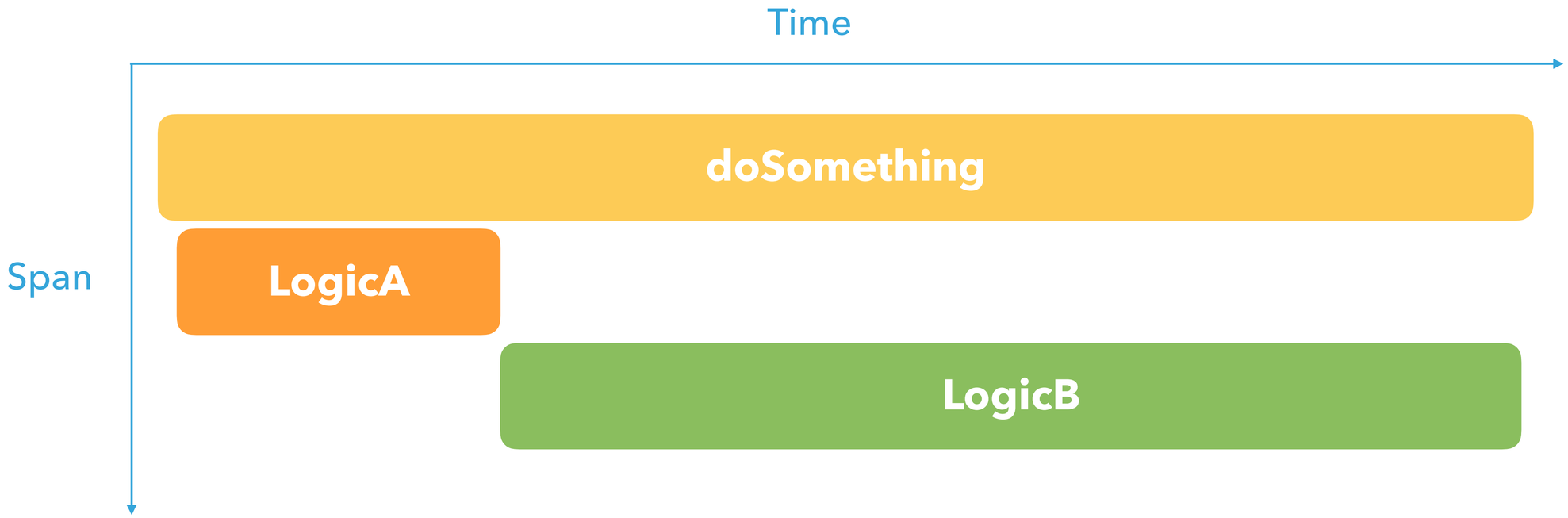
On Console:

doSomething	400ms
logicA	100ms
logicB	300ms

```
func doSomething() {
    funcTimer := timer.Start("doSomething")
    defer funcTimer.Stop()
    logicATimer := timer.Start("logicA")
    logicA()
    logicATimer.Stop()
    logicBTimer := timer.Start("logicA")
    logicB()
    logicBTimer.Stop()
}
```

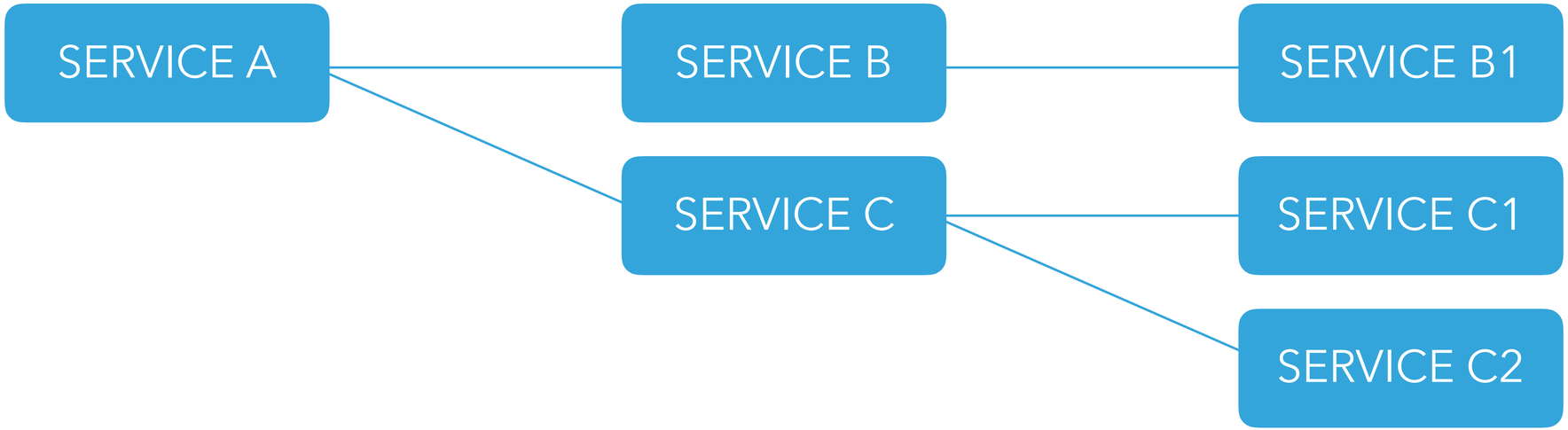
It works, but have some disadvantages

Some disadvantages



Lack of something like a Span Tree as above

Some disadvantages (cont.)



Lack of some component as Collectors for all span data

Some disadvantages (cont.)

SERVICE B

CPU Overhead on production

```
func doSomething() {
    funcTimer := timer.Start("doSomething")
    defer funcTimer.Stop()
    logicATimer := timer.Start("logicA")
    logicA()
    logicATimer.Stop()
    logicBTimer := timer.Start("logicA")
    logicB()
    logicBTimer.Stop()
}
```

That's why we need them

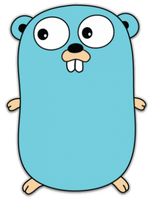
OpenCensus

Jaeger

A single distribution of libraries that automatically collect traces and metrics from your app, display them locally, and send them to any backend. Learn more

- Distributed Trace Collection
- Low Overhead
- Backend Support.

Language Support:



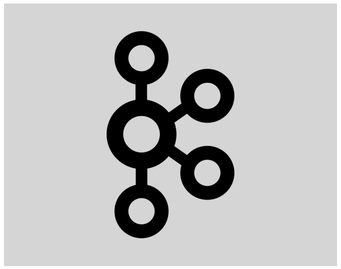

Backend Support:



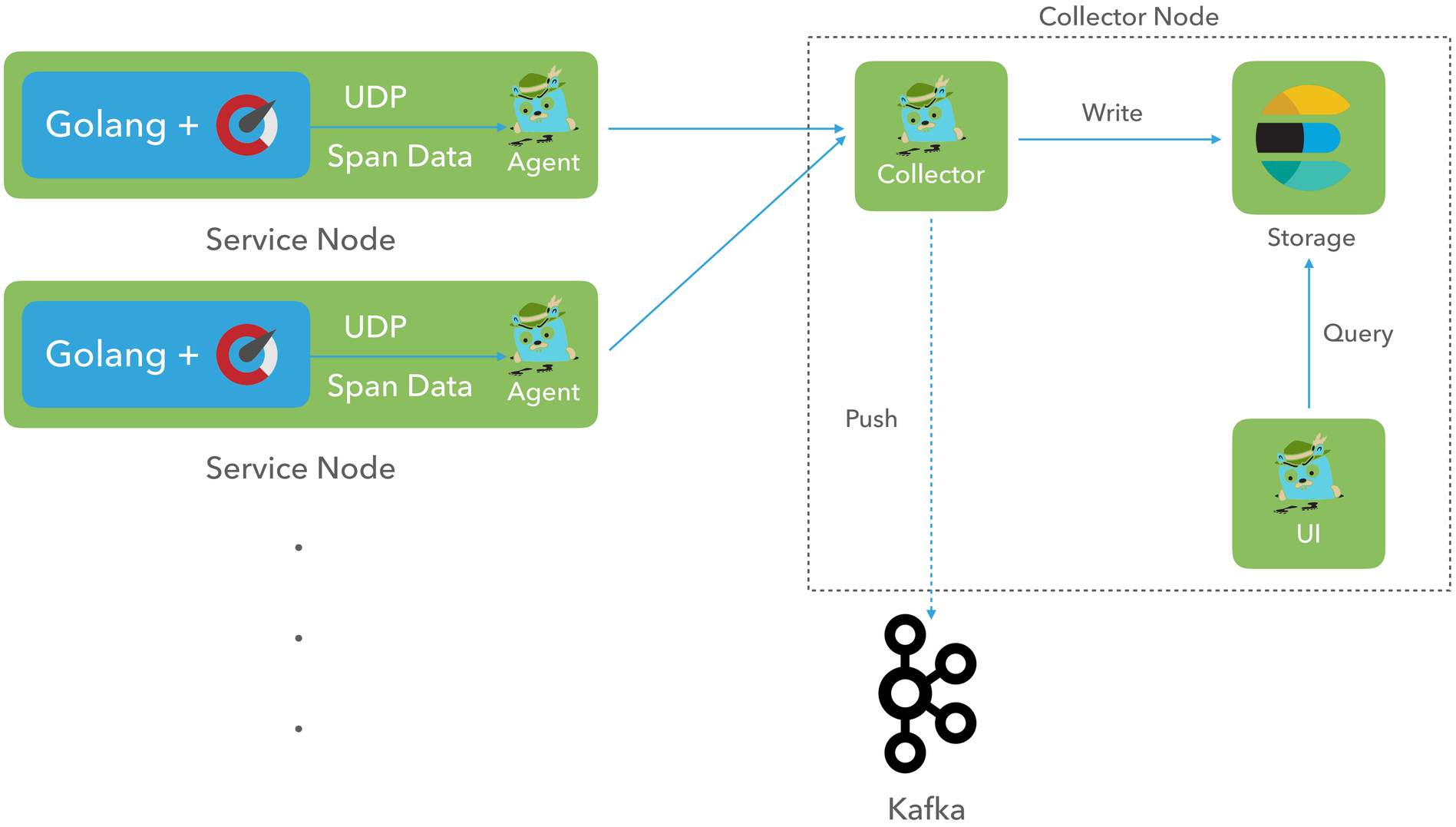

Jaeger is a distributed tracing system released as open source by Uber Technologies. Learn more

- Distributed context propagation
- Distributed transaction monitoring
- Root cause analysis
- Service dependency analysis
- Performance / latency optimization

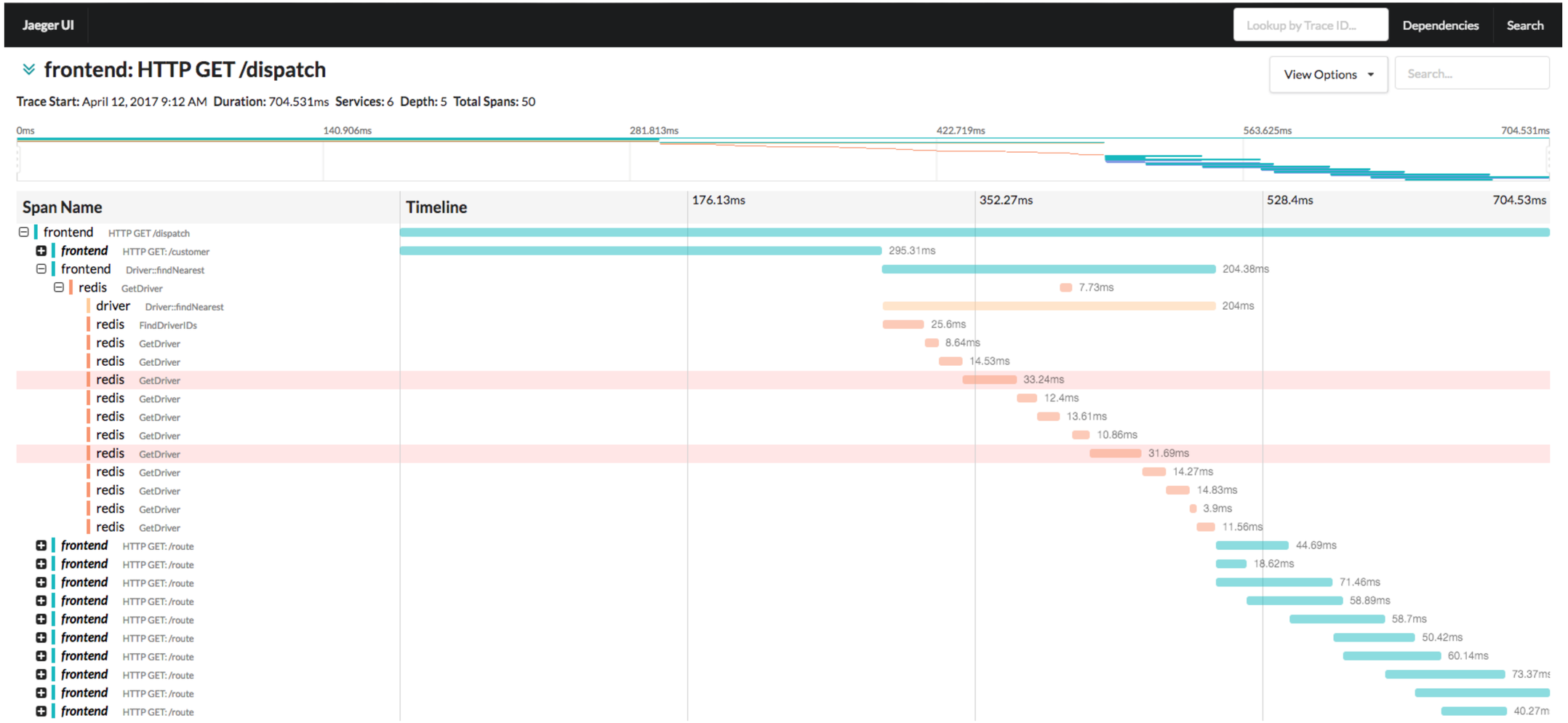
Storage Support:



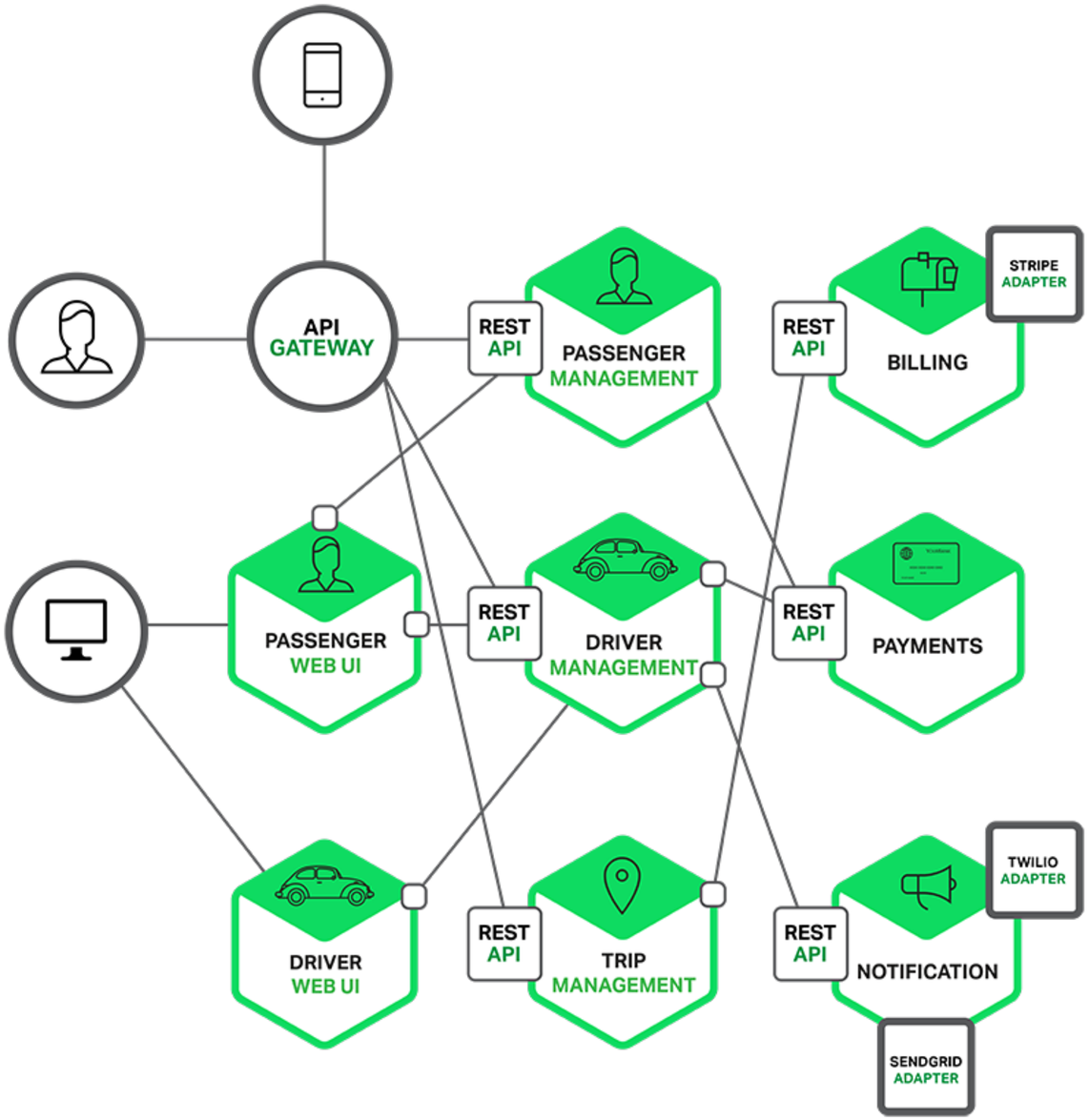
Opensensus & Jaeger Deployment



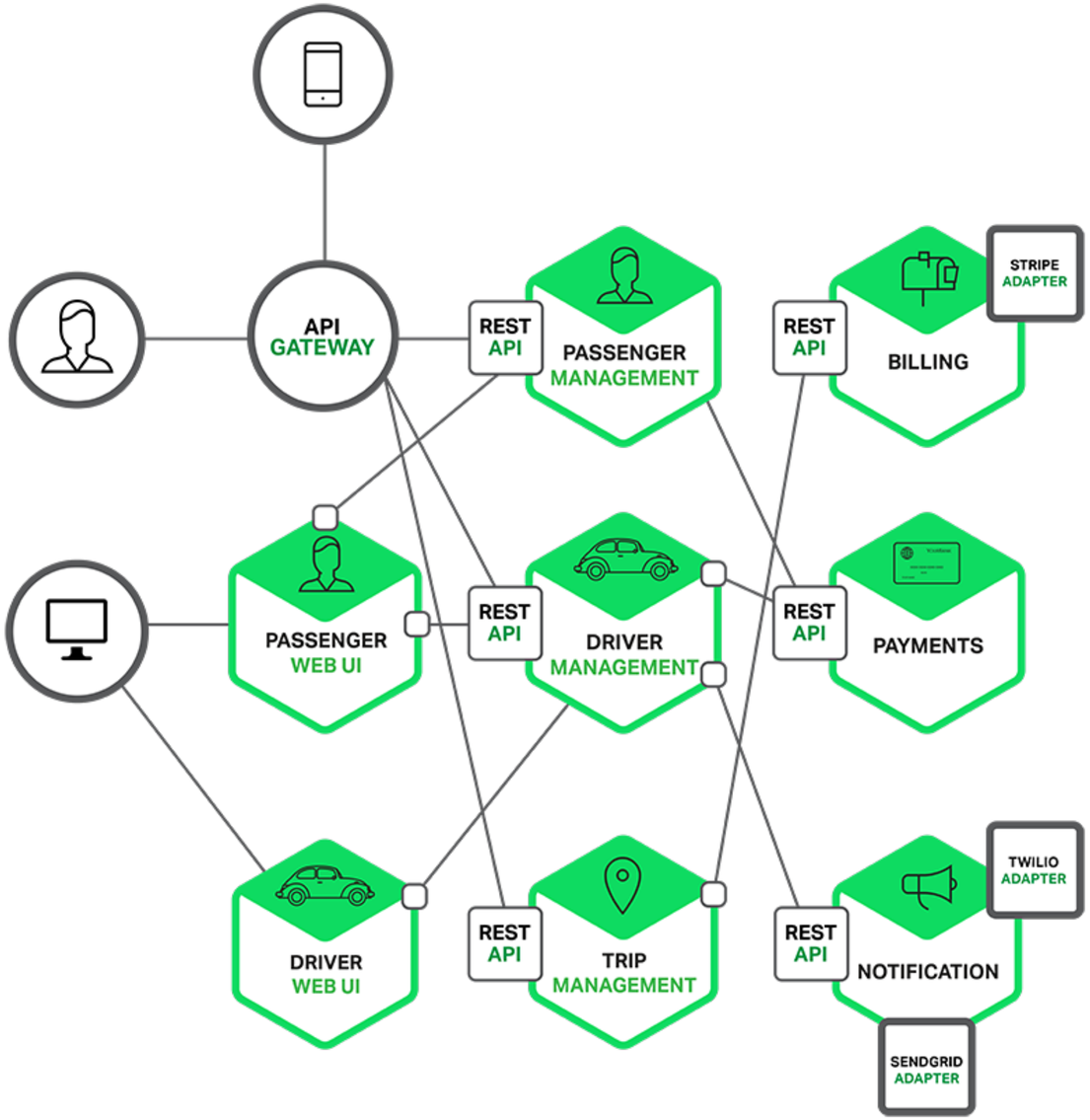
JAEGER UI


Jaeger UI
Lookup by Trace ID...
Dependencies
Search
frontend: HTTP GET /dispatch
View Options
Search...
Trace Start: April 12, 2017 9:12 AM Duration: 704.531ms Services: 6 Depth: 5 Total Spans: 50
0ms
140.906ms
281.813ms
422.719ms
563.625ms
704.531ms
Span Name
Timeline
176.13ms
352.27ms
528.4ms
704.53ms
frontend HTTP GET /dispatch
frontend HTTP GET: /customer
295.31ms
frontend Driver::findNearest
204.38ms
redis GetDriver
7.73ms
driver Driver::findNearest
204ms
redis FindDriverIDs
25.6ms
redis GetDriver
8.64ms
redis GetDriver
14.53ms
redis GetDriver
33.24ms
redis GetDriver
12.4ms
redis GetDriver
13.61ms
redis GetDriver
10.86ms
redis GetDriver
31.69ms
redis GetDriver
14.27ms
redis GetDriver
14.83ms
redis GetDriver
3.9ms
redis GetDriver
11.56ms
frontend HTTP GET: /route
44.69ms
frontend HTTP GET: /route
18.62ms
frontend HTTP GET: /route
71.46ms
frontend HTTP GET: /route
58.89ms
frontend HTTP GET: /route
58.7ms
frontend HTTP GET: /route
50.42ms
frontend HTTP GET: /route
60.14ms
frontend HTTP GET: /route
73.37ms
frontend HTTP GET: /route
frontend HTTP GET: /route
40.27m

Microservice



Microservice



Microservices remove dependent

But add more complex for system

A use case from Sendo

Brief history of Sendo system

In 2012: Sendo system was based on Magento

Brief history of Sendo system (cont)

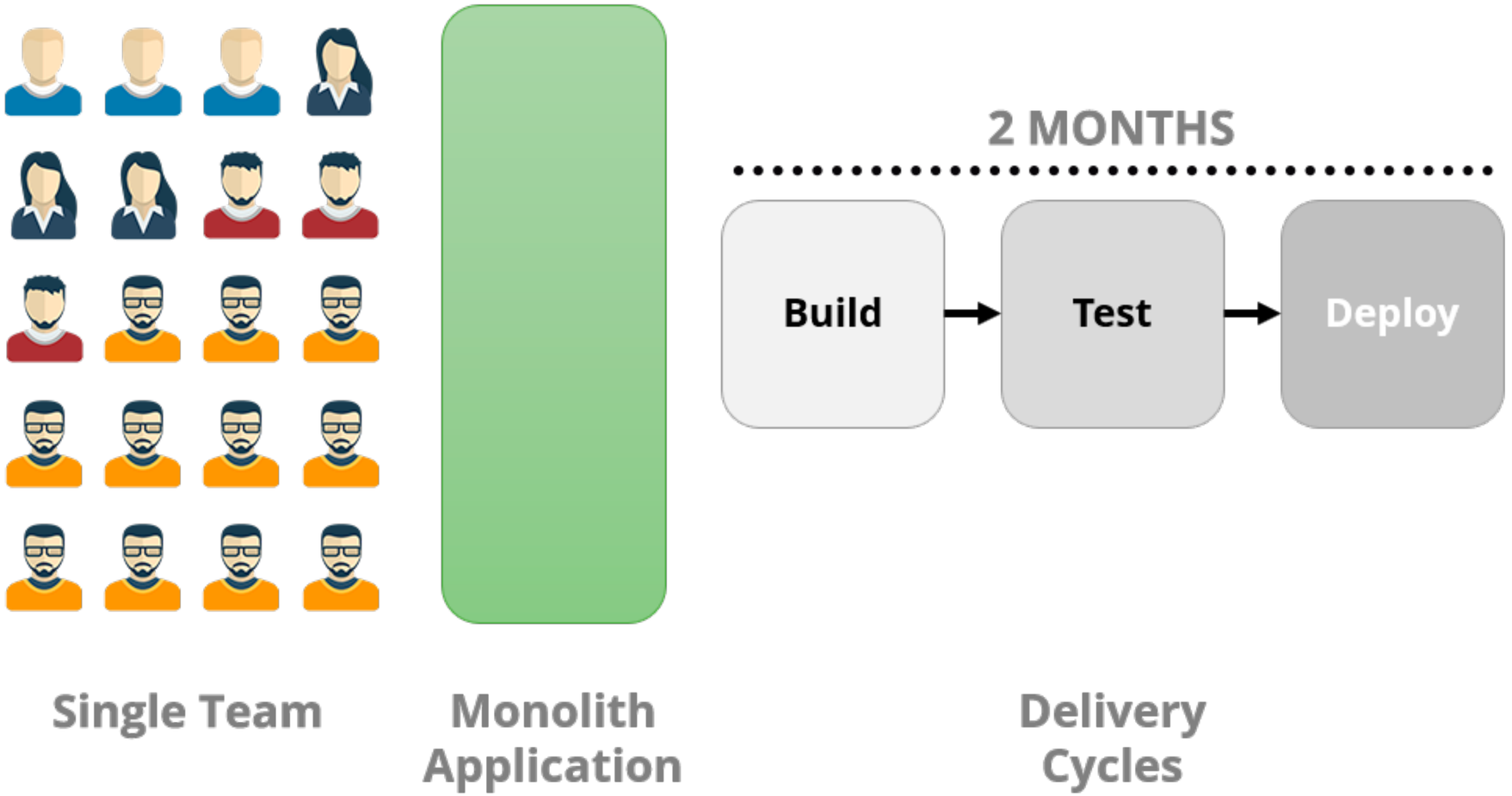
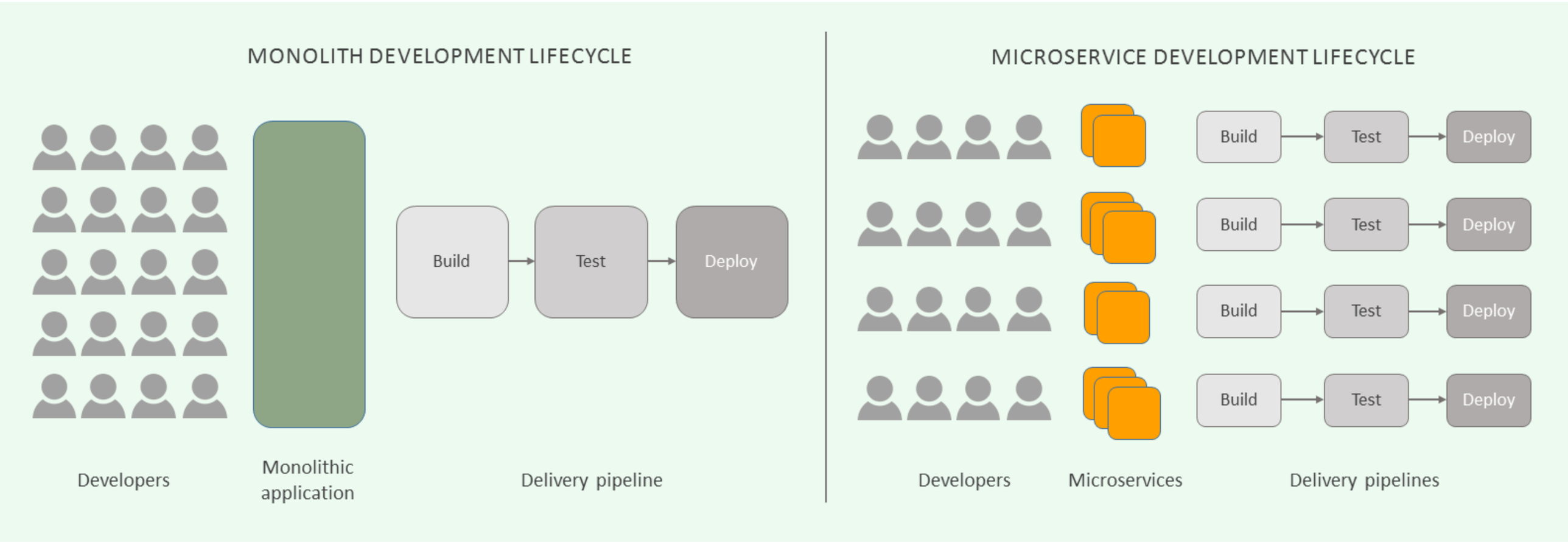


Image Source: Beesion Technologies

Brief history of Sendo system (cont)



In 2016: Sendo started to use microservices architecture

Image Source: DZone

Brief history of Sendo system (cont)

We use multiple languages to build the services

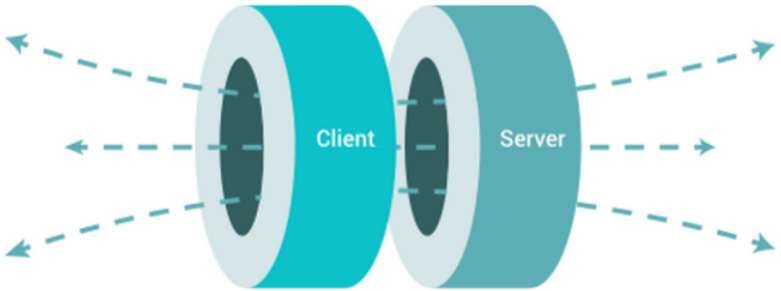
Image Source: Weaveworks

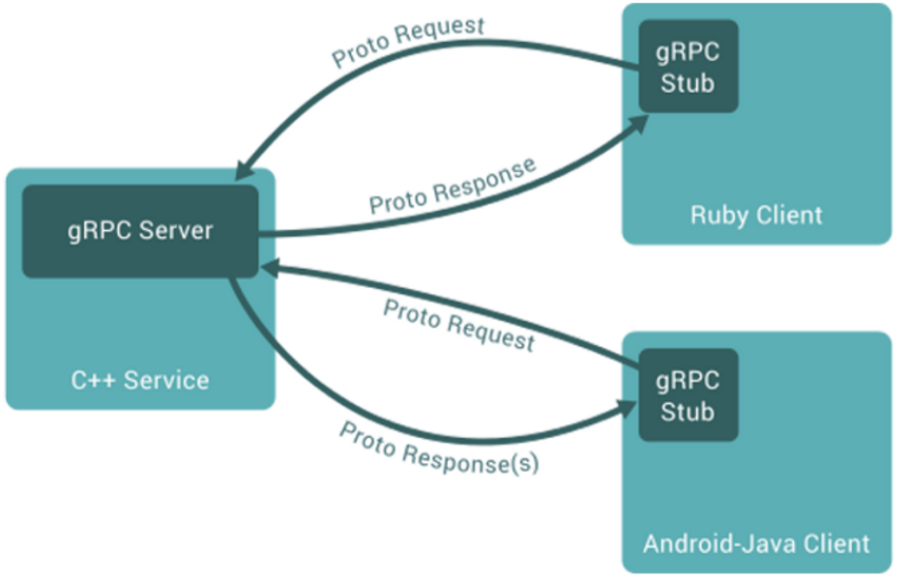
Problems

- High latency and low throughput
- Duplication data structures
- Hard to failure recovery on distributed system

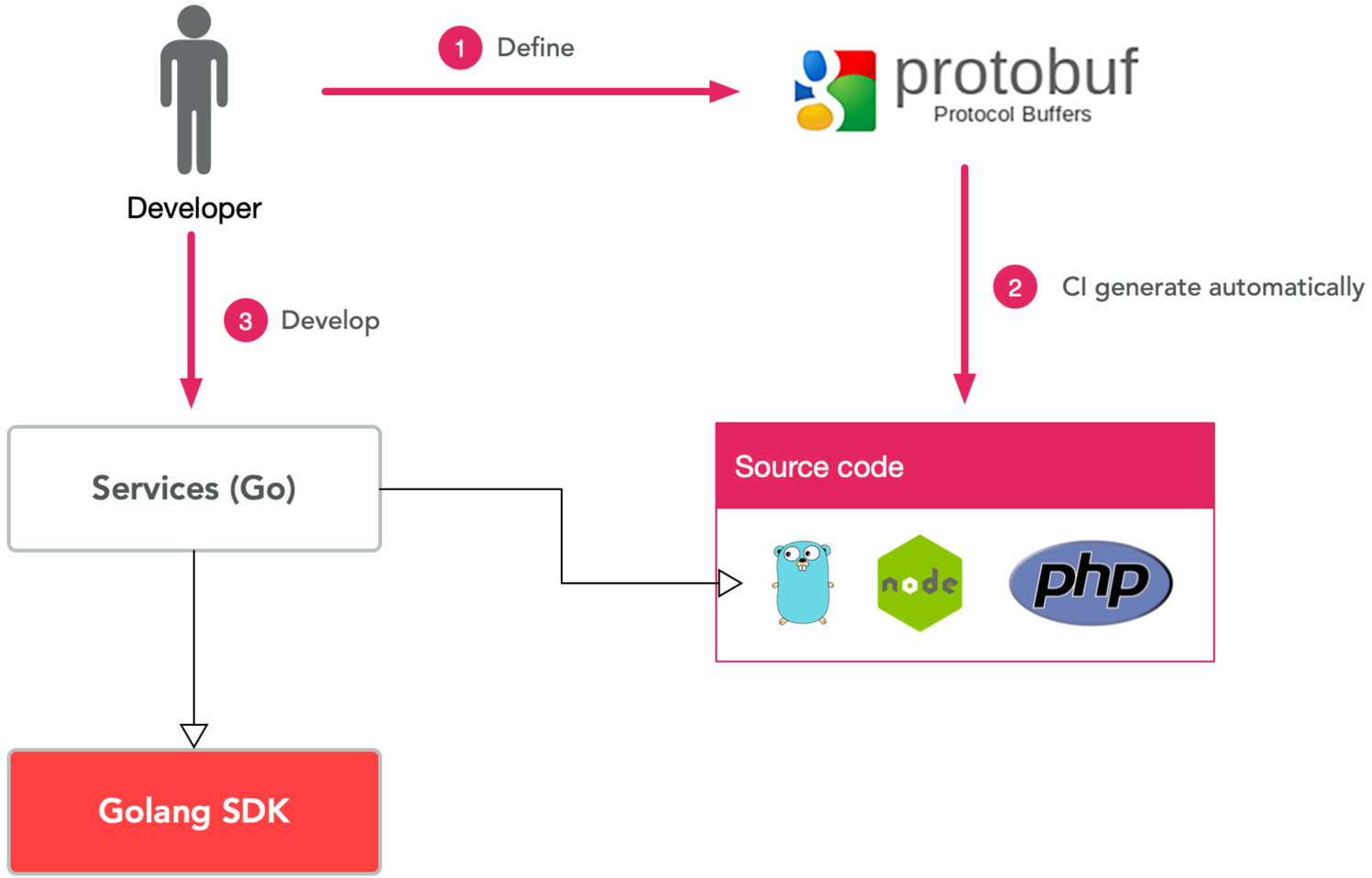
We decided to migrate almost source codes to

Microservices with gRPC and Protobuf





Developer Workflow



A demo Protobuf file

```
service NoteService {
  // add new note
  rpc Add(NoteAddReq) returns (Note) {
    option (google.api.http) = {
      post: "/demo/notes"
      body: "*"
    };
  }
  // list note
  rpc List(NoteListReq) returns (Notes) {
    option (google.api.http) = {
      get: "/demo/notes"
    };
  }
  // ...
}
```

A demo Protobuf file (cont.)

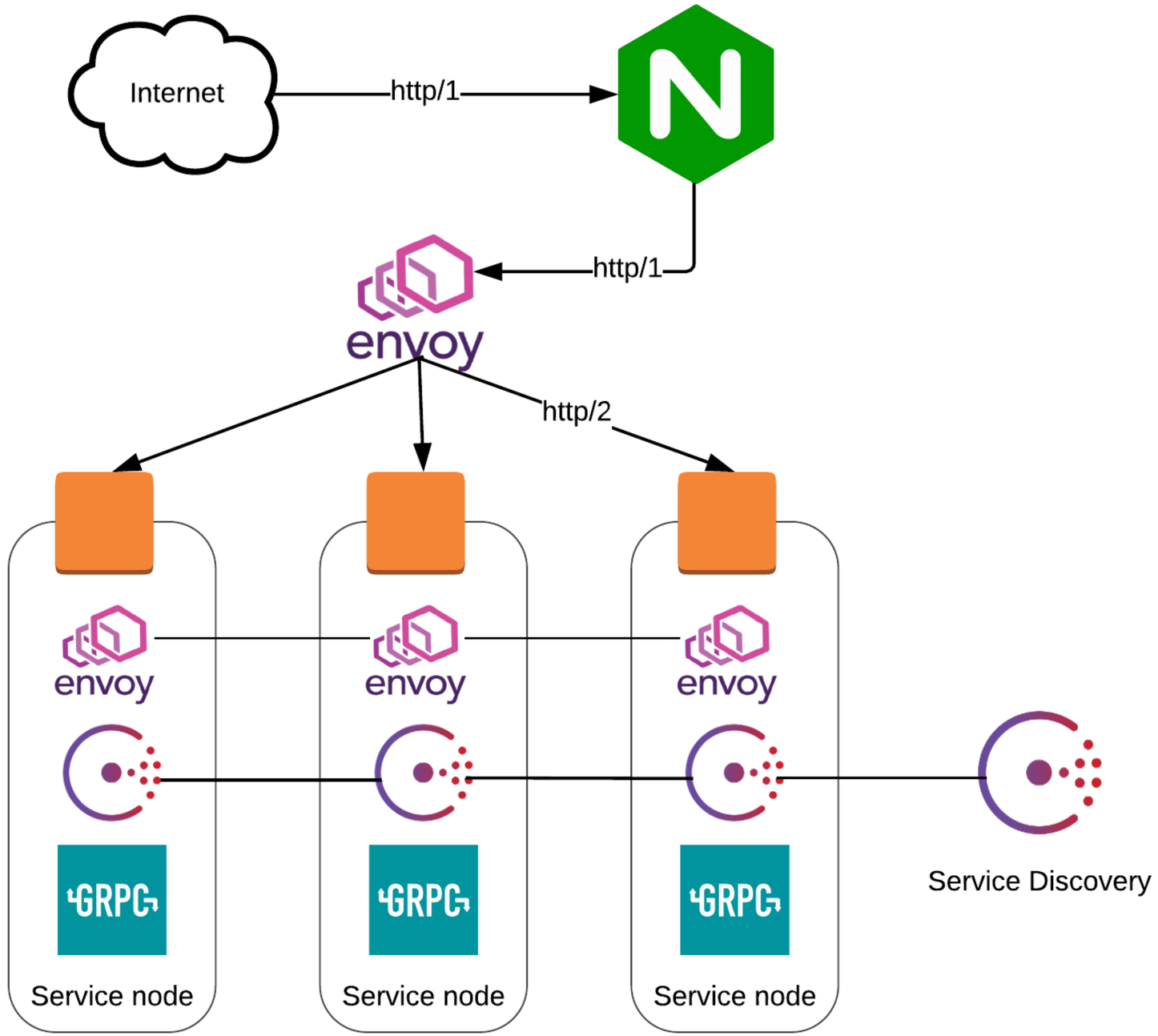
```
42  message Note {
43    int64 id = 1;
44    string text = 2;
45    google.protobuf.Timestamp created = 3;
46    google.protobuf.Timestamp modified = 4;
47  }
48
49  message Notes {
50    int64 total = 1;
51    repeated Note notes = 2;
52  }
53
54  message NoteListReq {
55    base.Pagination pagination = 1;
56  }
```

Generated files

Name	Last commit
..	
demo.pb.go	update build for v1.2.50
demo.pb.gw.go	update build for v1.1.88
demo.pb.validate.go	update build for v1.1.88
demo.sendo.go	update build for v1.2.50

Sendo Microservices

- Service Mesh with Envoy
- Service Discovery with Consul
- Load balancing with Nginx and Envoy
- Very high throughput with Protobuf



Problems solved so far

- High latency and low throughput
  Use Go and gRPC for Inter-service communication.

- Duplication data structures
  Use protobuf for generating data structures

- Hard to failure recover on distributed system
  The hardest part. How ?

Control Plane with Istio

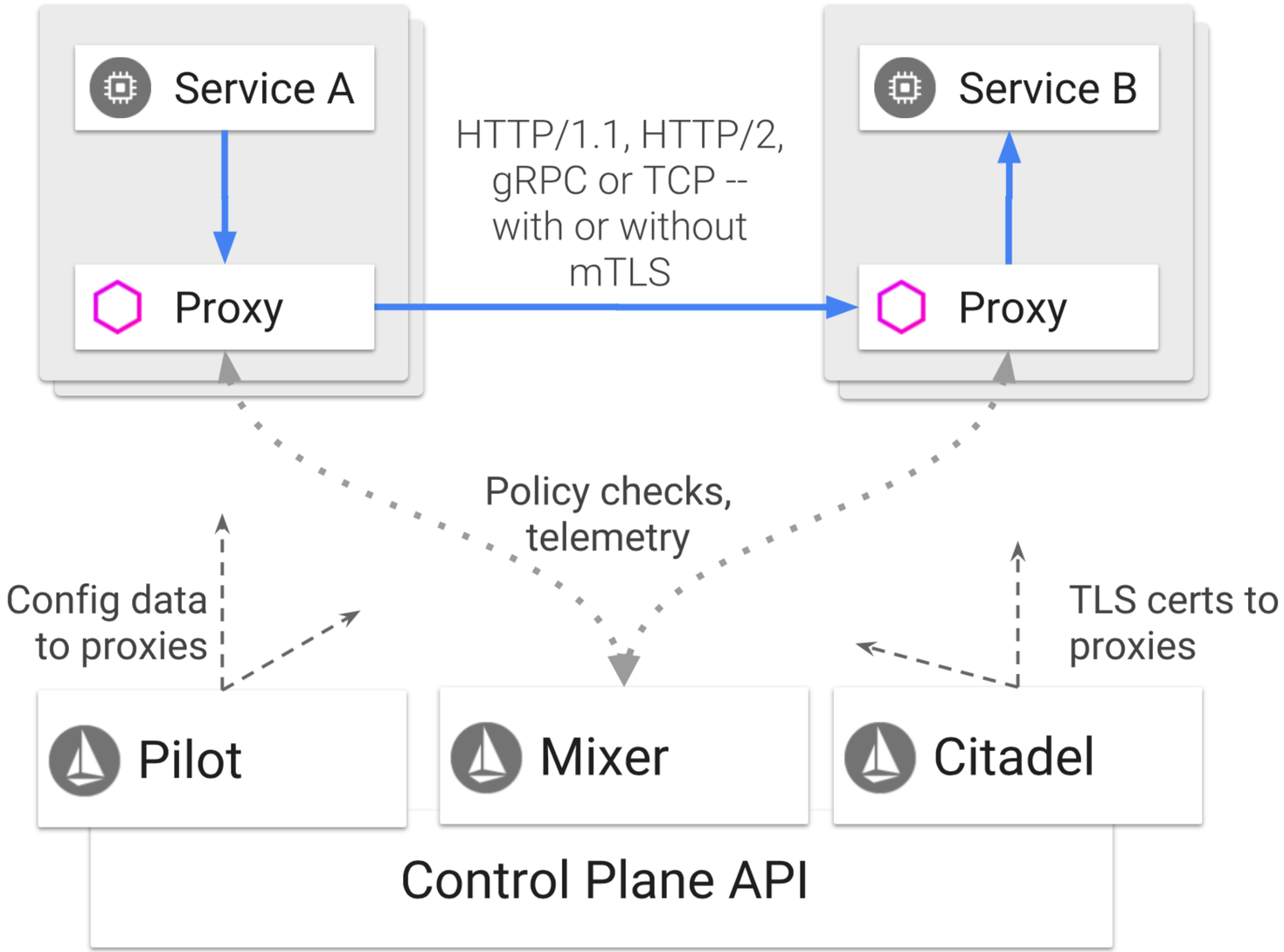


Image Source: istio.io

Dynamic Routing

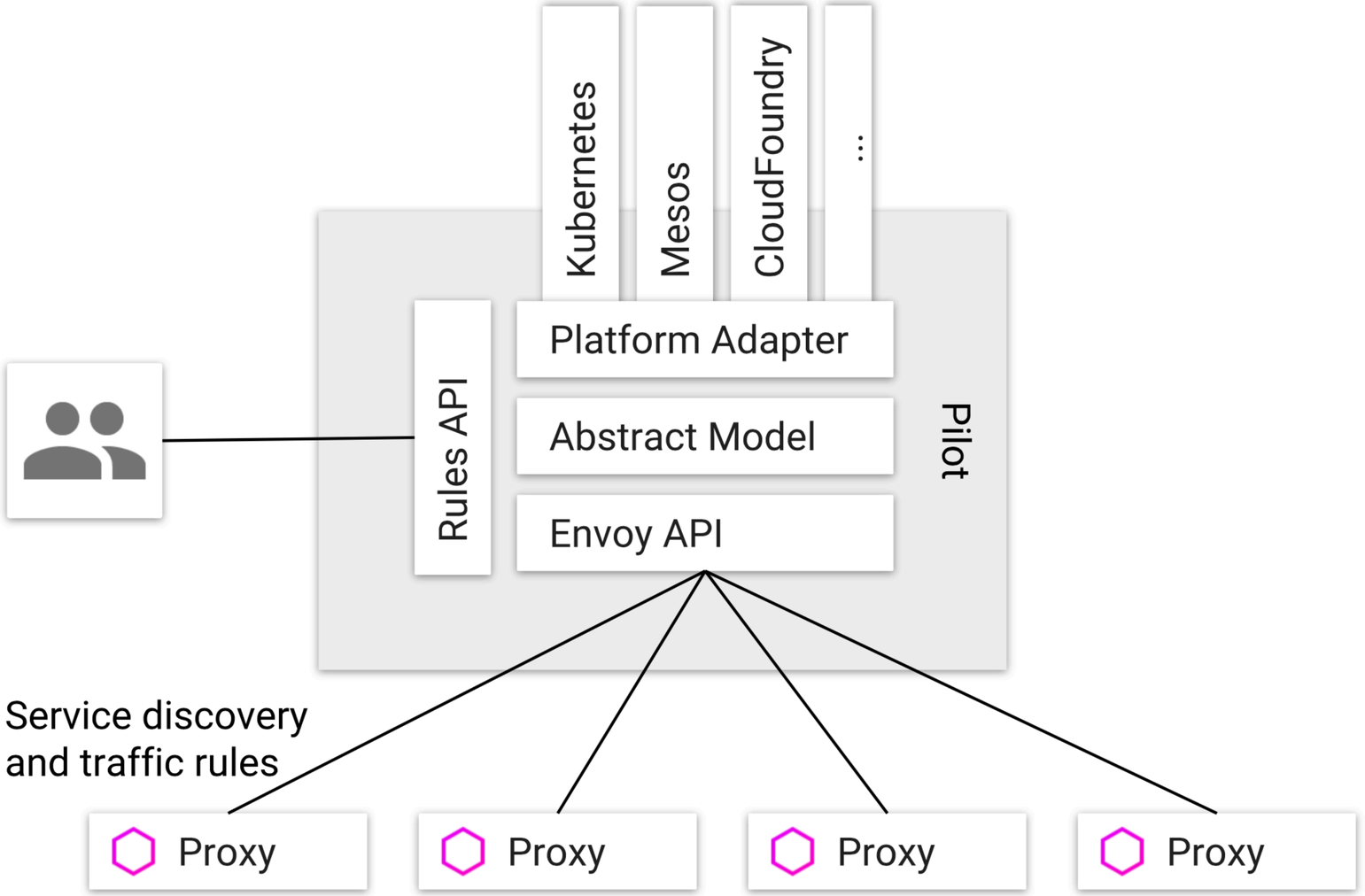


Image Source: istio.io

Istio Mixer

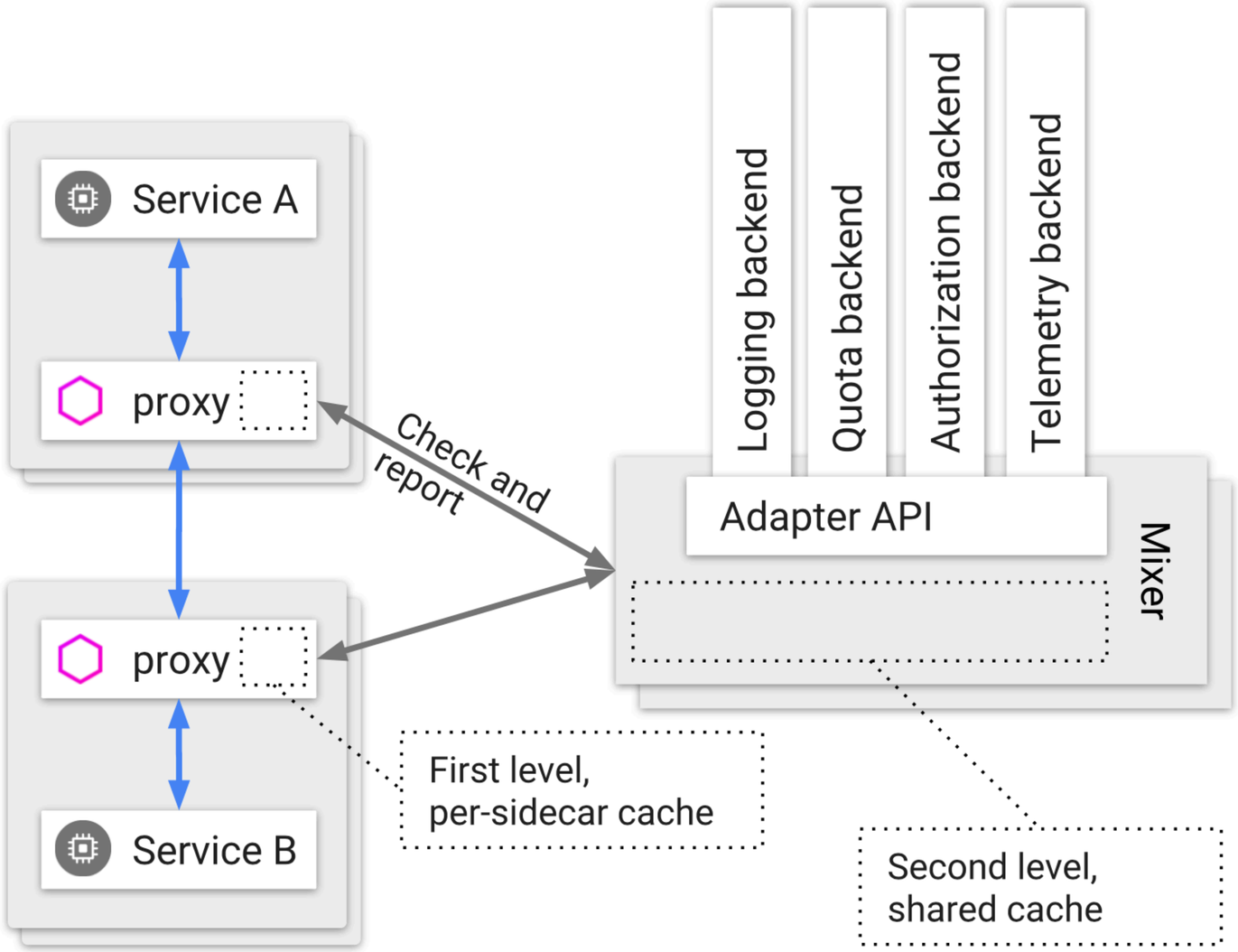
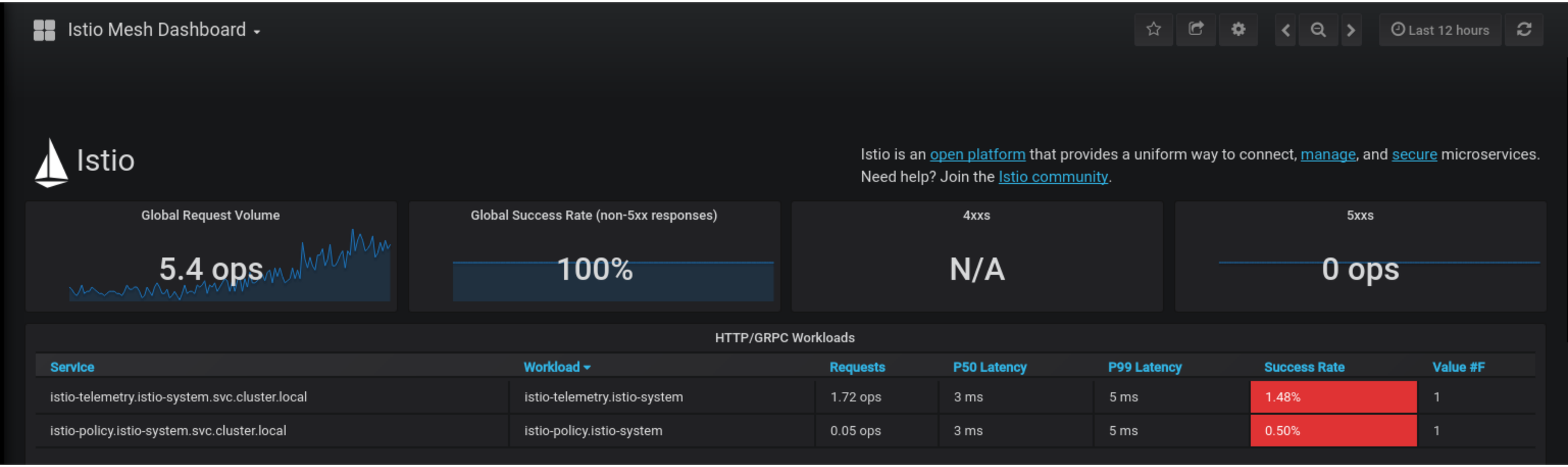
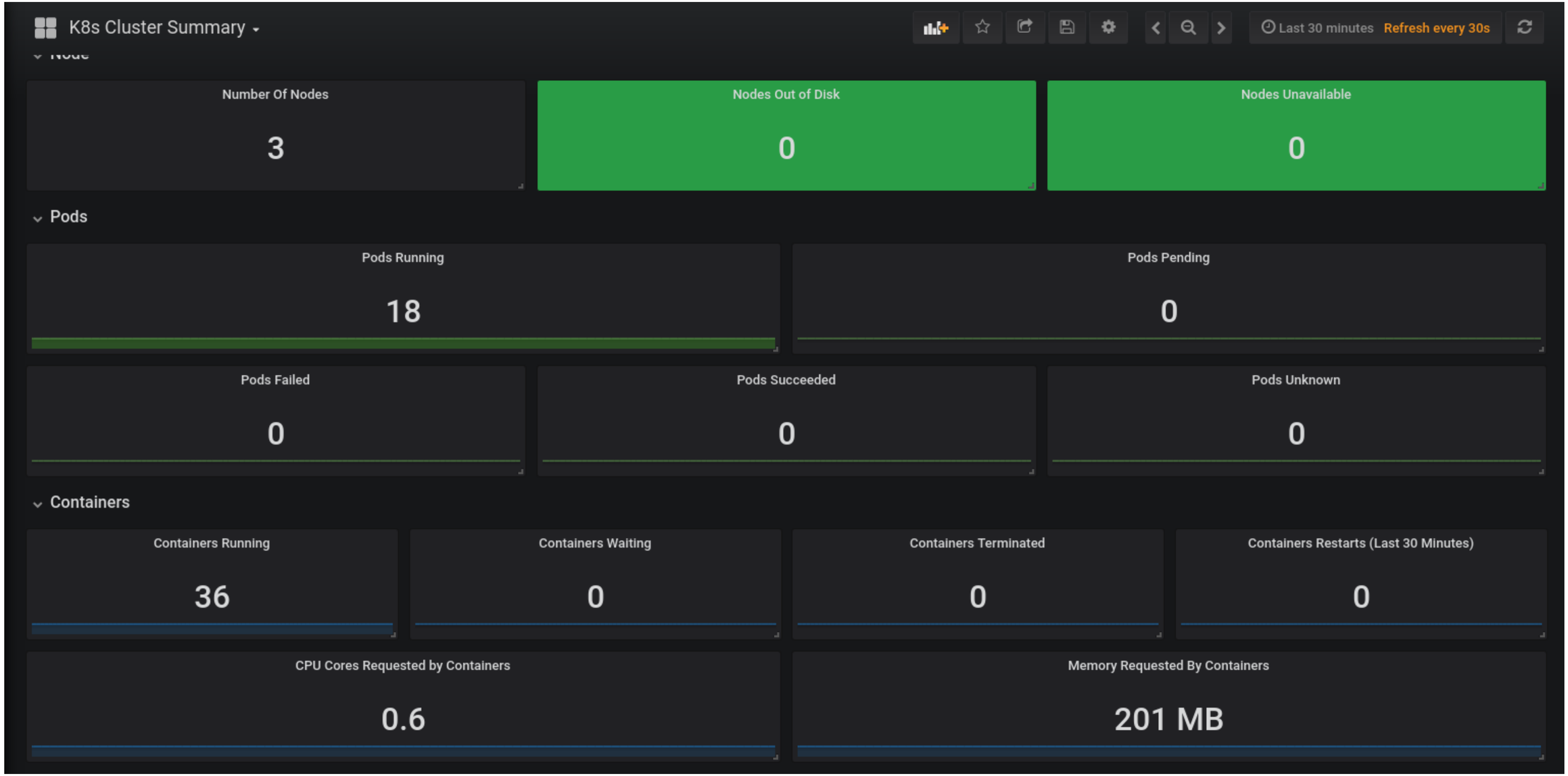


Image Source: istio.io

Istio

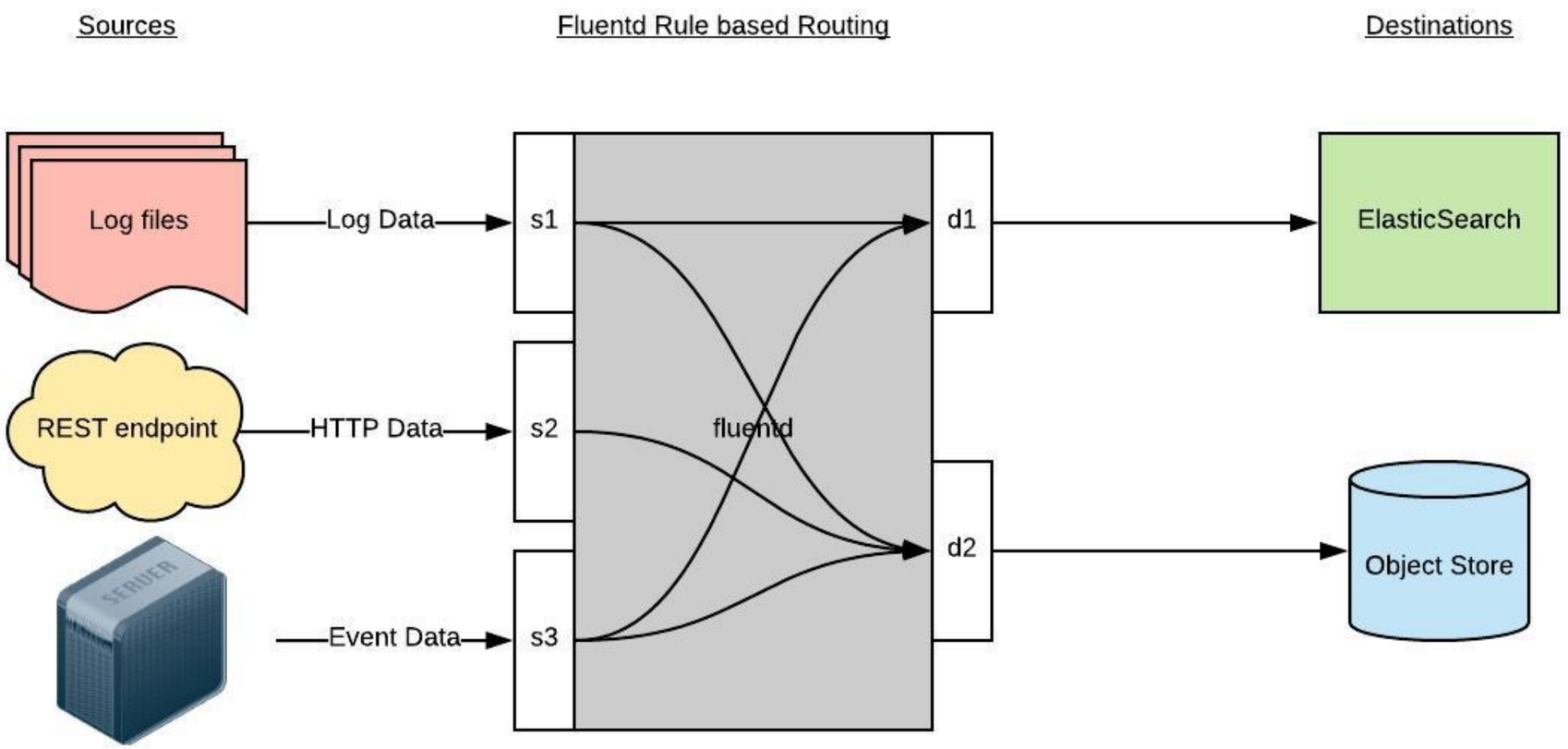

Istio Mesh Dashboard
Last 12 hours
Istio
Istio is an open platform that provides a uniform way to connect, manage, and secure microservices.
Need help? Join the Istio community.
Global Request Volume
5.4 ops
Global Success Rate (non-5xx responses)
100%
4xxs
N/A
5xxs
0 ops
HTTP/GRPC Workloads
Service
Workload
Requests
P50 Latency
P99 Latency
Success Rate
Value #F
istio-telemetry.istio-system.svc.cluster.local
istio-telemetry.istio-system
1.72 ops
3 ms
5 ms
1.48%
1
istio-policy.istio-system.svc.cluster.local
istio-policy.istio-system
0.05 ops
3 ms
5 ms
0.50%
1

Kubernets Summary

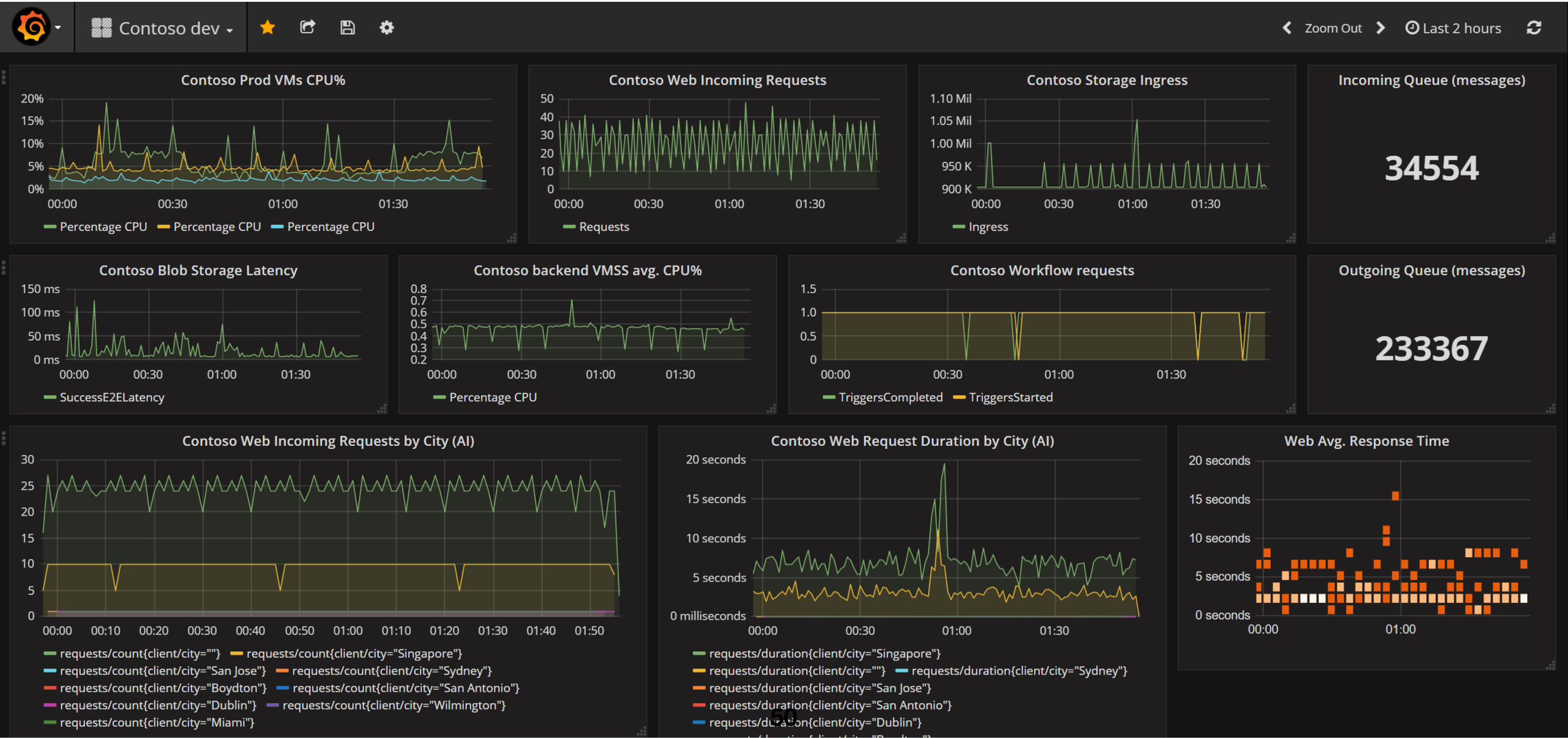


Kubernets Summary (cont)

Logging with Fluentd & ElasticSearch



Visualize logging data: Grafana



Problems solved so far

- High latency and low throughput
  Use Go and gRPC for Inter-service communication.

- Duplication data structures
  Use protobuf for generating data structures

- Hard to failure recover on distributed system
  Data Plane and Control Plane: **Istio**, **Kubernetes**
  Logging system: **Fluentd**, **Elastic Search**, **Grafana**
  Monitoring system: **Netdata**, **Graphite**, **Grafana, Prometheus**
  Distributed Tracing: **Jeager [,Opencensus]**

Thank you