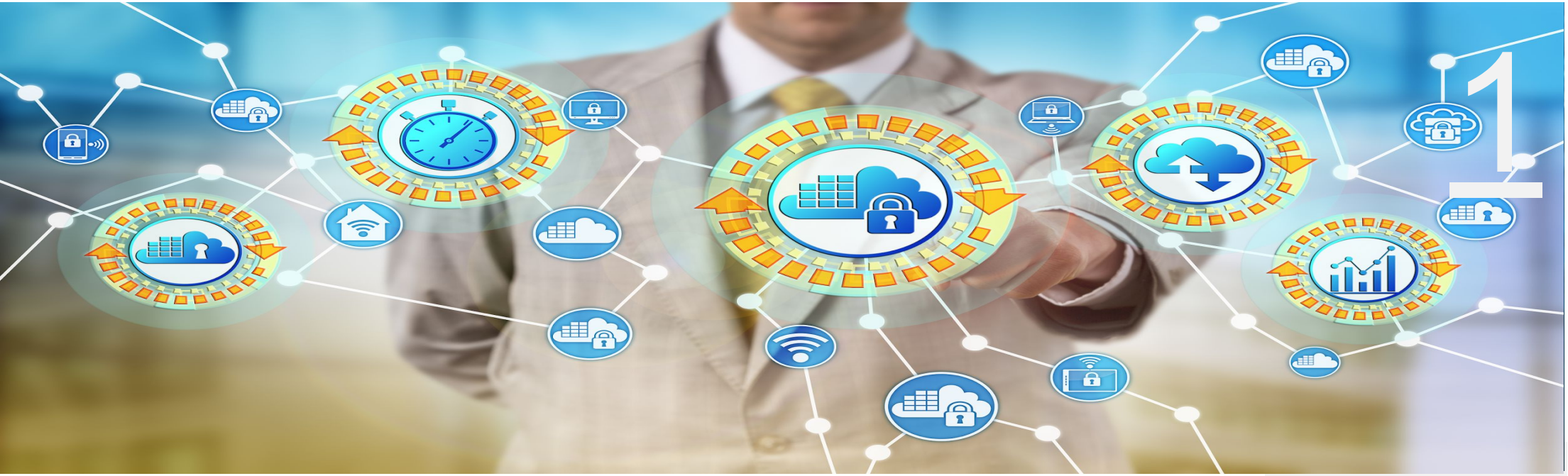
Microservice Application Architecture…



Day 1 ReCap

- Monolithic Overview,Pros & Cons
- Microservices Overview,Pros & Cons
- The 12 Factor Apps
- Microservices Challenges
- Building Blocks of Microservices
- Docker Architecture
- Docker Compose

- Contracts, Cohesion, Coupling, Observability, Bounded Contexts
- Service Decomposition
- Microservice Boundaries
- Microservices Communication
- Message Patterns - Pub/Sub,Anycast,Unicast, Multicast
- Message Brokers

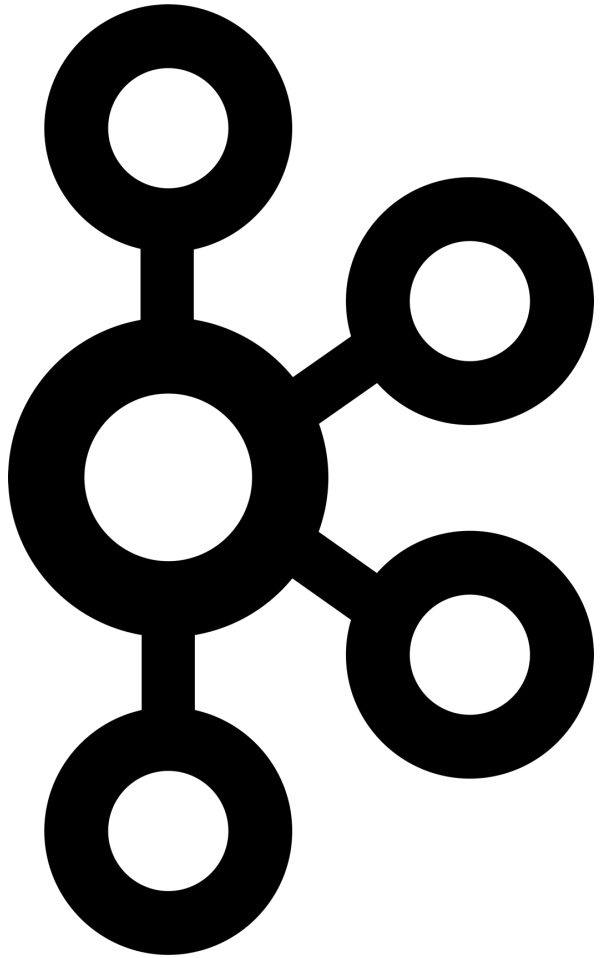
APACHE KAFKA

Kafka® is used for building real-time data pipelines and streaming apps.

APACHE KAFKA

It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

APACHE KAFKA



PUBLISH & SUBSCRIBE

- Read and write streams of data like a messaging system.

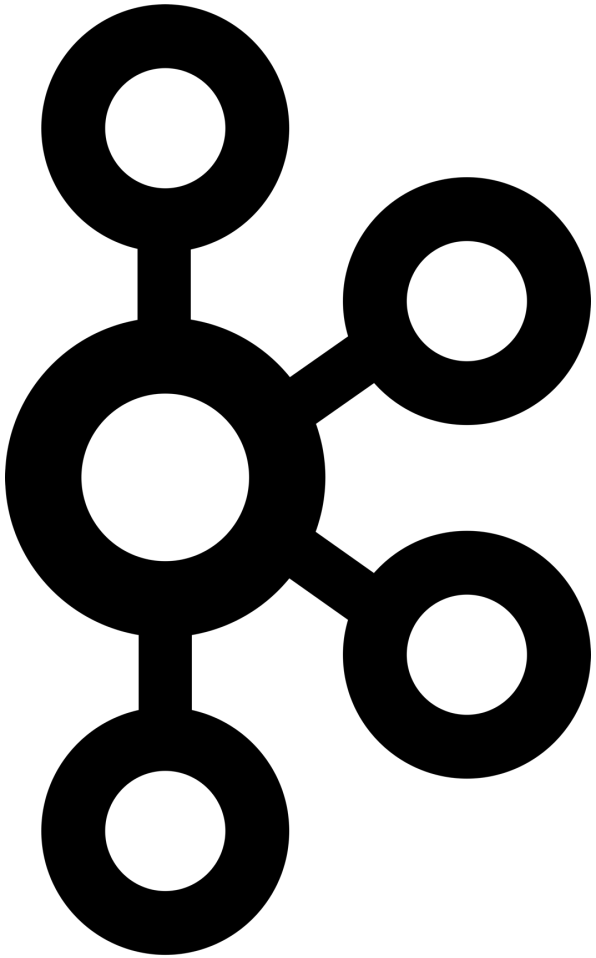
PROCESS

- Write scalable stream processing applications that react to events in real-time.

STORE

- Store streams of data safely in a distributed, replicated, fault-tolerant cluster.

APACHE KAFKA



- Messages are stored in flat files
- Consumers able to ask messages that are based on an offset
- Routing capable through Zookeeper
- Rules are not available
- Stats available through JMX
- Great performance and low resource usage

ZEROMQ \ZERO-EM-QUEUE\:

- Connect your code in any language, on any platform.
- Carries messages across inproc, IPC, TCP, TIPC, multicast.
- Smart patterns like pub-sub, push-pull, and router-dealer.
- High-speed asynchronous I/O engines, in a tiny library.
- Backed by a large and active open source community.
- Supports every modern language and platform.
- Build any architecture: centralized, distributed, small, or large.

APACHE ACTIVEMQ

Apache ActiveMQ ™ is the most popular and powerful open source messaging and Integration Patterns server.

APACHE ACTIVEMQ

Supports a variety of Cross Language Clients and Protocols including:

- OpenWire, Stomp, AMQP, MQTT

Full support for the Enterprise Integration Patterns

- JMS client and the Message Broker

MONITORING, API SECURITY AND INIT CONTAINERS

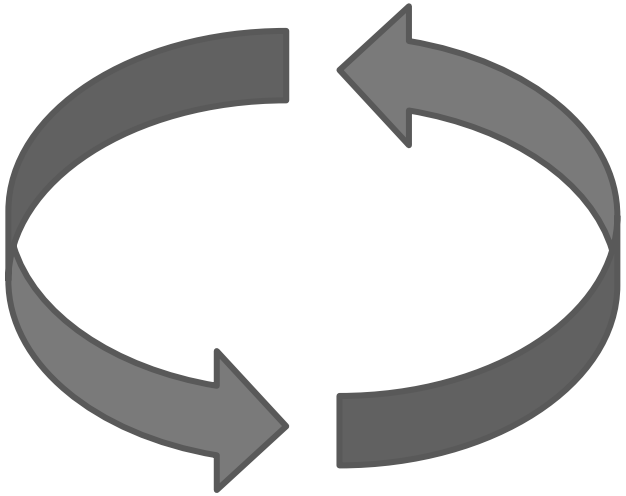
MONITORING

Why monitor?


HYUNDAI FORTUNE
PANAMA
HYUNDAI

MONITORING

Quality Assurance



Continuous Improvement

MONITORING

Not about collecting data

- Why vs How

MONITORING

Why monitor?

Know when things go wrong

Debug and learn from issues

Historical view/Trends

Drive technical/business decisions

Feed into other systems/processes (QA, Security, automation)

MONITORING

Challenges

Monitoring tools are limited

- Technically
- Conceptually

Tools don’t scale well and are difficult to manage

Operational practices don’t align with business needs

MONITORING

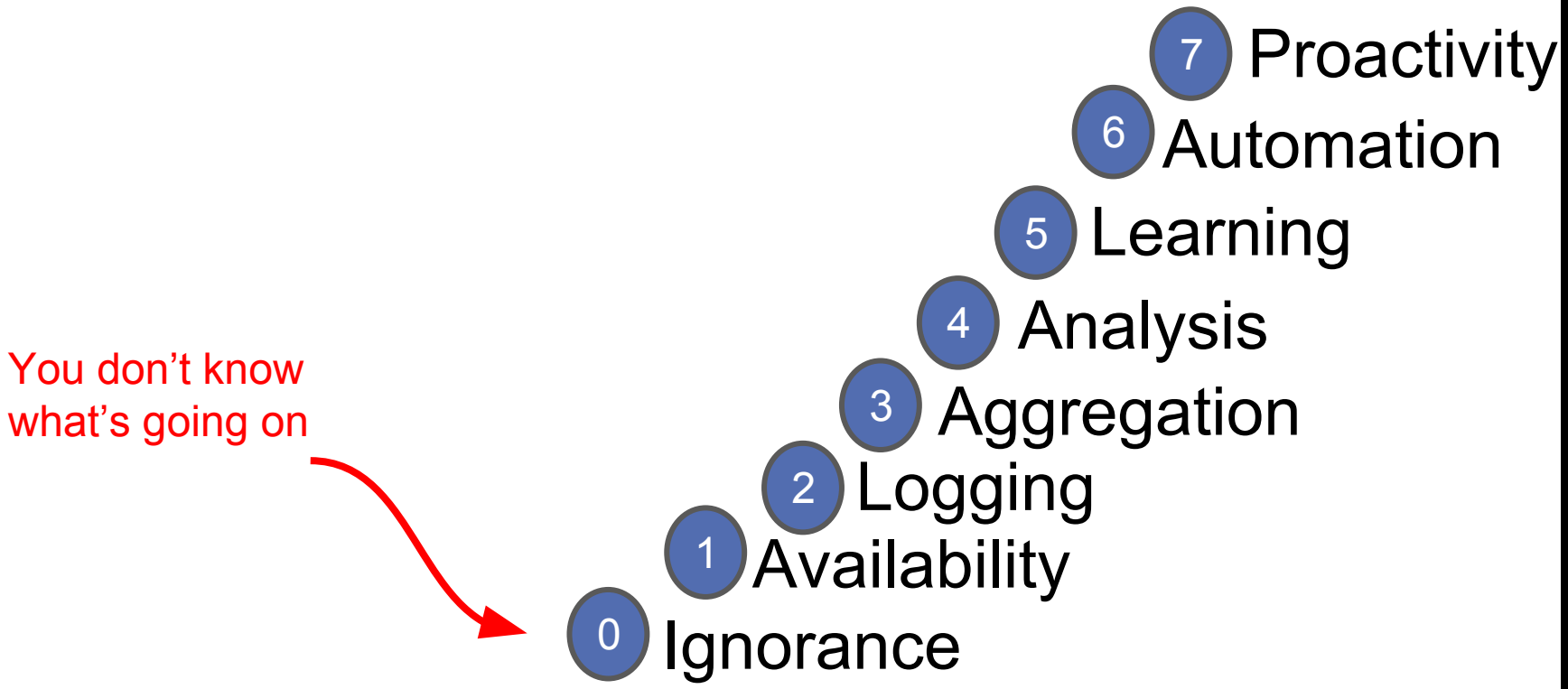
Challenges

Example:

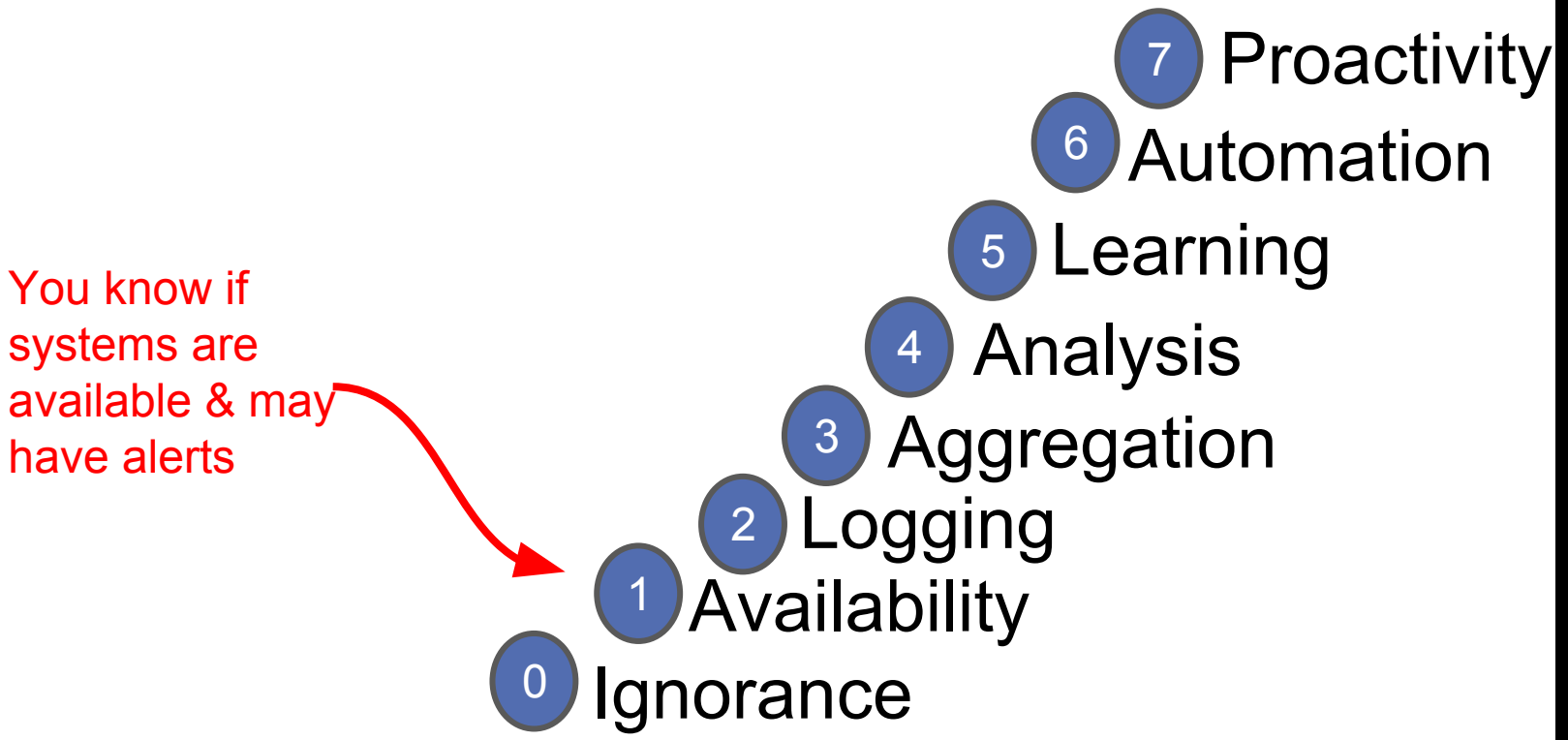
Customers care about increased latency and it's in your SLAs. Monitoring only supports alerting on individual machines performance and not the entire clusters.

Result: Engineers get a bunch of non critical/false-positive alerts and are woken up for non-issues. This leads to fatigue and ignoring alerts.

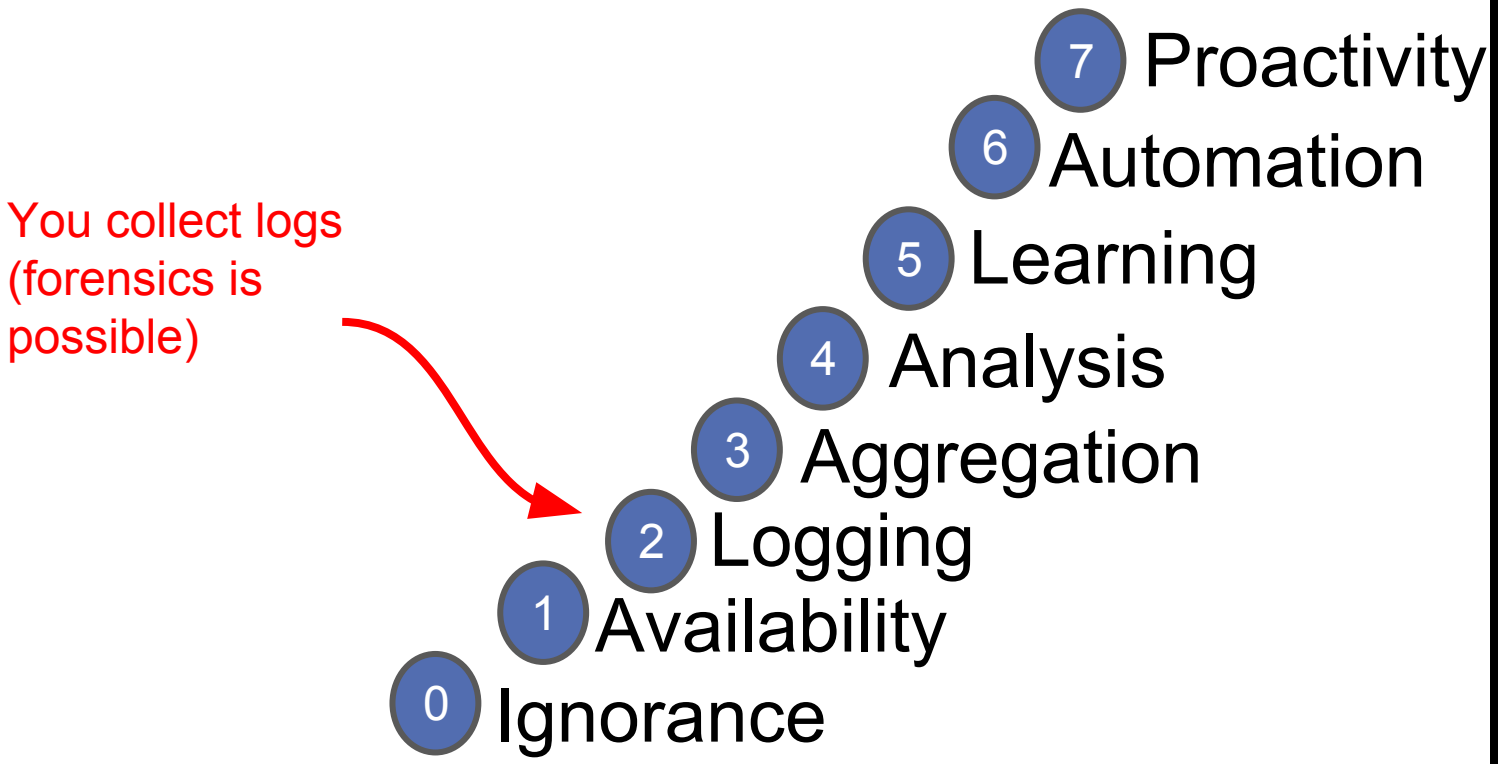
MONITORING


You don’t know
what’s going on
0 Ignorance
1 Availability
2 Logging
3 Aggregation
4 Analysis
5 Learning
6 Automation
7 Proactivity

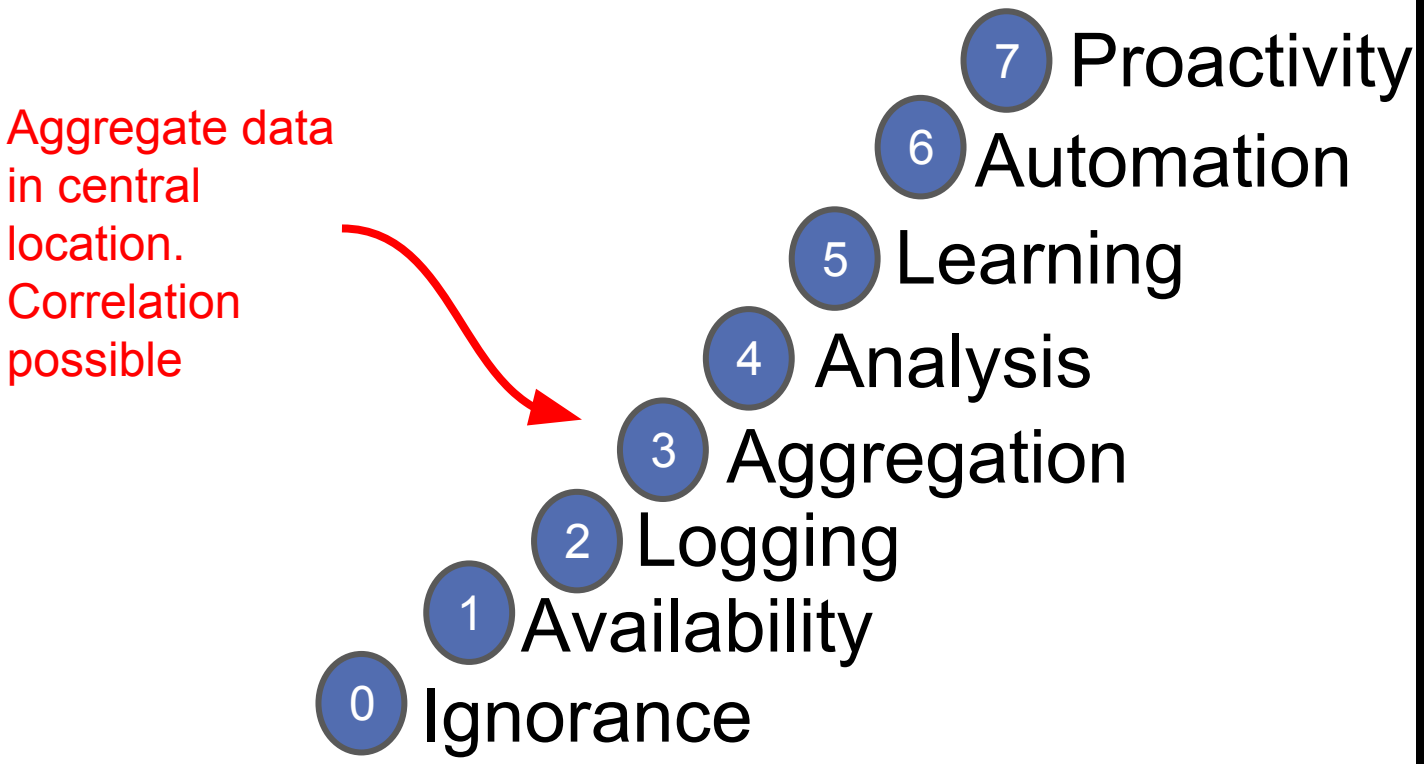
MONITORING


You know if systems are available & may have alerts
0 Ignorance
1 Availability
2 Logging
3 Aggregation
4 Analysis
5 Learning
6 Automation
7 Proactivity

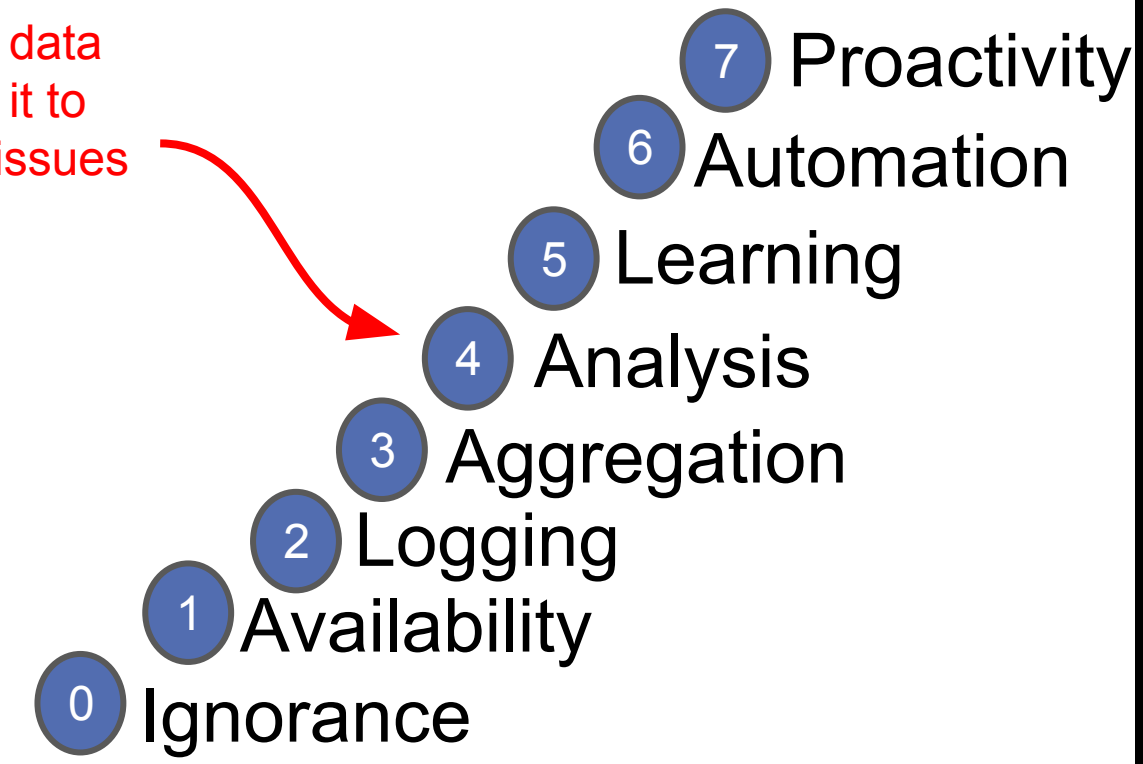
MONITORING


You collect logs (forensics is possible)
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

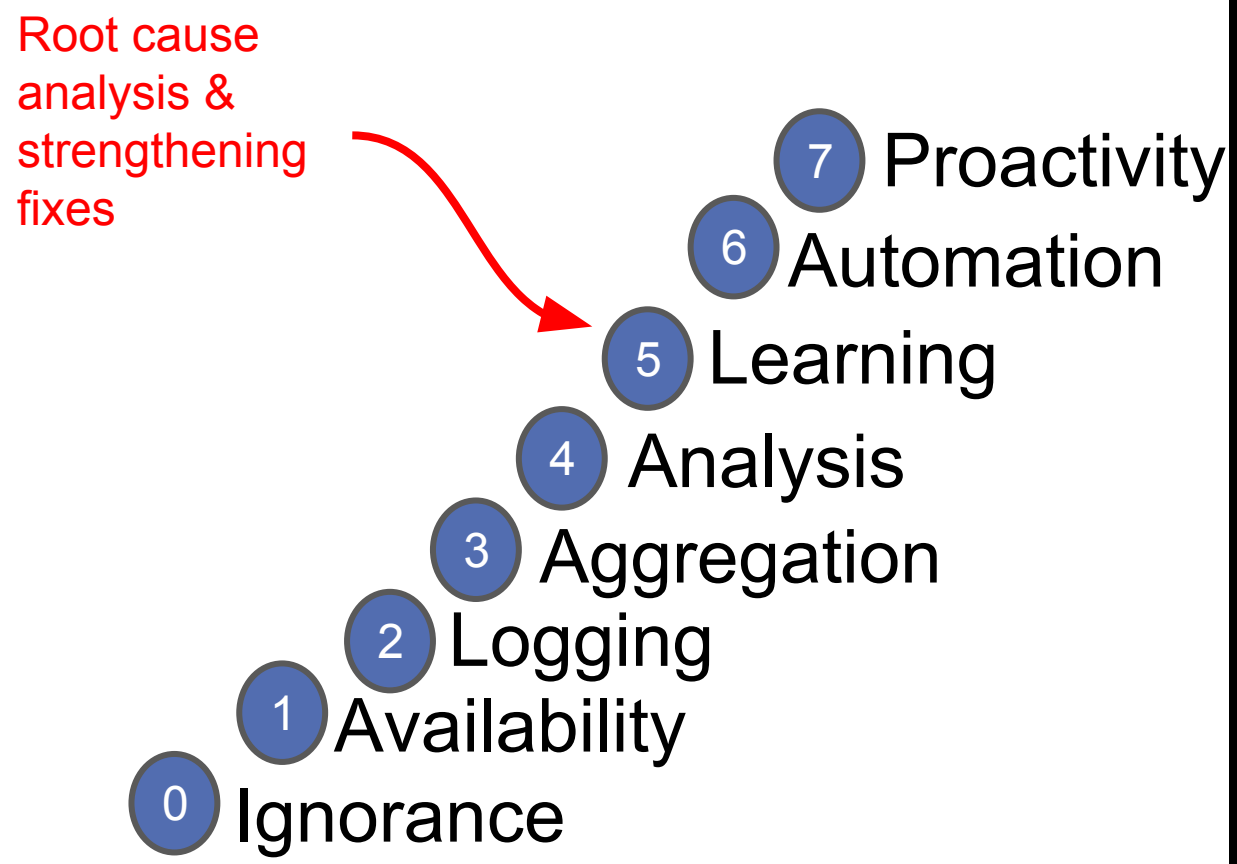
MONITORING


Aggregate data in central location. Correlation possible
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

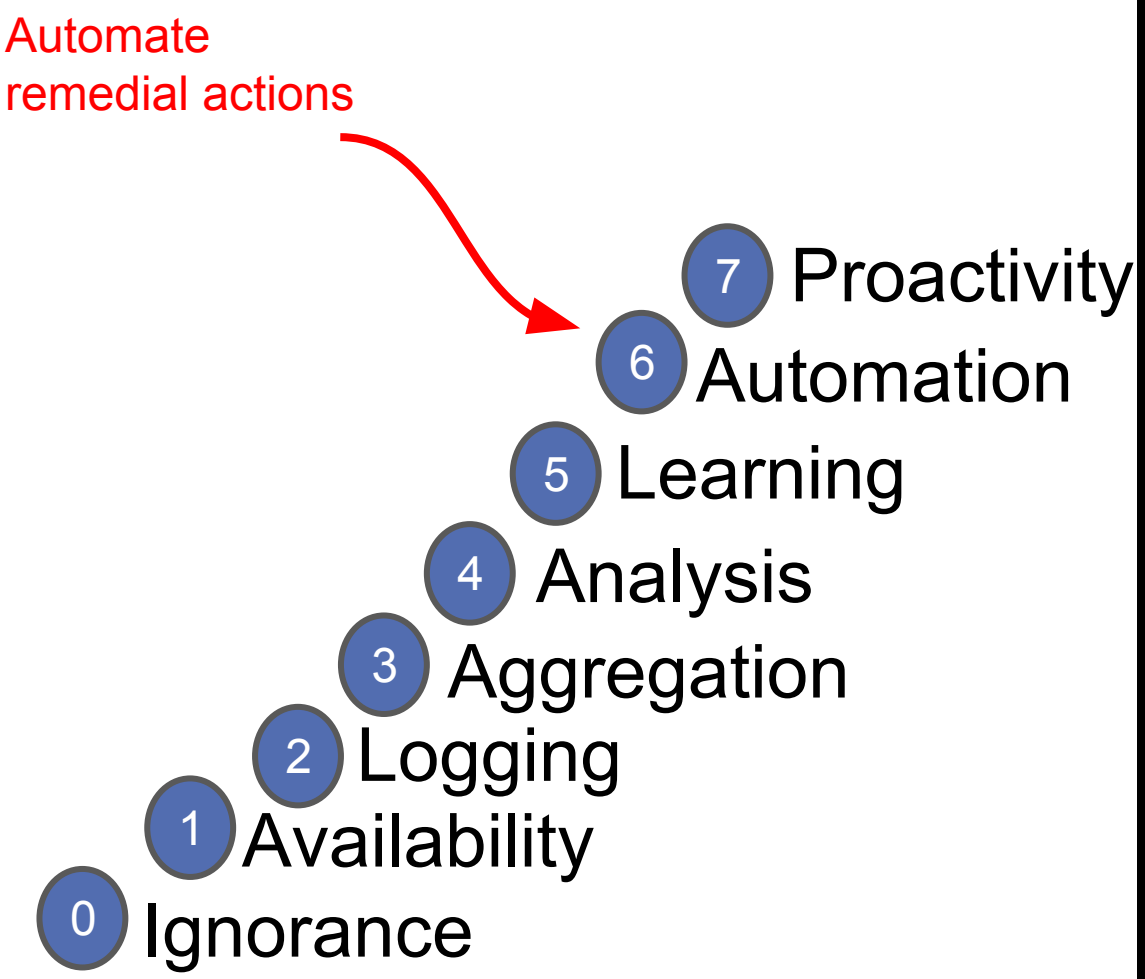
MONITORING


Analyze data
and use it to
resolve issues
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

MONITORING


Root cause analysis & strengthening fixes
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

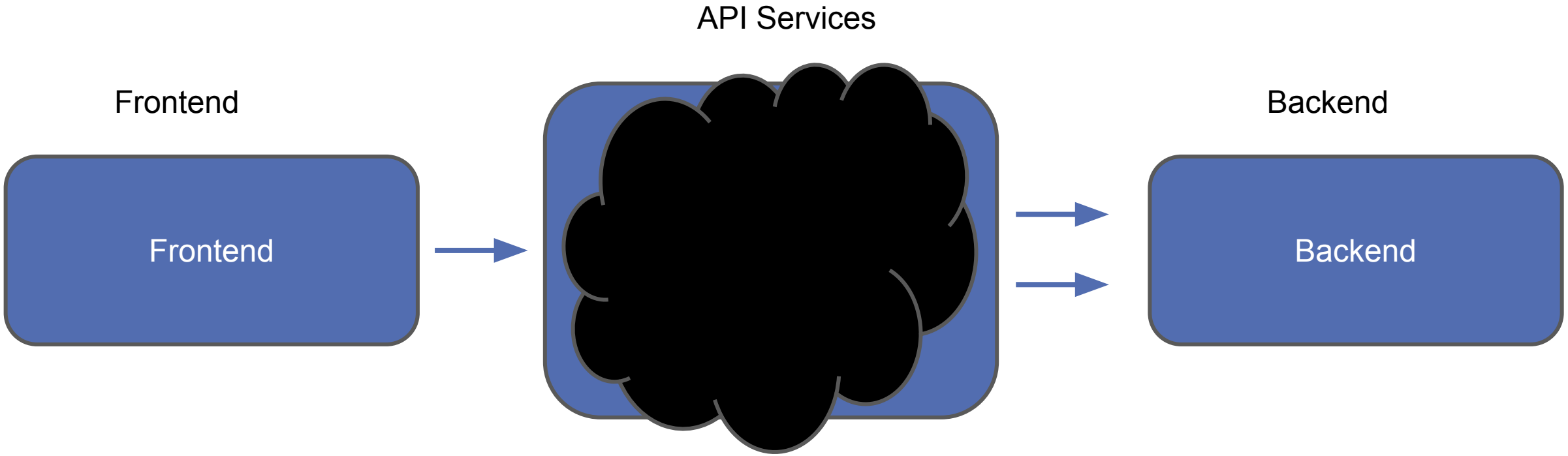
MONITORING


Automate
remedial actions
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

MONITORING

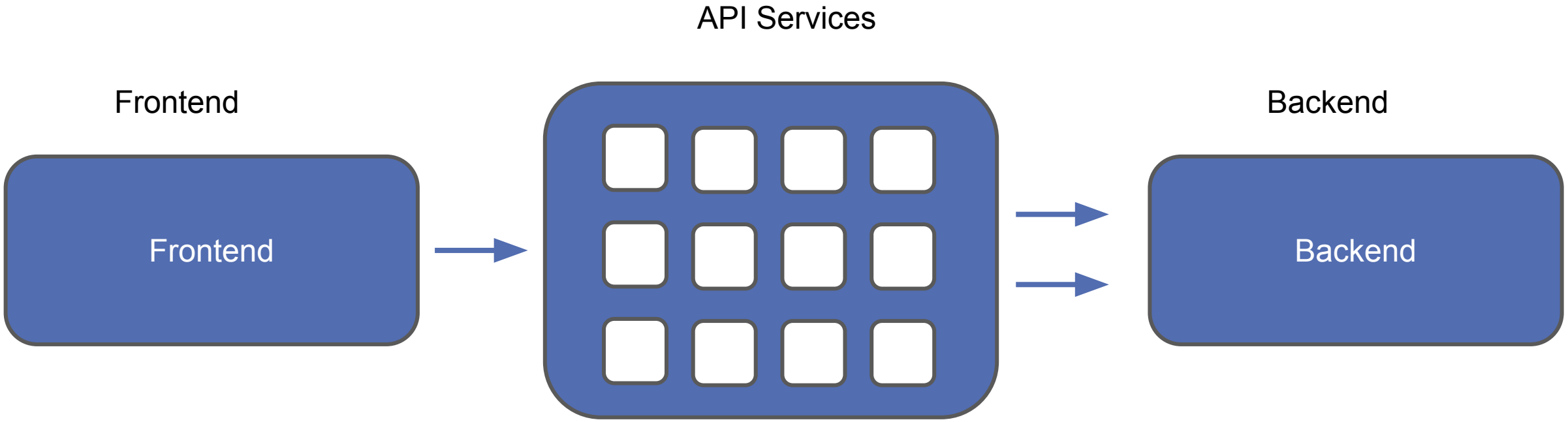

Active strengthening (Attack prevention)
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

MONITORING



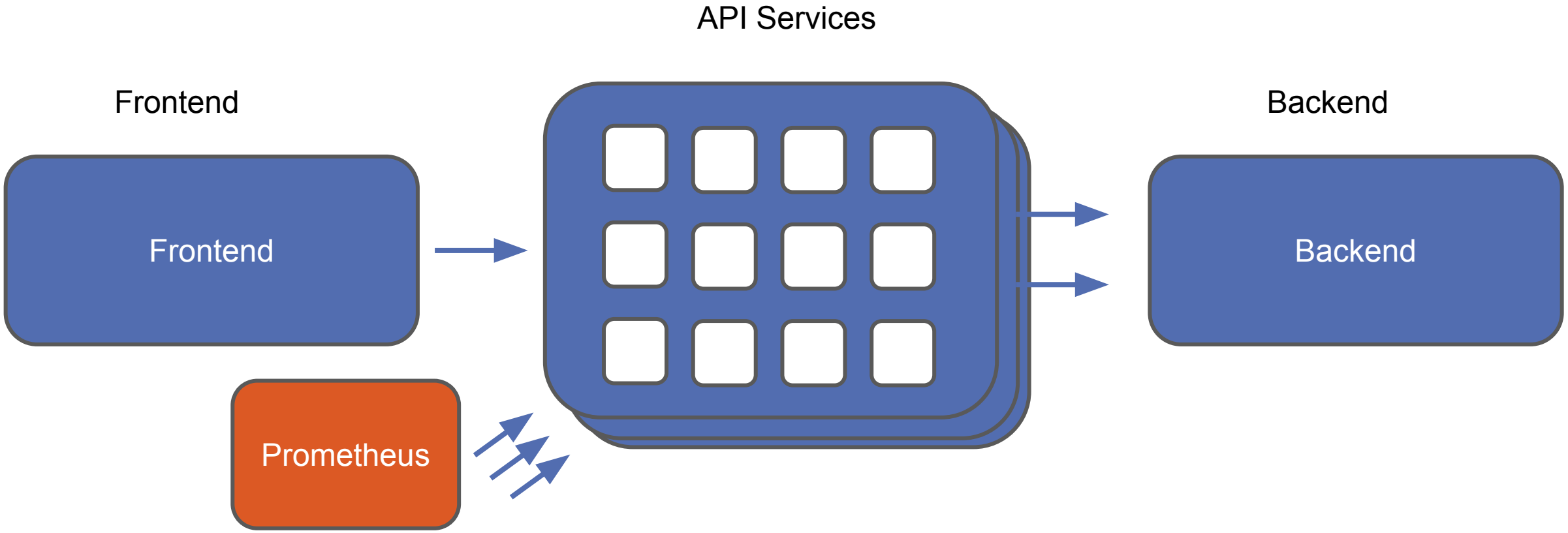
Limited visibility

MONITORING



Monitor Internals

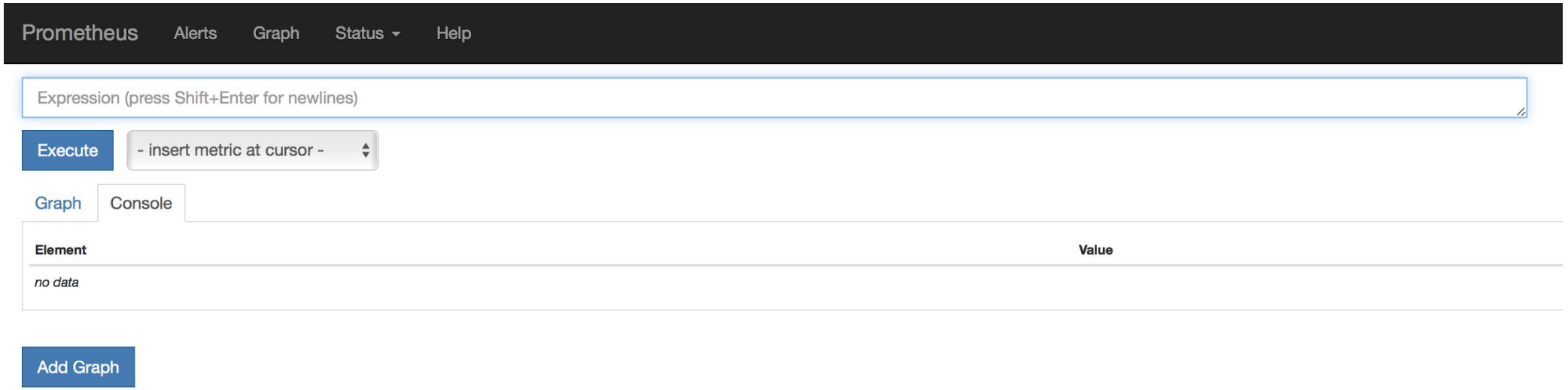
MONITORING



Monitor Services

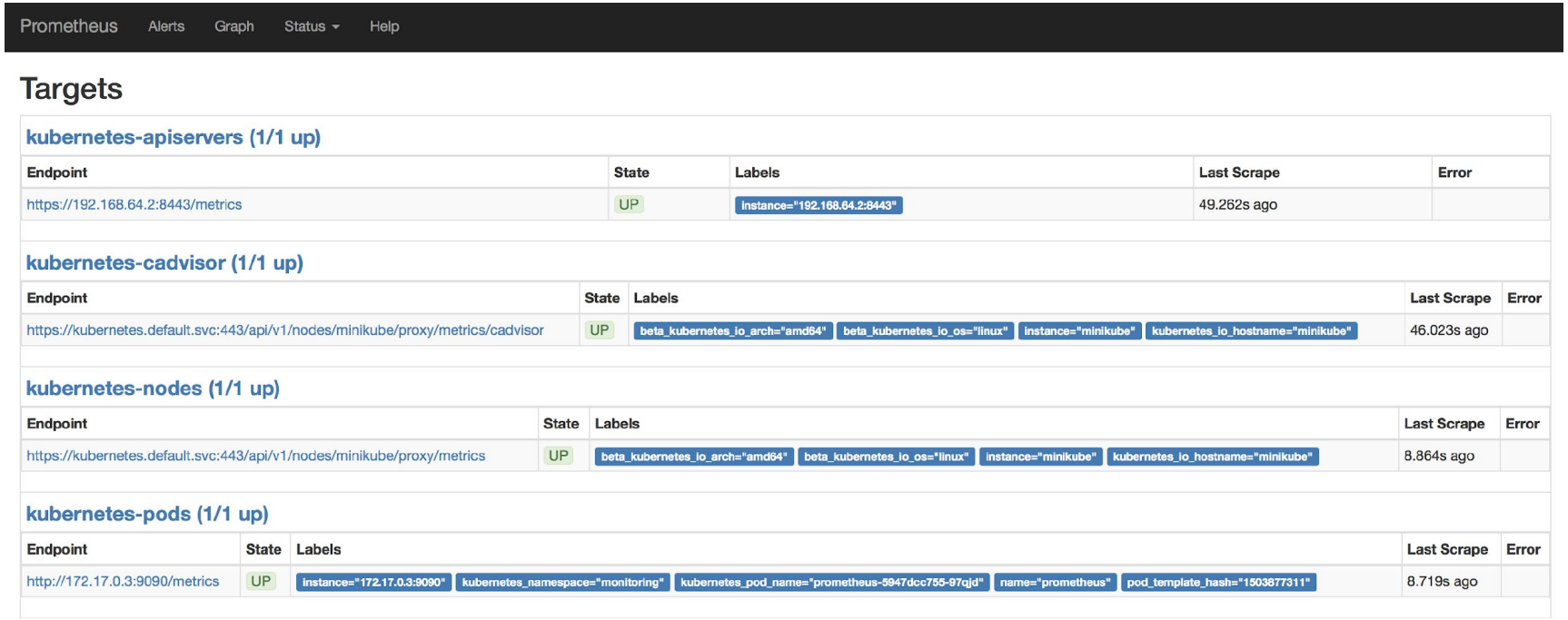
MONITORING

Prometheus



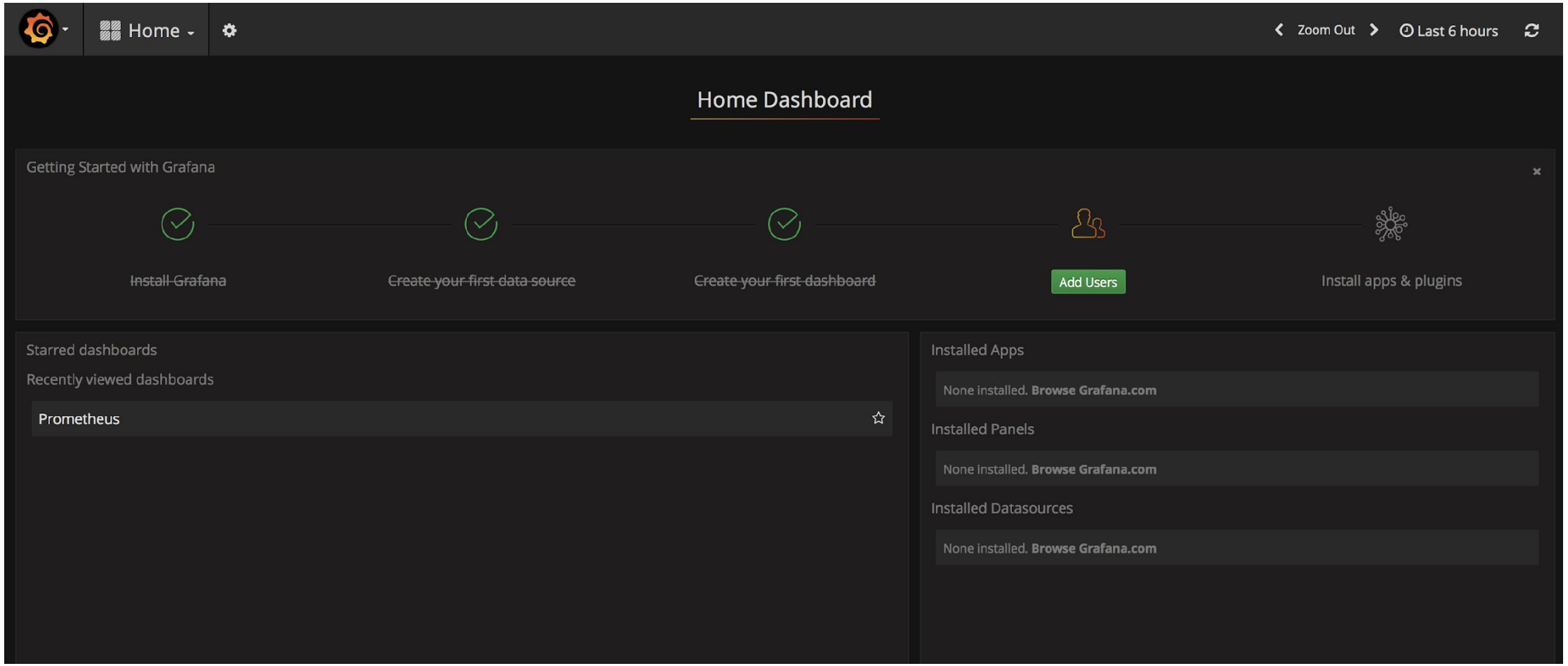
MONITORING

Prometheus



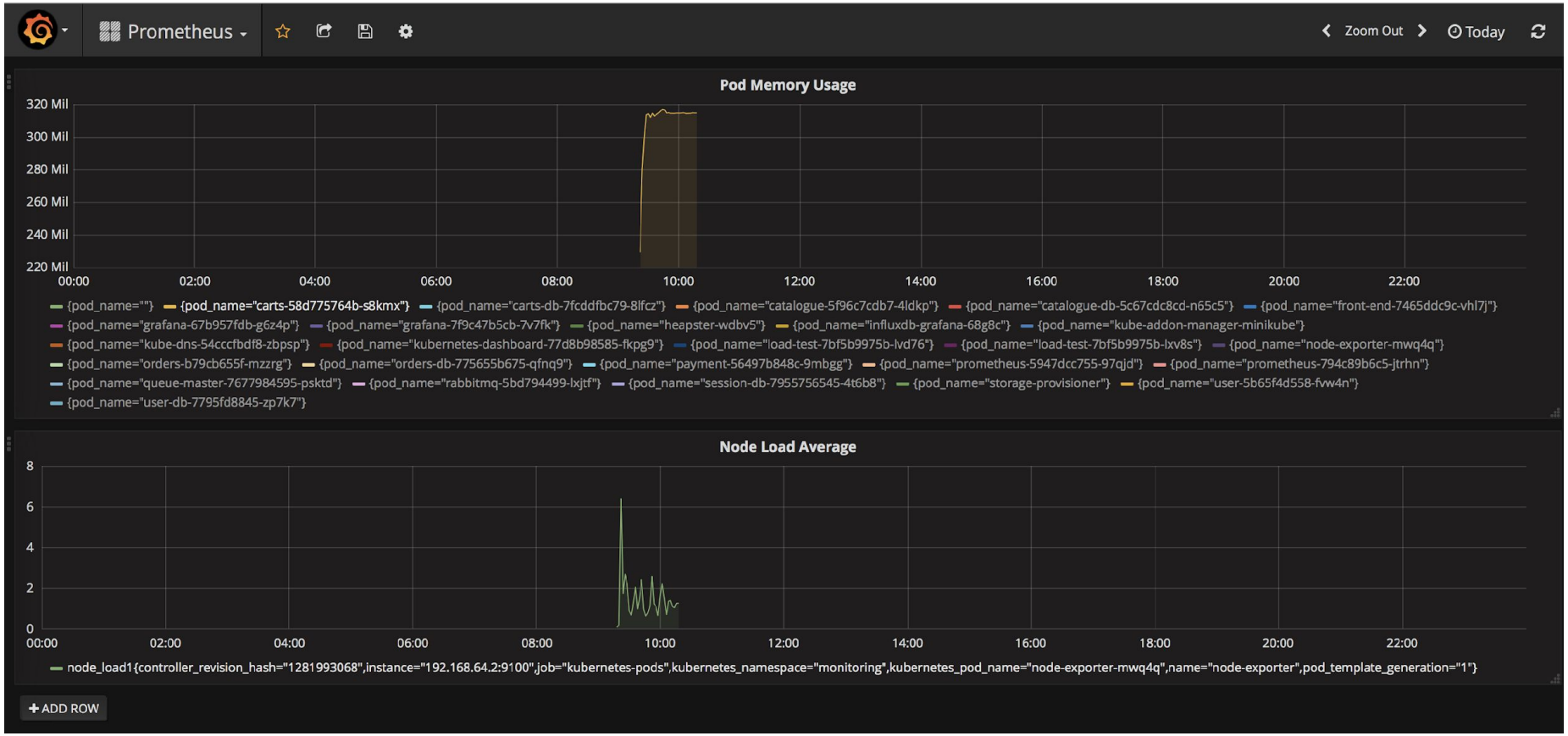
MONITORING

Grafana



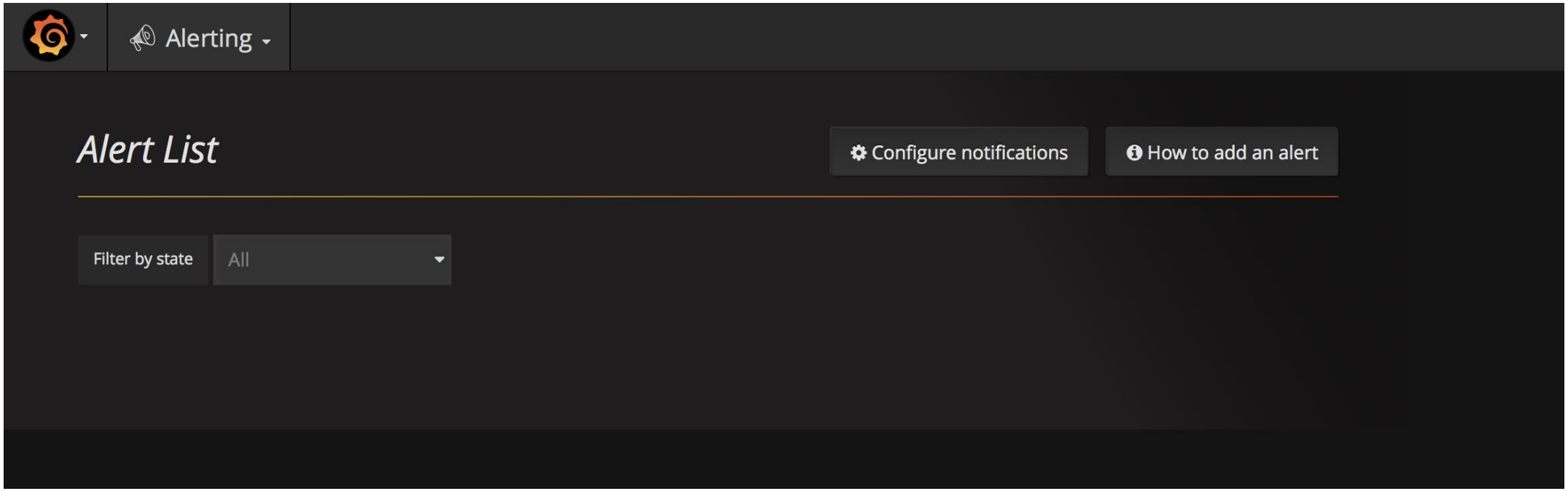
MONITORING

Grafana



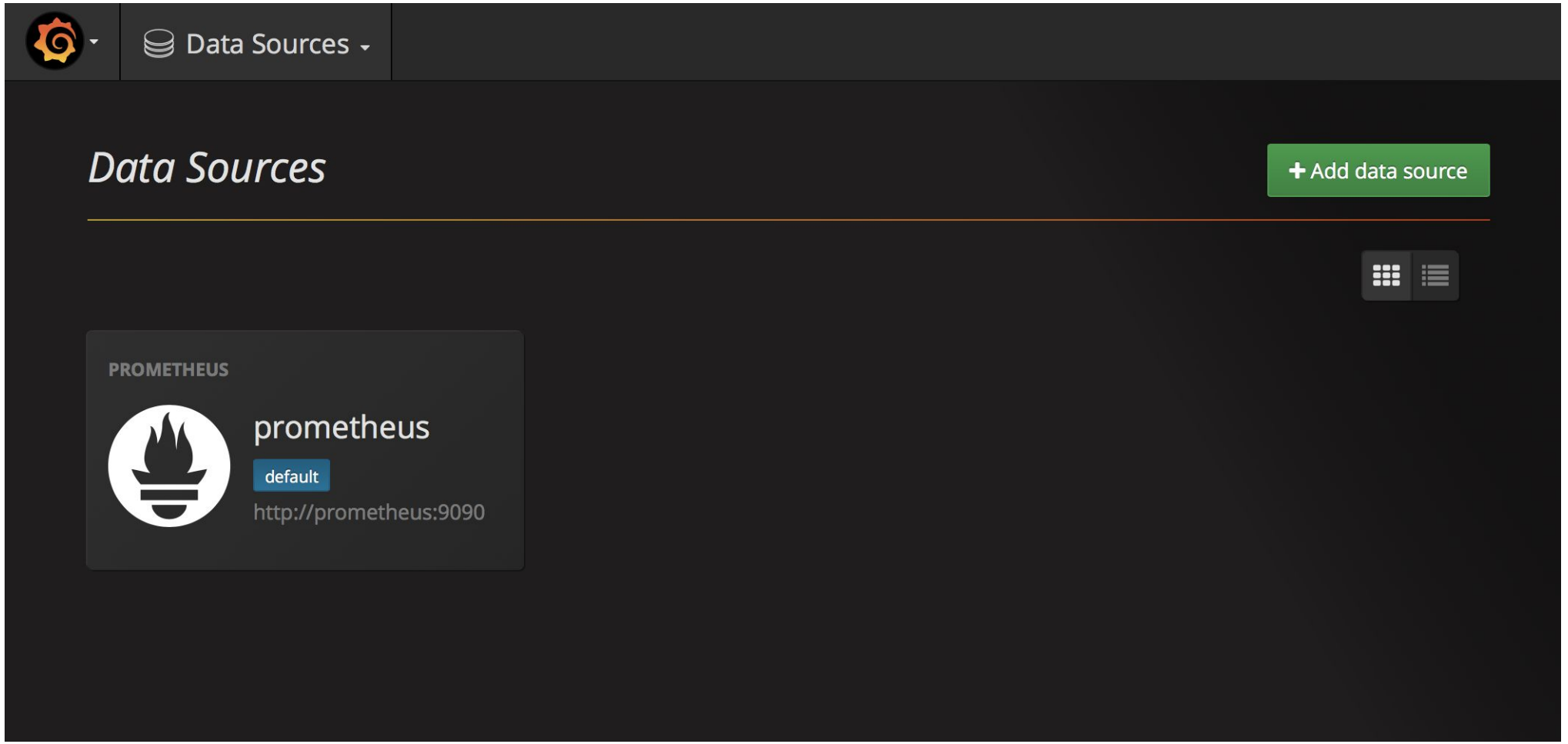
MONITORING

Grafana



MONITORING

Grafana

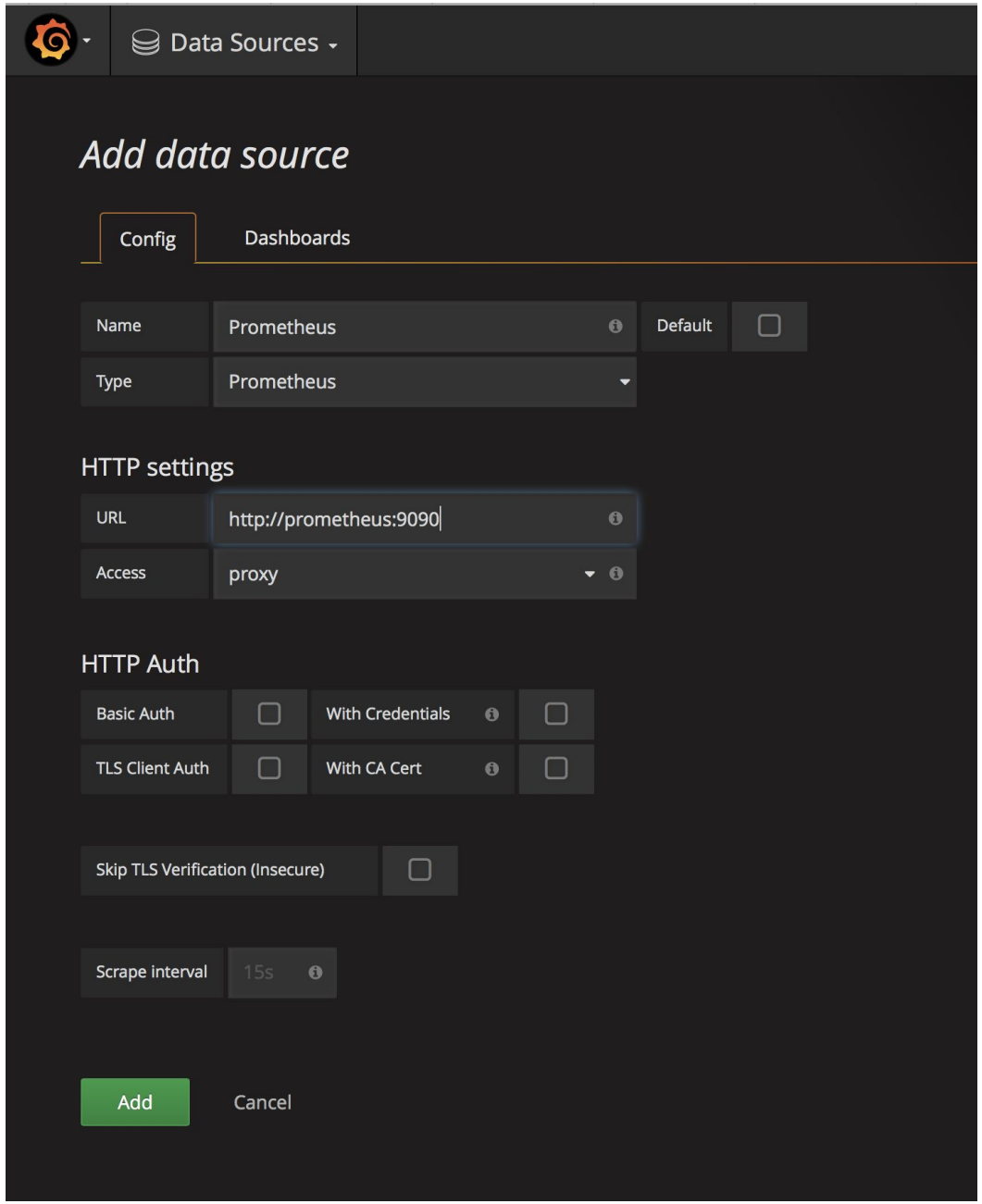


MONITORING

Grafana

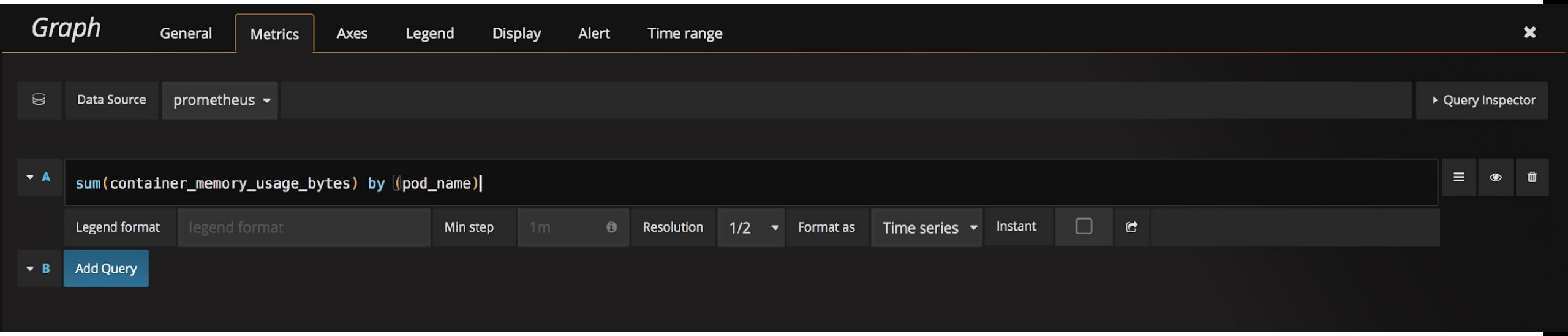
&

Prometheus



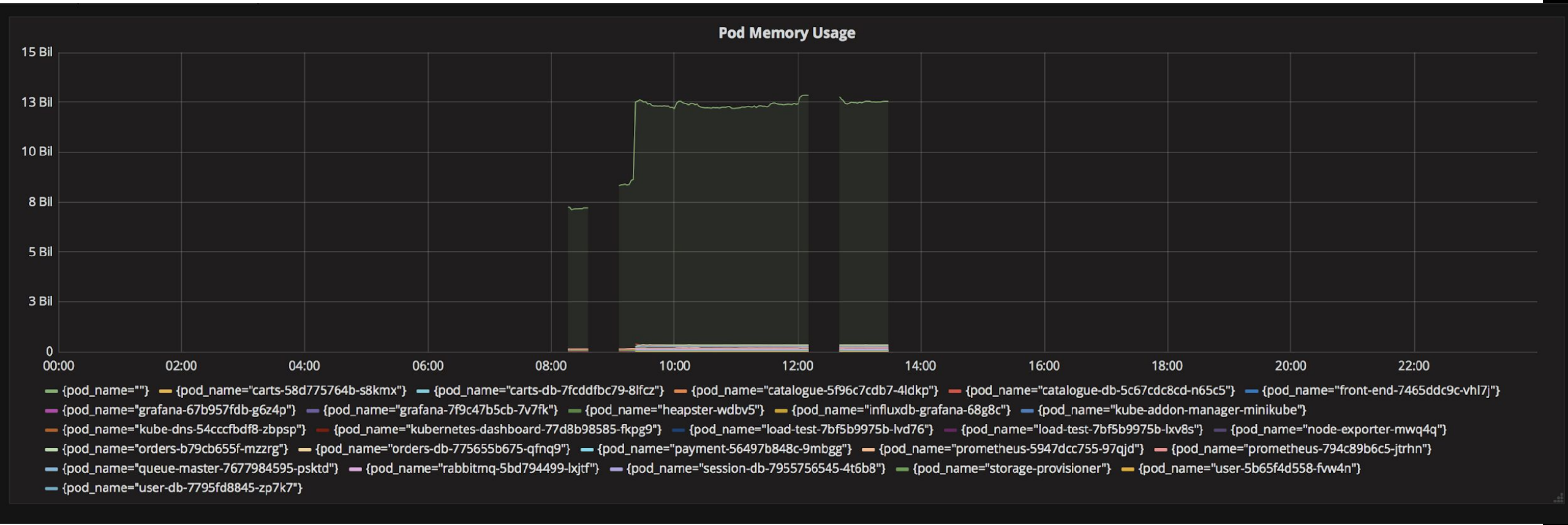
MONITORING

Add graph



MONITORING

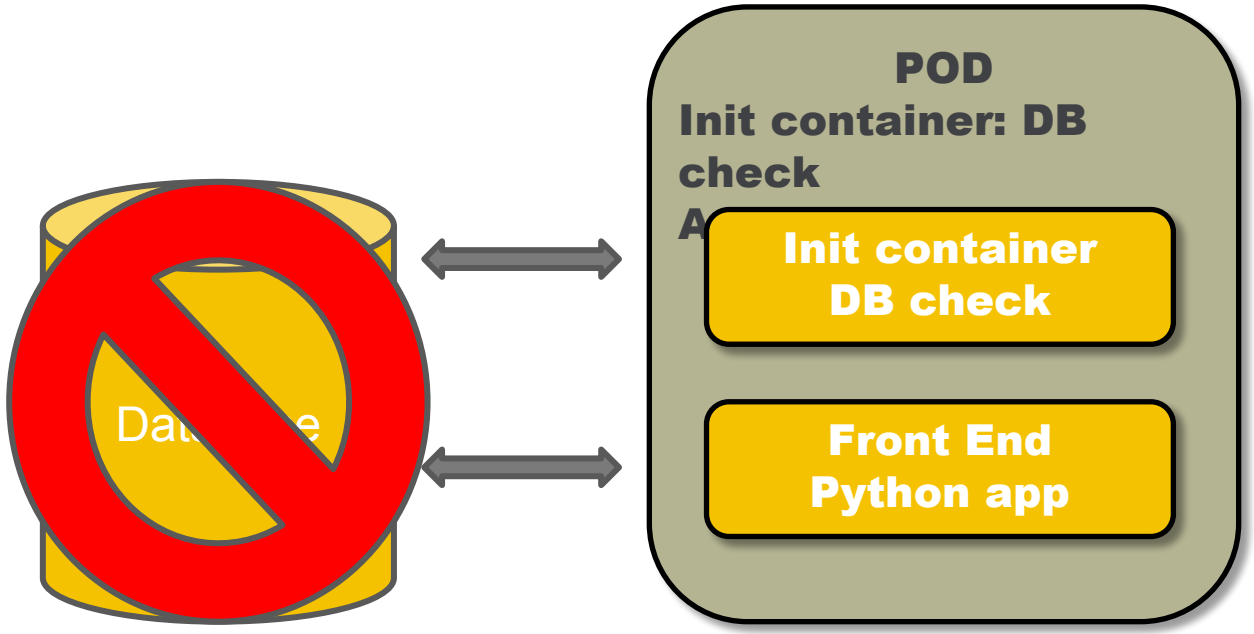
Grafana graph



INIT CONTAINERS

Application orchestration

- **Init container runs first to completion**
  - If failure, POD is restarted until successful.



INIT CONTAINERS

Support same features as application containers.

- Resource limits
- Volumes
- Security settings
- Etc.

Do not support readiness probes since they must run to completion before POD can be “ready”

KUBERNETES 1.5 SYNTAX

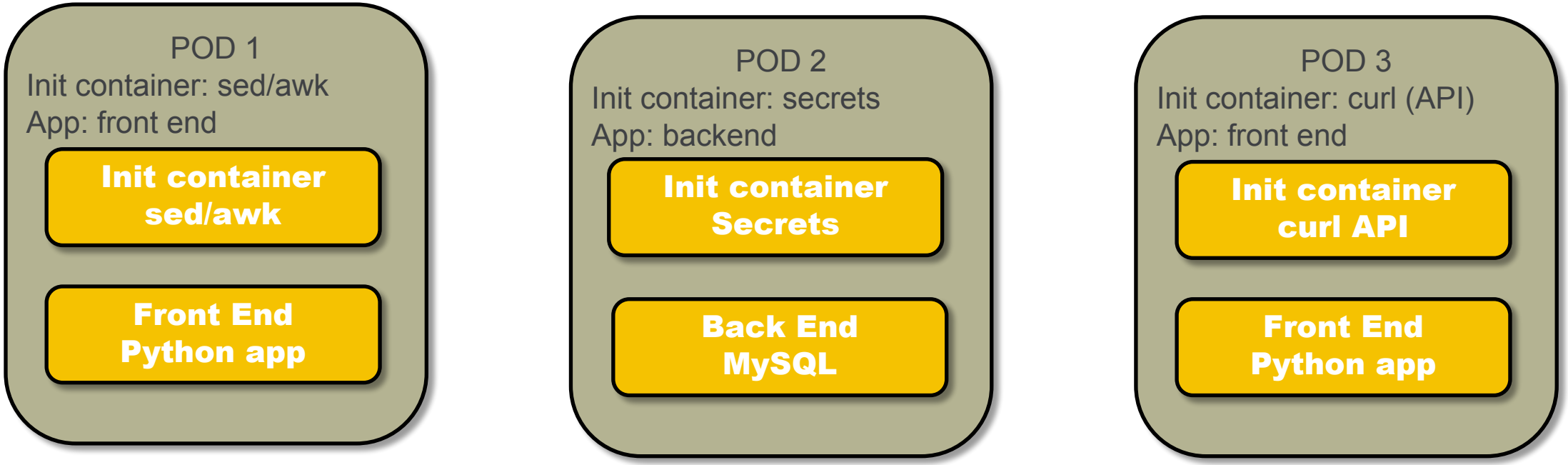
```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; do echo waiting for myservice; sleep 2; done;"]
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb; sleep 2; done;"]
        }
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

KUBERNETES 1.6+ SYNTAX

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

INIT CONTAINER USE-CASES

Lots of reasons



STORING SECRETS INSIDE KUBERNETES

What secrets do applications have?

- Database credentials
- API credentials & endpoints (Twitter, Facebook etc.)
- Infrastructure API credentials (Google, Azure, AWS)
- Private keys (TLS, SSH)
- Many more!

STORING SECRETS INSIDE KUBERNETES

It is a bad idea to include these secrets in your code.

- Accidentally push up to GitHub with your code
- Push into your file storage and forget about
- Etc.

STORING SECRETS INSIDE KUBERNETES

There are bots crawling GitHub searching for secrets

Real life example:

- Dev put keys out on GitHub, woke up next morning with a ton of emails and missed calls from Amazon
- 140 instances running under Dev's account.
- $2,375 worth of Bitcoin mining.

CREATE SECRET

- **Designed to hold all kinds of sensitive information**
- **Can be used by Pods (filesystem & environment variables) and the underlying kubelet when pulling images**

```
apiVersion: v1
kind: Secret
metadata:
   name: mysecret
Type: Opaque
data:
    password: mmyWfoidfIuL==
    username: NyhdOKwB
```

POD SECRET

```
kind: Pod
metadata:
   name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
    - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
```

VOLUME SECRET

```
spec:
  containers
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: “secrets”
      mountPath: “/etc/my-secrets”
      readOnly: true
  volumes:
   - name: “secrets”
      secret:
        secretName: “mysecret”
```

MICROSERVICES IN PRACTICE

MICROSERVICES IN PRACTICE

Implementing a microservice architecture poses specific design, development and operational challenges.

Specifically, we want to address the following challenges:

- Microservice Patterns
- Microservice Implementation Solutions

MICROSERVICE PATTERNS

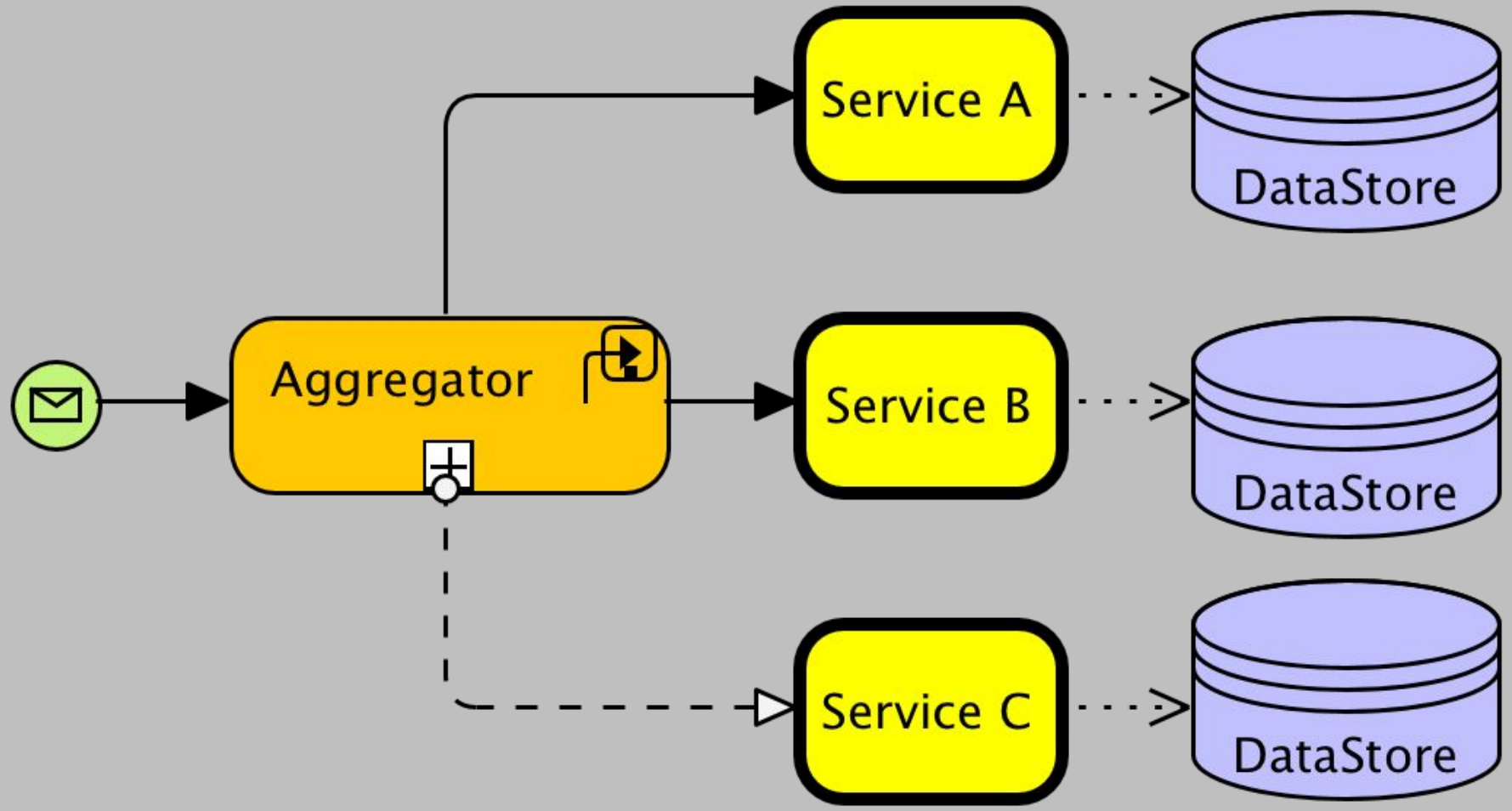
MICROSERVICES PATTERNS

- Aggregator
- Branch
- Proxy
- Chained
- Shared Resources (Data)
- Asynchronous Messaging
- Bulkhead Isolation

PATTERN: AGGREGATOR

Description:

- An Aggregator is a web page that invokes multiple services to achieve the required functionality
- An Aggregator could also be a composite service
- This design pattern follows the DRY principle
- An Aggregator is similar to the Façade Pattern

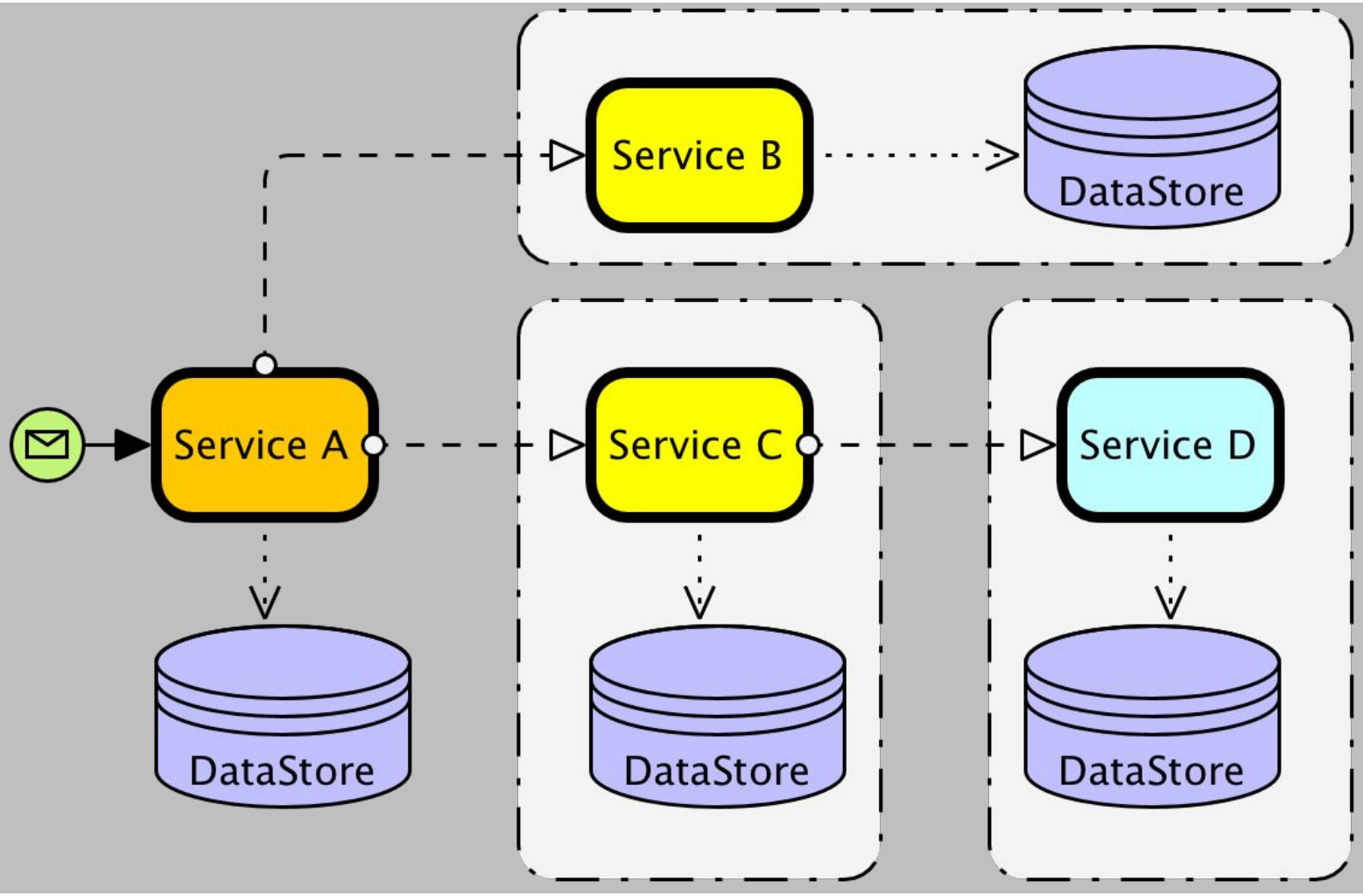


PATTERN: AGGREGATOR

PATTERN: BRANCH

Description:

- The Branch design pattern extends the Aggregator design pattern
- Allows simultaneous response processing from two, likely mutually exclusive, chains of microservices
- This pattern can also be used to call different chains, or a single chain, based upon the business needs
- Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern
- This may be configured using routing of EIP endpoints, and would need to be dynamically configurable

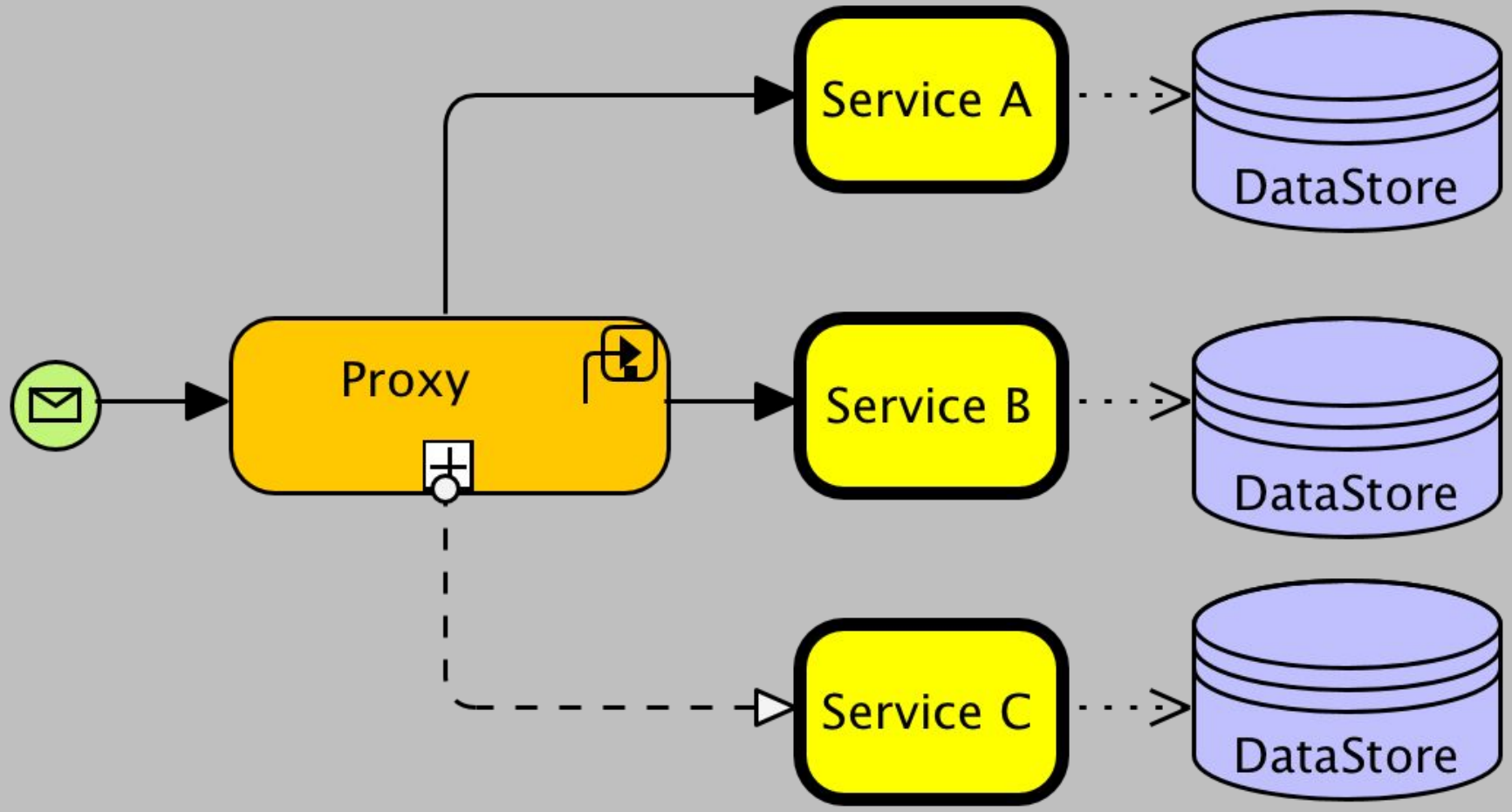


PATTERN: BRANCHING

PATTERN: PROXY

Description:

- Proxy microservice design pattern is a variation of the Aggregator Pattern
- Different microservice may be invoked based upon the business needs
- Each Proxy can scale independently
- A proxy is a consumer facing interface in leu of exposing individual services
- May be a dumb proxy that delegates requests to one of the services
- May be a Smart Proxy, where data is transformed before response returned to consumer

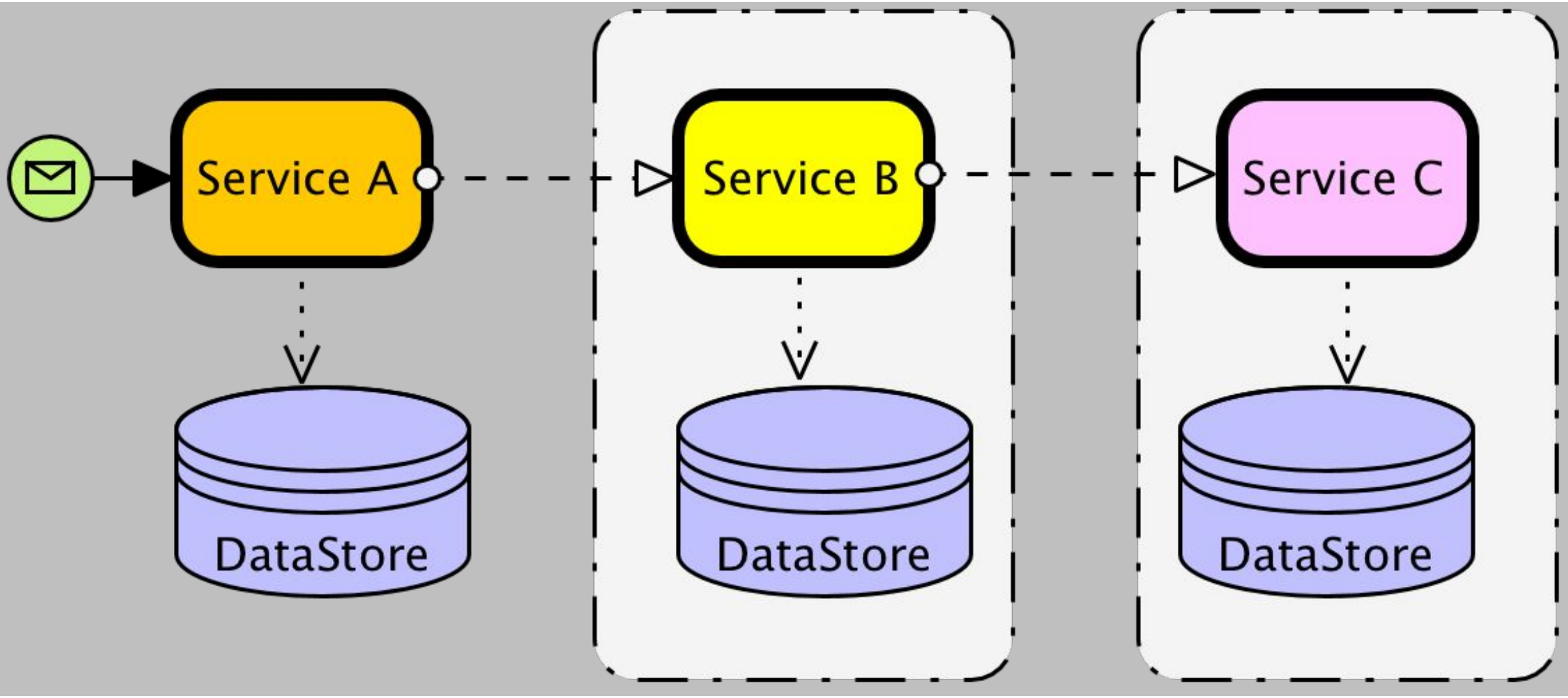


PATTERN: PROXY

PATTERN: CHAIN

Description:

- Produces single consolidated response to request
- Request is received by Service A, which is chained to Service B, which in turn which is changed to Service C
- Services are likely using a synchronous HTTP request/response messaging
- Client is blocked until request/response chain is completed
- Important not to make chains too long
- A chain with a single microservice is called singleton chain

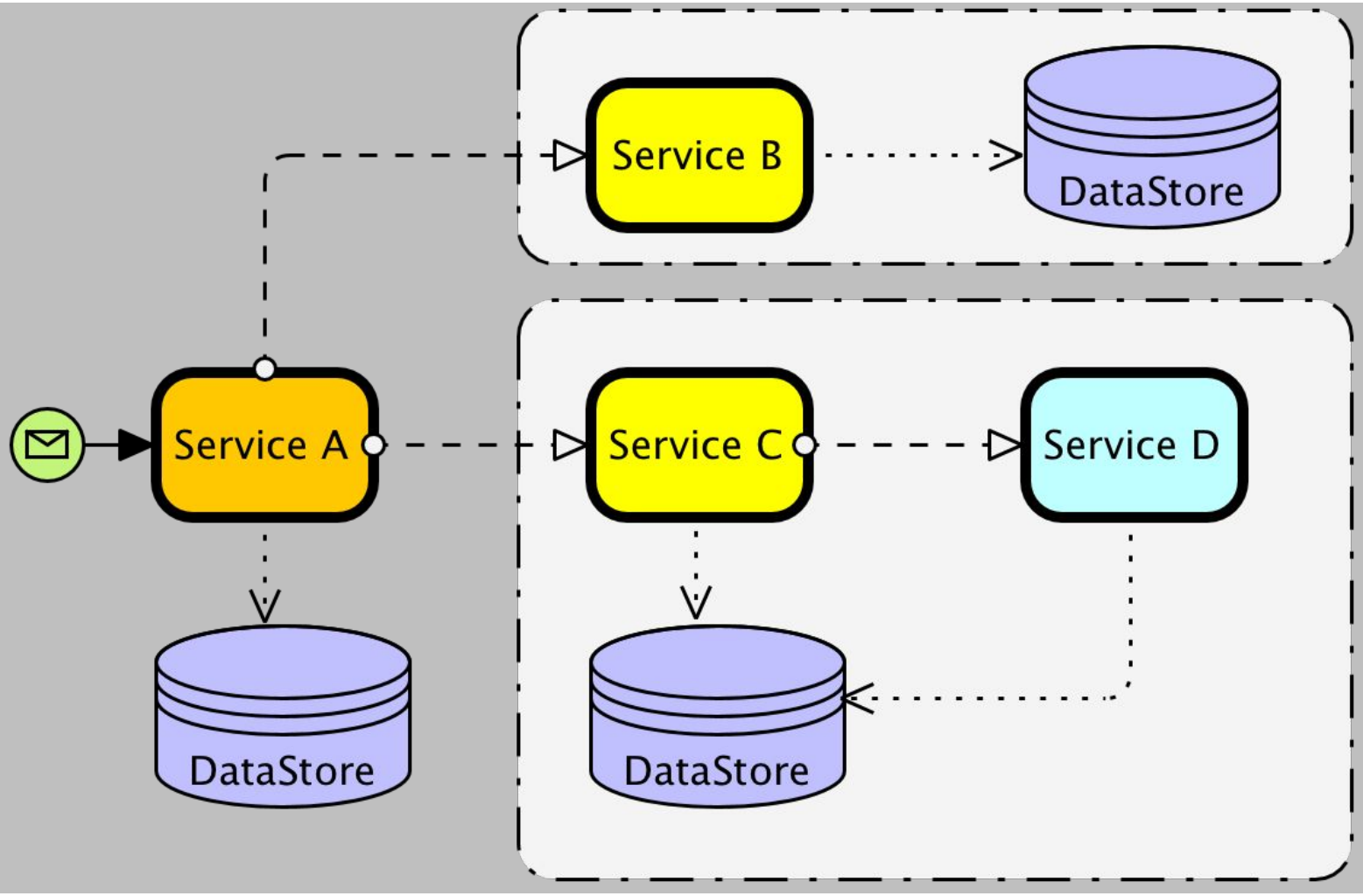


PATTERN: CHAIN

PATTERN: SHARED RESOURCES (DATA)

Description:

- A fundamental design principle of microservice is autonomy
- A service should be full-stack and have control of all the components
    - UI, middleware, persistence, transaction, etc.
- Service should be polyglot
    - Use the right tool for the right job
- Data normalization is typical issue when refactoring existing monolithic applications
- This pattern may benefit from shared caching and database stores
- This is a transitional pattern until service can be fully autonomous

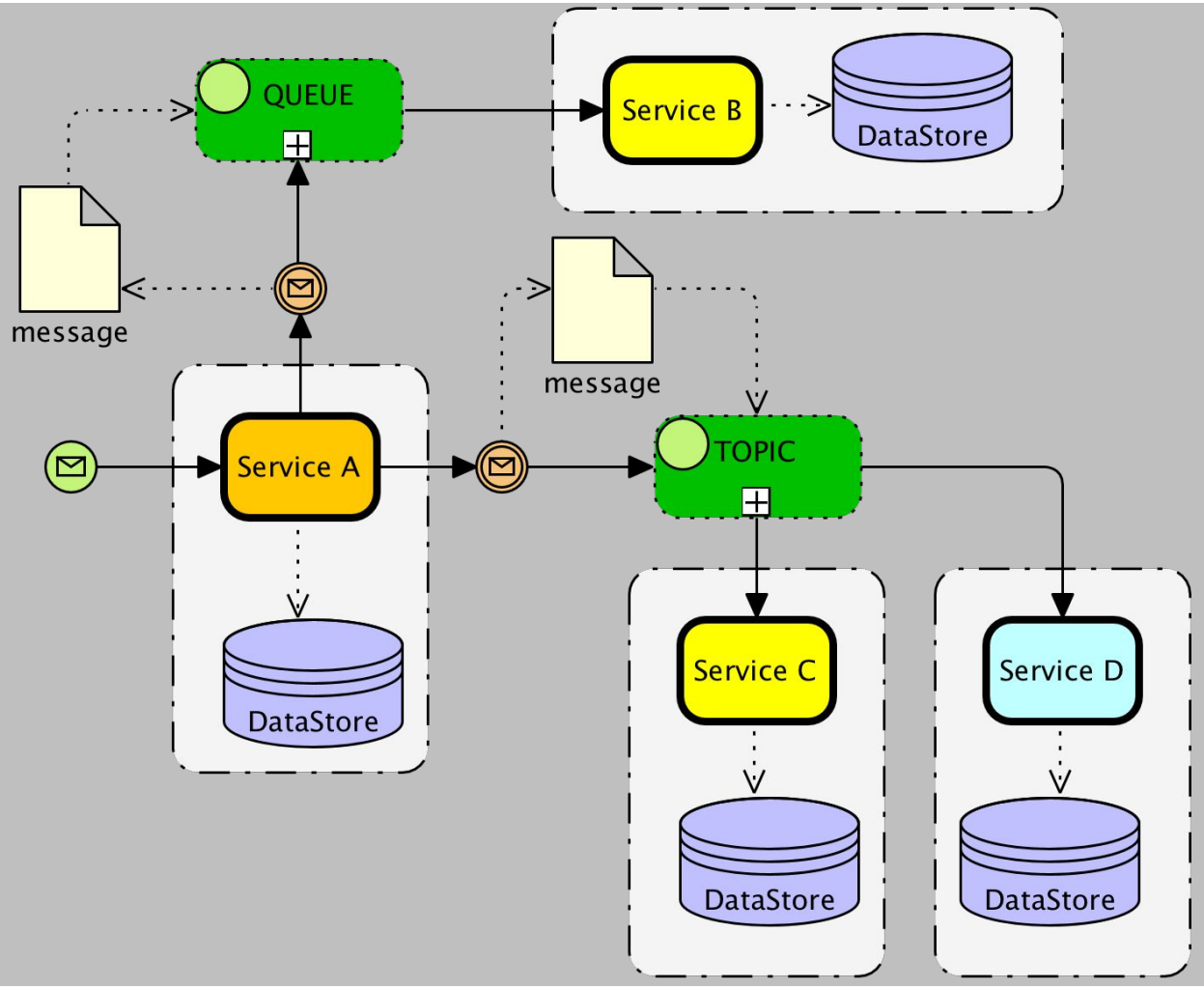


PATTERN: SHARED RESOURCE (DATA)

PATTERN: ASYNCHRONOUS MESSAGING

Description:

- RESTful services are inherently synchronous, and thus blocking
- Microservice architectures may use asynchronous messaging instead of RESTful request/response
- A service may call another service synchronously, which then communicates with other services asynchronously
- Could communicate asynchronously in many different forms:
    - WebSockets, Message Queues, Reactive Streams
- Can combine RESTful service and publish/subscribe messaging to accomplish the business needs

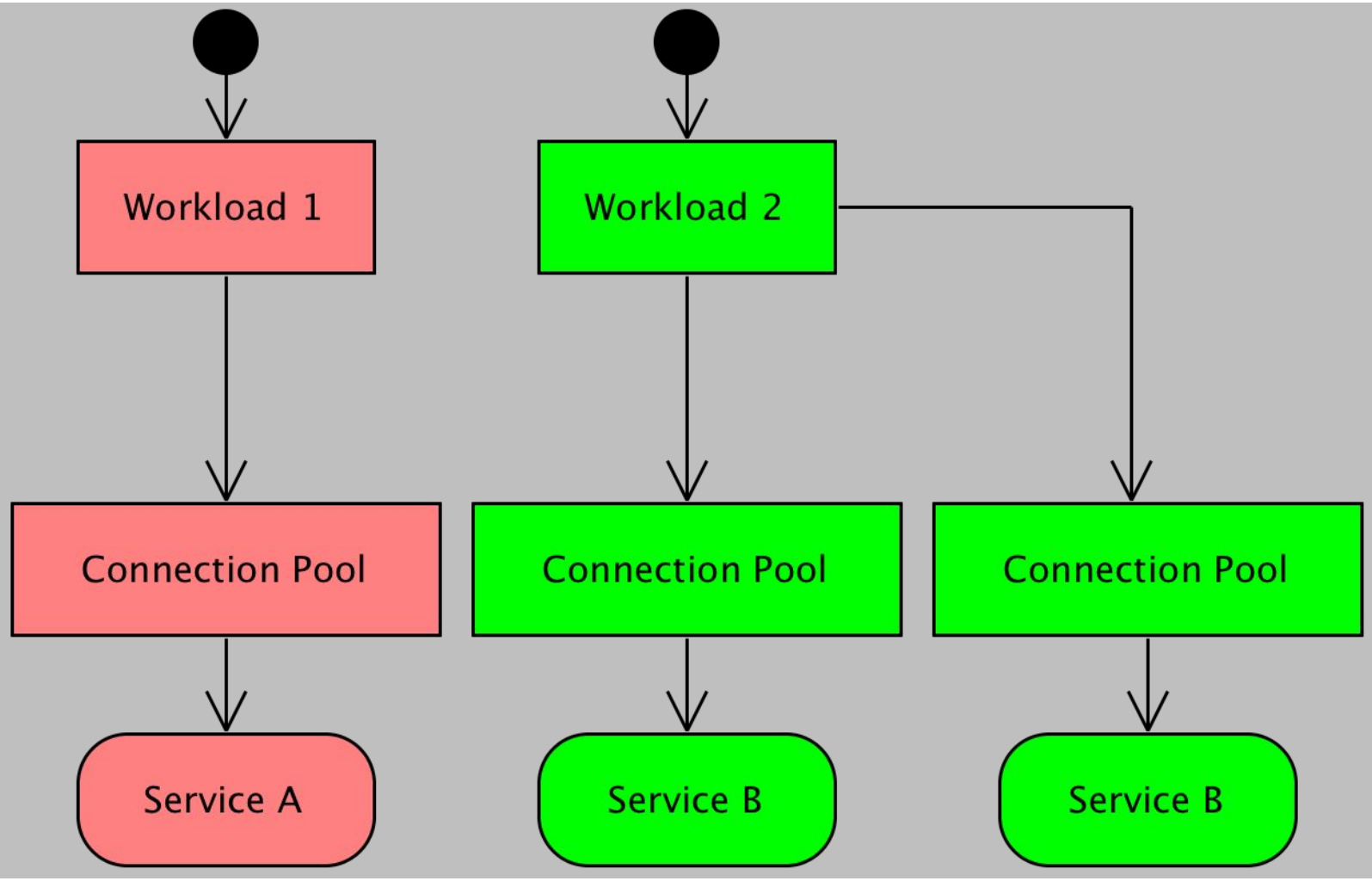


PATTERN: ASYNCHRONOUS MESSAGING

PATTERN: BULKHEAD ISOLATION

Description:

- Isolate elements of an application into pools so that if one fails, the others will continue to function
- This pattern resembles the sectioned partitions of a ship's hull
  - If ship hull is compromised, only the damaged section fills with water, which prevents the ship from sinking
- Partition service instances into different groups, based on consumer load and availability requirements
- Isolating failures, and allowing application to sustain service functionality for some consumers, even during a failure



PATTERN: BULKHEAD

PATTERN: BULKHEAD ISOLATION

Issues and considerations

- Define partitions around the business and technical requirements of the application
- Consider level of isolation offered by the technology as well as the overhead in terms of cost, performance and manageability
- Consider combining bulkheads with retry, circuit breaker, and throttling patterns to provide more sophisticated fault handling
- Use processes, thread pools, and semaphores
- Use frameworks for creating consumer bulkheads
- Deploy into separate VM's, containers, or processes
- Isolate asynchronous service into different queues
- Monitor each partition's performance and SLA

DATA ISLAND'S

DATA ISLAND'S

- Microservices are not isolated islands, they come in systems
- Data systems are about exposing data. Services are about hiding data
- The more shared data hidden inside a service boundary, the more complex the interface
- The more complex the interface the harder it will be to join datasets across different services
- Extracting whole dataset into mutable islands, the more they will diverge over time
- Better to take a decentralized approach to data and services with data-streams such as Kafka

IMMUTABILITY

Definition

Unchanging over time or unable to be changed, fixed, permanent, carved in stone

IMMUTABILITY

Once you created an artifact, be that a container image, or a VM image, or maybe a package from compiled code - you declare that you will never ever change it.

Often if any changes are required, you declare that a new version of "thing" will be created instead.

IDEMPOTENCE

Definition

The result of a successful performed request is independent of the number of times it is executed.

IDEMPOTENCE

NATURAL IDEMPOTENCY

Some services can be executed as often as you want because they just alter state.

Example:

confirmCustomer()

BUSINESS IDEMPOTENCY

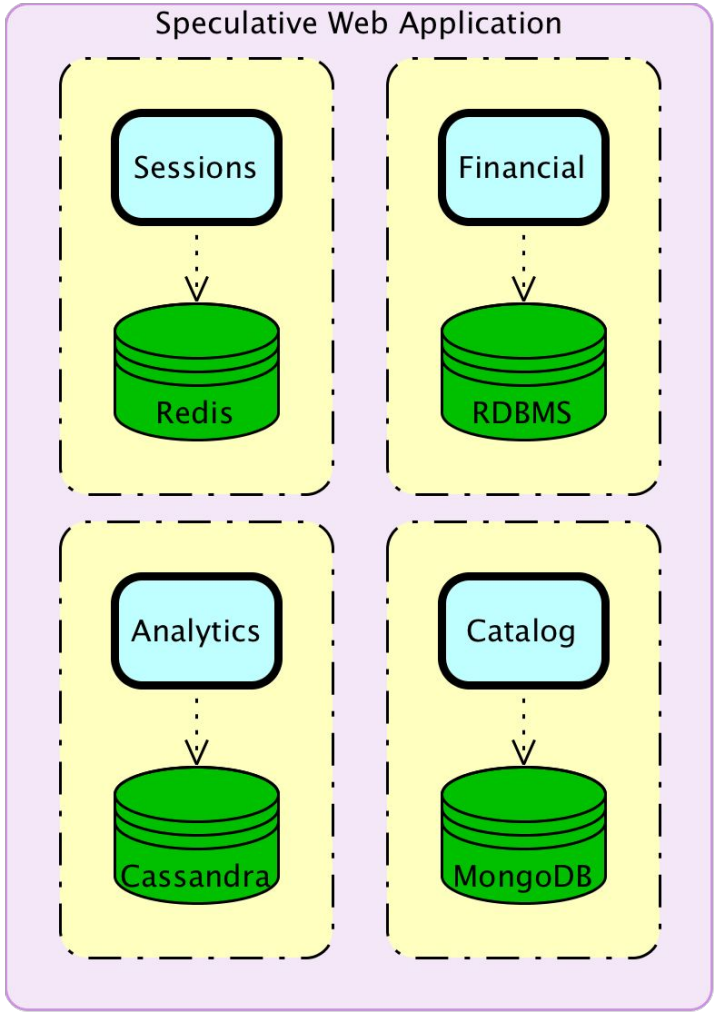
Sometimes business identifiers allow services to detect duplicate requests.

Example:

createCustomer(email)

POLYGLOT PERSISTENCE

A significant advantage of decentralized data management is the ability to take advantage of polyglot persistence



POLYGLOT PERSISTENCE

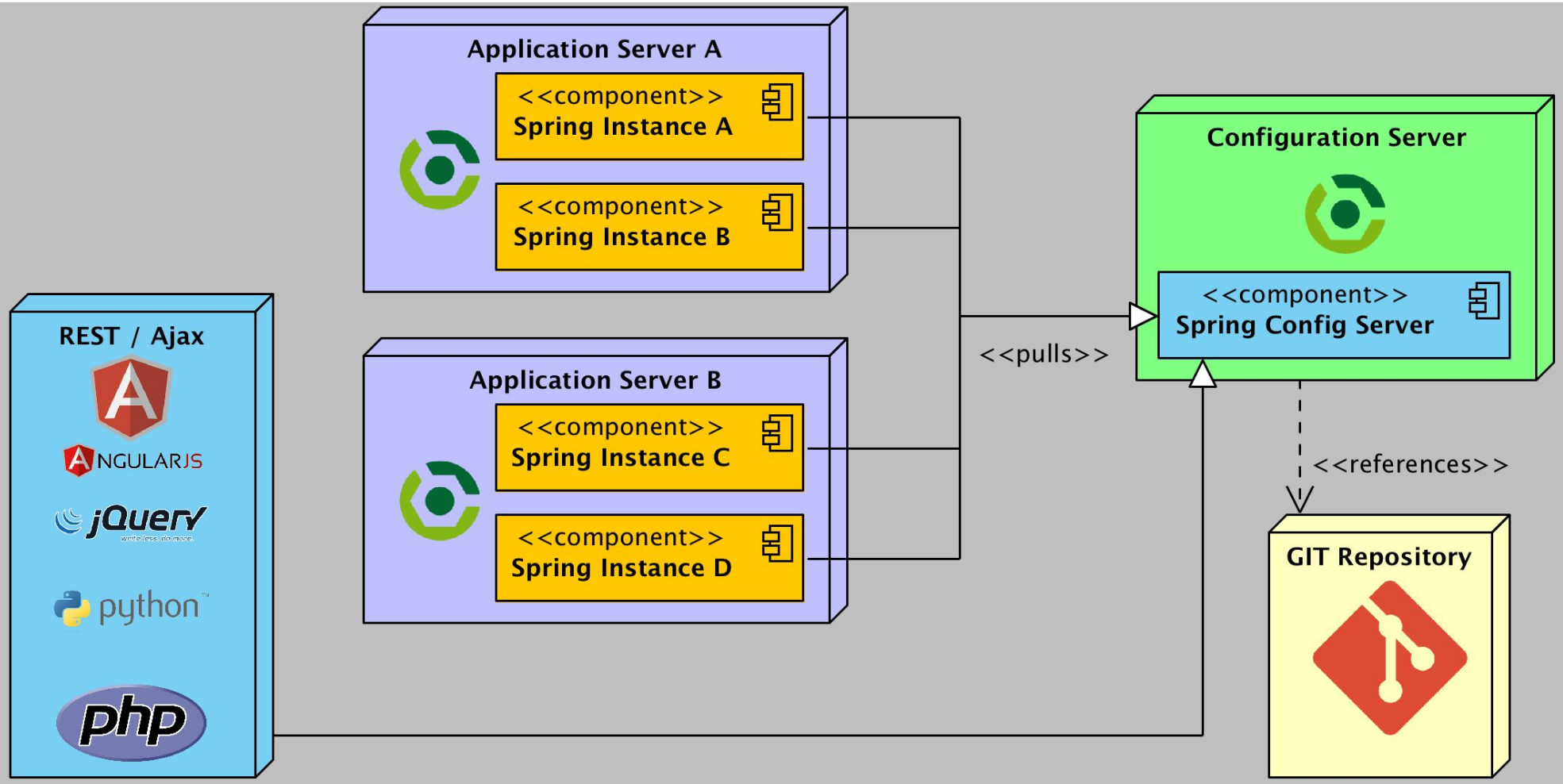
Different types of data have different storage requirements:

- Read/Write Balance (Some types of data have a very high write volume. This can require a different type of data store compared to data that has low write volume but high read volume.)
- Data Structure (Some types of highly structured data such as JSON documents may be better stored in a NoSQL database such as MongoDB, while flat relational objects may be more efficiently stored in a SQL database.)
- Data Querying (Some data may be accessible using a simple key value store, while other types of data may require advanced querying based on the values of multiple columns.)
- Data Lifecycle (Some data is temporary in nature and can be stored in a fast, in-memory store such as Redis or Memcached, while other data must be retained for all time and needs very durable storage on disk.)
- Data Size (Some data is made up of fairly uniform rows of consistent byte size, while other data may include large blobs that need to be stored in something like AWS S3.)

MICROSERVICE CONFIGURATION

Microservices require externalized configuration as per the 12-factor app requirement to inject different configuration for different environments.

- Configuration service is another service in the mix
- Configuration clients can request configuration based on
  - Service Name
  - Service Type
  - Service Environment
  - Service Version



MICROSERVICE CONFIGURATION

API GATEWAYS

A single point of entry into the system.

The API is used to encapsulate the entire system and provide only the required API for the service to the client.

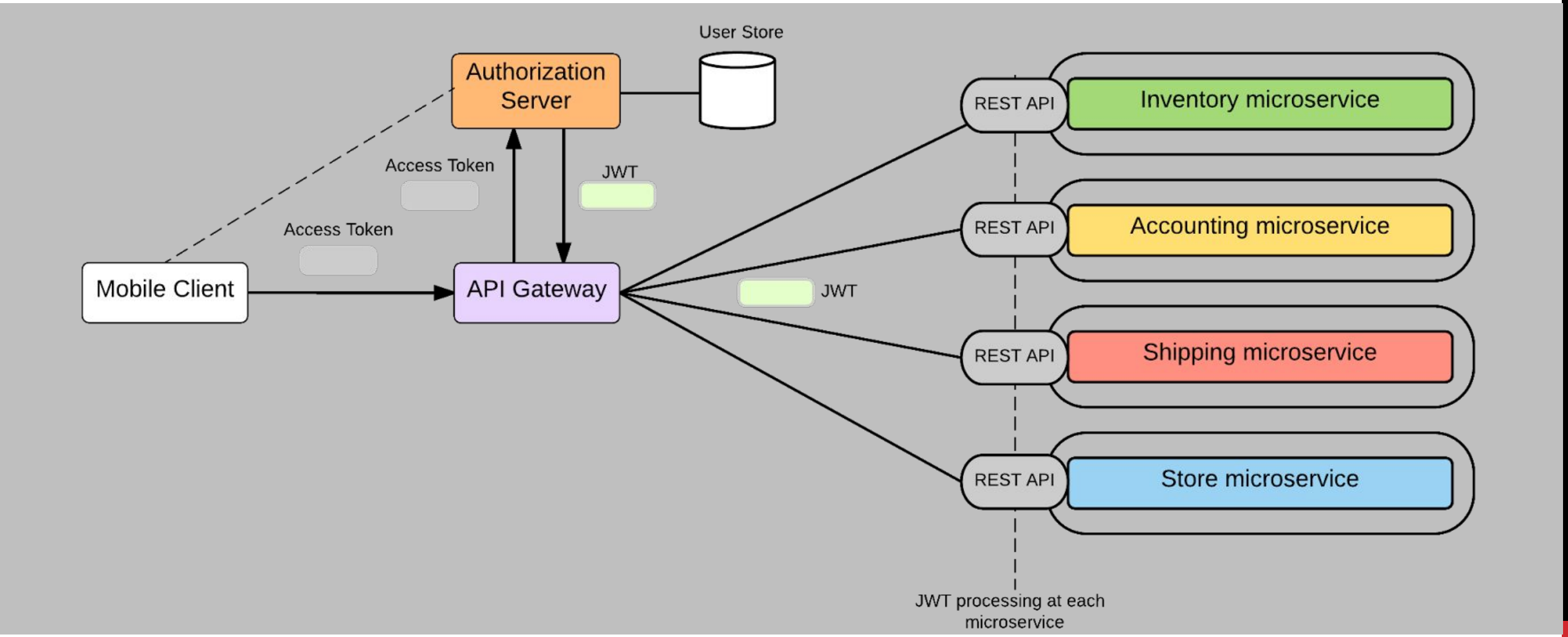
The nature of the API definition depends on which IPC mechanism you are using.

- If you are using messaging, the API consists of the message channels and the message types.
- If you are using HTTP, the API consists of the URLs and the request and response formats.

API GATEWAYS (CONT'D)

It might have other responsibilities such as:

- Authentication
- Monitoring
- Load balancing
- Caching
- Static response handling
- More…



API GATEWAY

API GATEWAY CONSIDERATIONS

Benefits

- A major benefit of using an API Gateway is that it encapsulates the internal structure of the application.

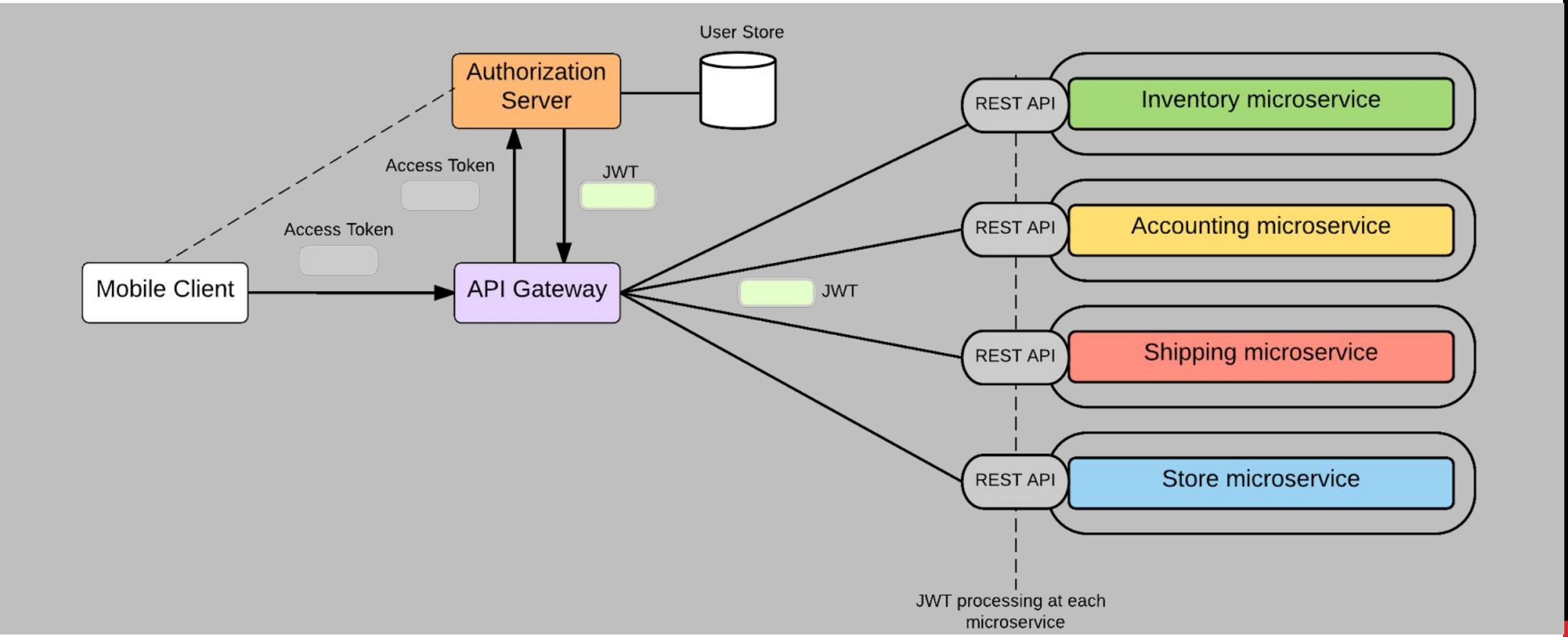
Drawbacks

- There is also a risk that the API Gateway becomes a development bottleneck.
- Developers must update the API Gateway in order to expose each microservice's endpoints.

API GATEWAY (AUTHENTICATION)

Authentication

API Gateways usually handle the authentication and authorization from the external callers to the microservice level.



API GATEWAY

API GATEWAY (AUTHENTICATION)

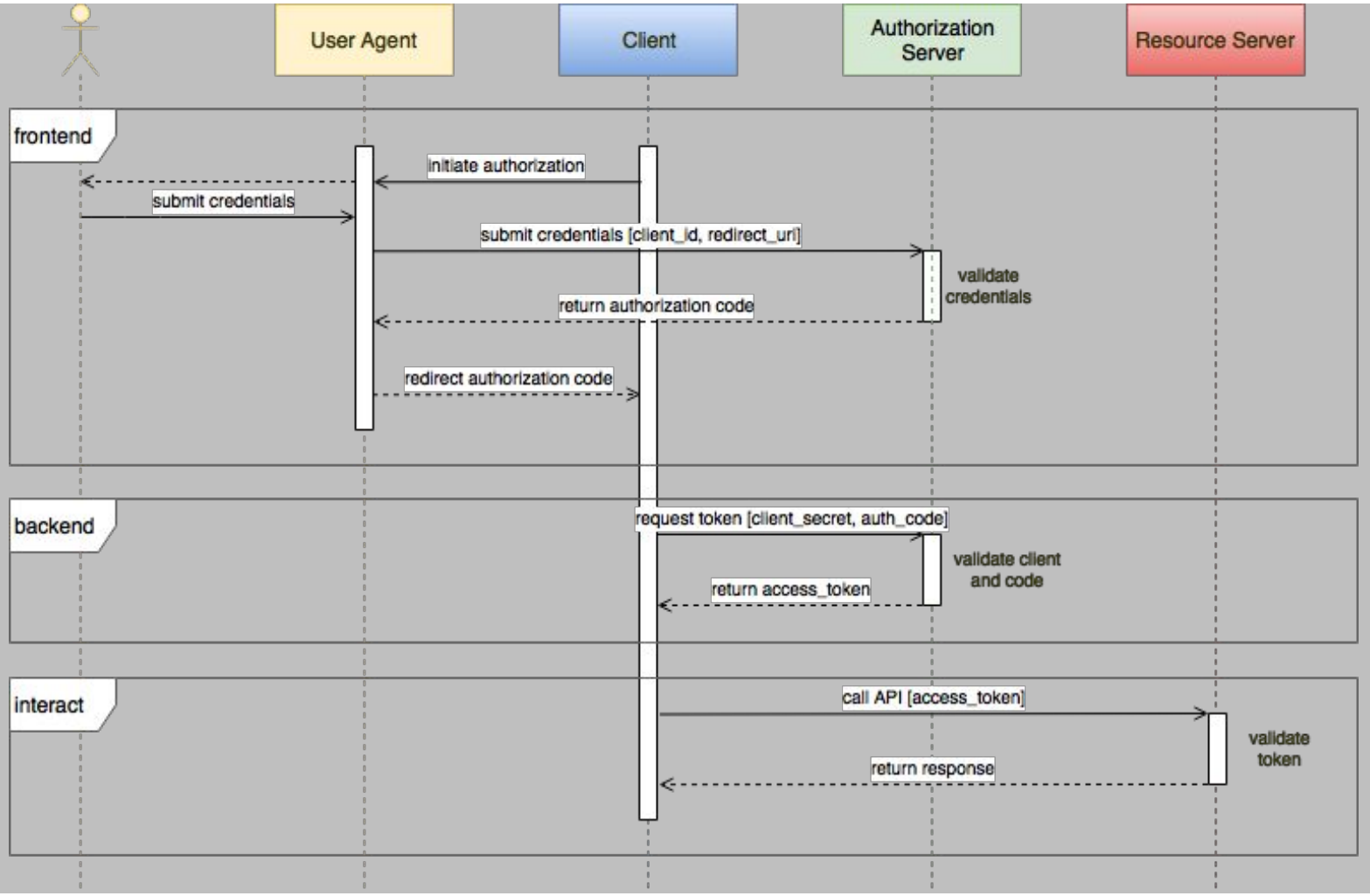
OAuth 2.0

OAuth delegated authorization with JSON Web Tokens (JWT) is an efficient and scalable solution for authentication and authorization for microservices.

API GATEWAY (AUTHENTICATION)

JSON Web Token (JWT)

A JSON web token, or JWT (“jot”) for short, is a standardized, optionally validated and/or encrypted container format that is used to securely transfer information between two parties.



API GATEWAY (OAUTH)

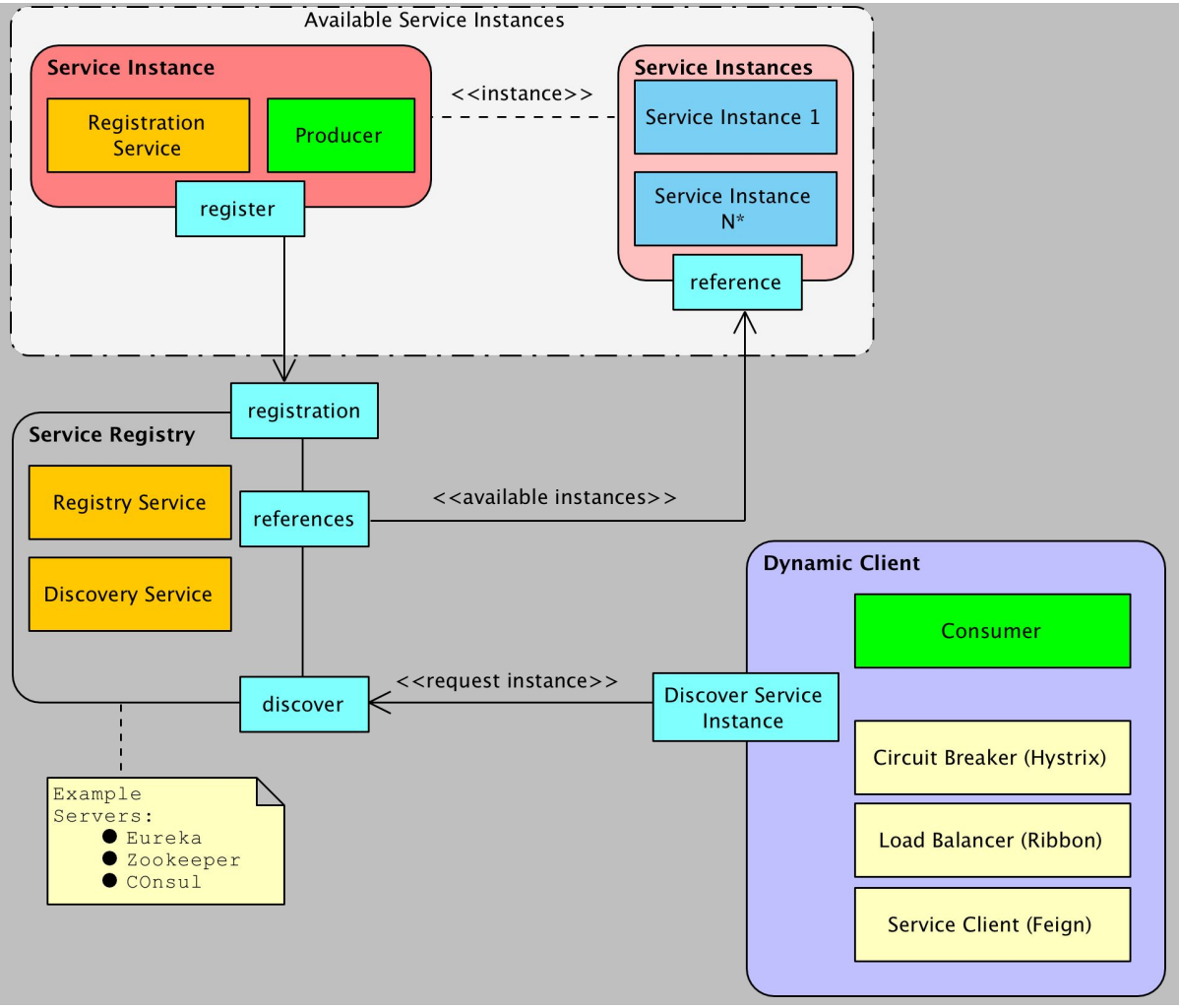
OAuth and JWT Authentication

API PROXYING

In simple terms you can think of the Proxy providing you a new endpoint for an existing API

Adds some lightweight security, monitoring, etc.

MICROSERVICE REGISTRY / DISCOVERY



MICROSERVICE REGISTRY / DISCOVERY

MICROSERVICE REGISTRY / DISCOVERY

Context

Clients of a service use either Client-side discovery or Server-side discovery to determine the location of a service instance to which to send requests.

MICROSERVICE REGISTRY / DISCOVERY

Issue

How do clients of a service (in the case of Client-side discovery) and/or routers (in the case of Server-side discovery) know about the available instances of a service?

MICROSERVICE REGISTRY / DISCOVERY

Forces

- Each instance of a service exposes a remote API such as HTTP/REST at a particular location (host:port)
- Number of services instances and their locations changes dynamically.
- Virtual machines and containers are assigned a dynamic IP address.
- Can adjust the number of instances based on load.

MICROSERVICE REGISTRY / DISCOVERY

Solution

- Implement a service registry, which is a database of services, their instances and their locations.
- Service instances are registered with the service registry on startup and deregistered on shutdown.
- Client of the service and/or routers query the service registry to find the available instances of a service.
- A service registry might invoke a service instance's health check API to verify that it is able to handle requests

MICROSERVICE DISCOVERY (SERVER-SIDE)

MICROSERVICE DISCOVERY (SERVER-SIDE)

Issue

- How does the client of a service - the API gateway or another service - discover the location of a service instance?

MICROSERVICE DISCOVERY (SERVER-SIDE)

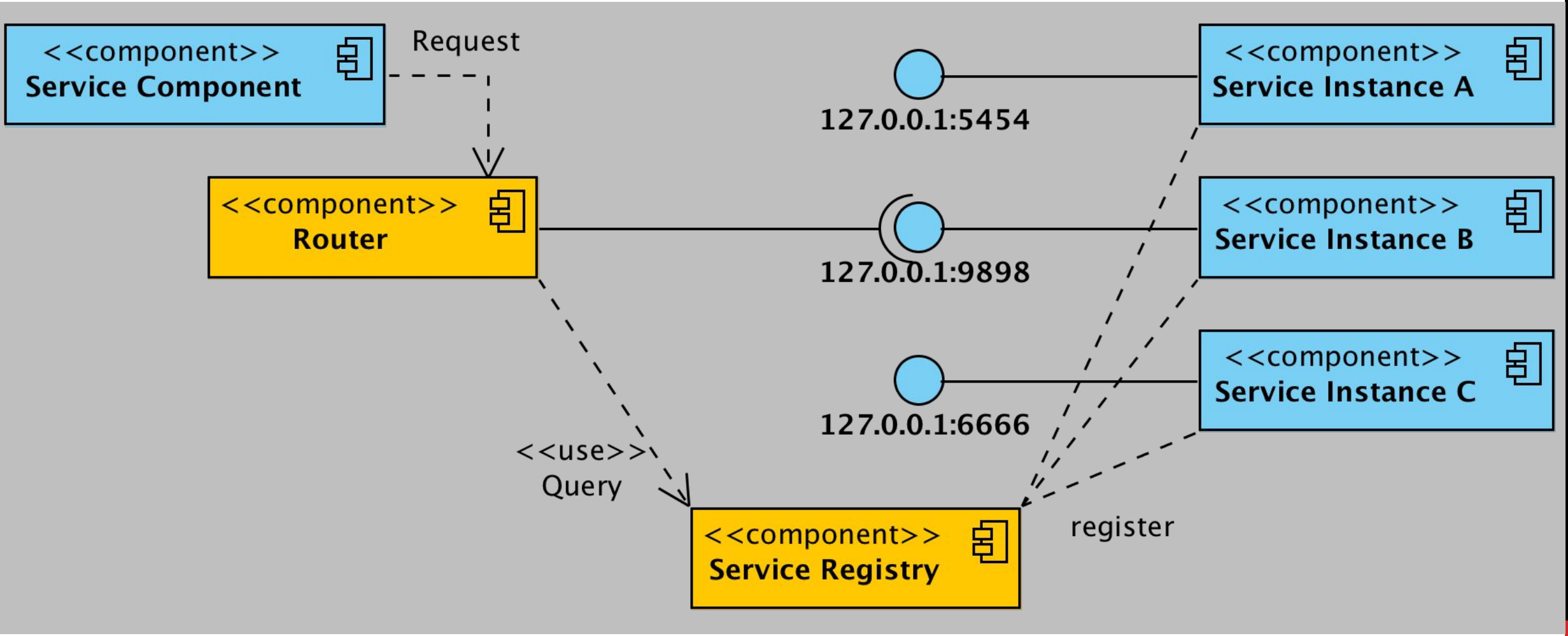
Forces

- Each instance of a service exposes a remote API such as HTTP/REST at a particular location (host:port)
- Number of services instances and their locations changes dynamically.
- Virtual machines and containers are assigned a dynamic IP address.
- Can adjust the number of instances based on load.

MICROSERVICE DISCOVERY (SERVER-SIDE)

Solution

- When making a request to a service, the client makes a request via a router (load balancer) that runs at a well known location.
- Router queries a service registry, which might be built into the router, and forwards the request to an available service instance.



MICROSERVICE DISCOVERY (SERVER-SIDE)

MICROSERVICE DISCOVERY (CLIENT-SIDE)

MICROSERVICE DISCOVERY (SERVER-SIDE)

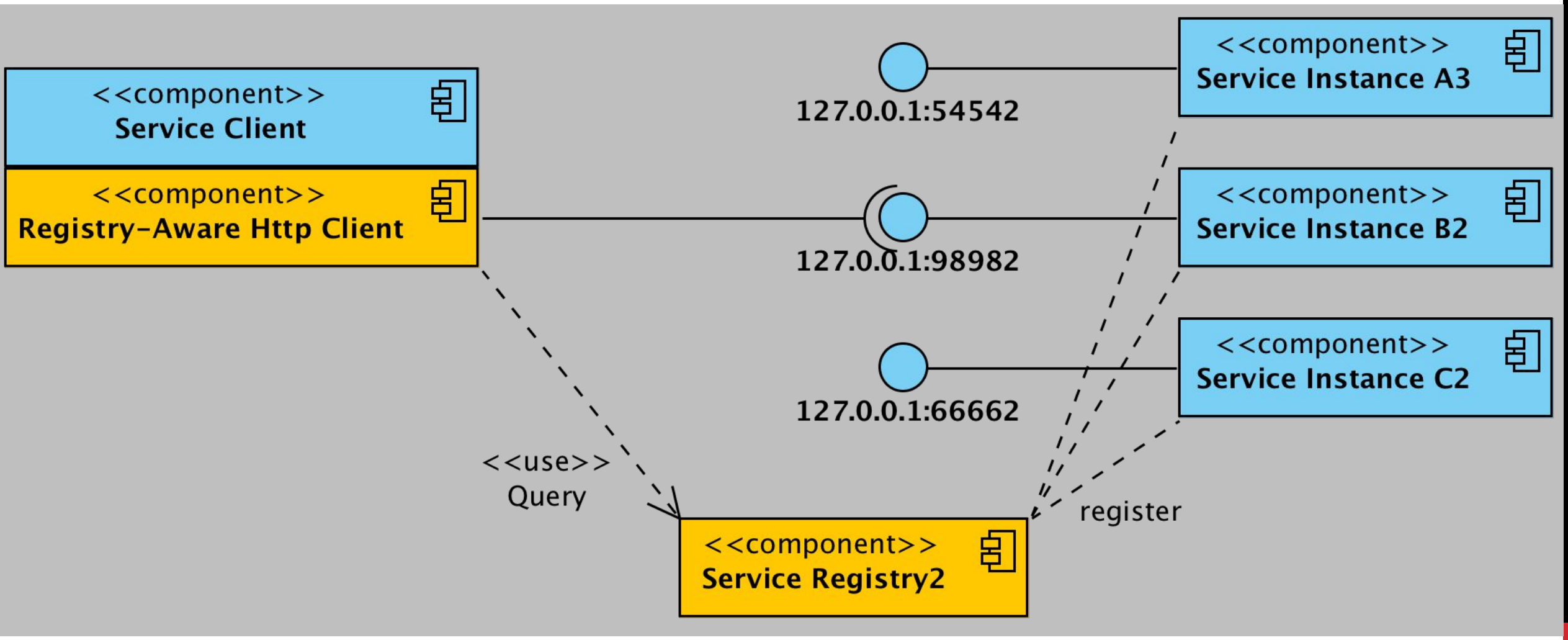
Forces

- Each instance of a service exposes a remote API such as HTTP/REST at a particular location (host:port)
- Number of services instances and their locations changes dynamically.
- Virtual machines and containers are assigned a dynamic IP address.
- Can adjust the number of instances based on load.

MICROSERVICE DISCOVERY (SERVER-SIDE)

Solution

- When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.

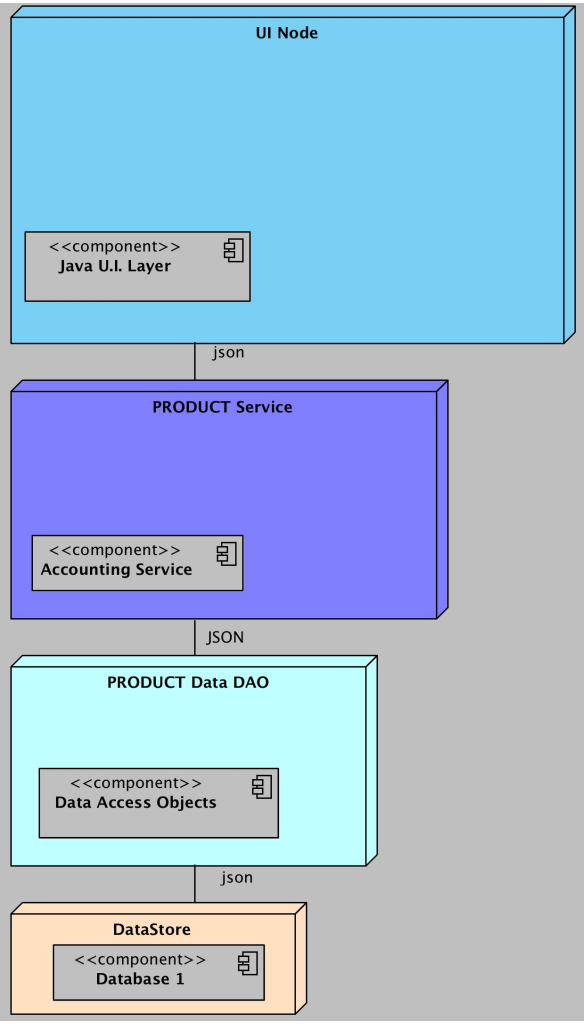


MICROSERVICE DISCOVERY (CLIENT-SIDE)

FLEXIBLE SCALING OPTIONS

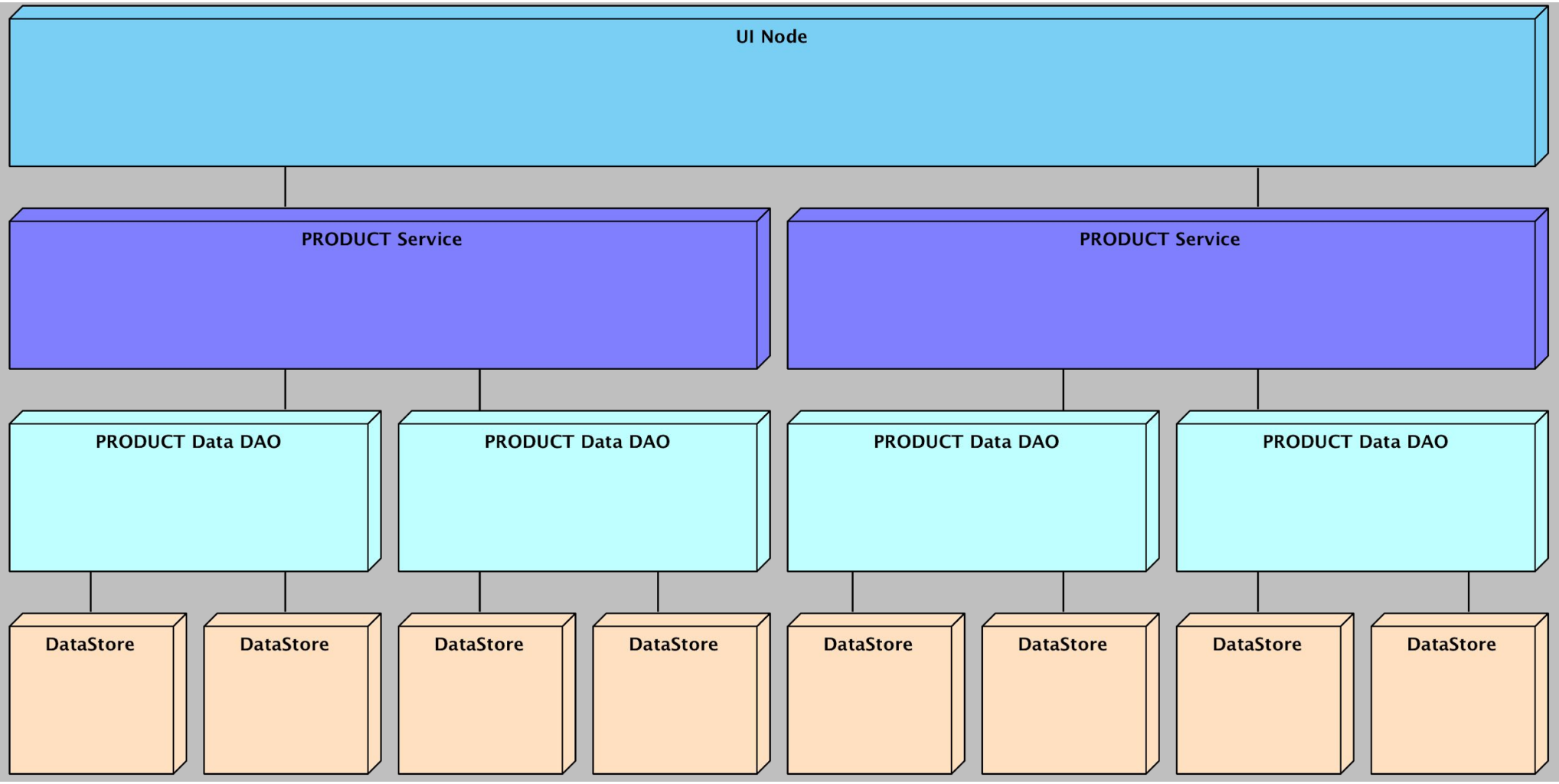
Each service has:

- Independent release schedule
- Independent load requirements
- Independent scaling strategy



SCALING MICROSERVICES (VERTICALLY)

Larger instances



SCALING MICROSERVICES (HORIZONTALLY)

More instances

SERVICE ORCHESTRATION

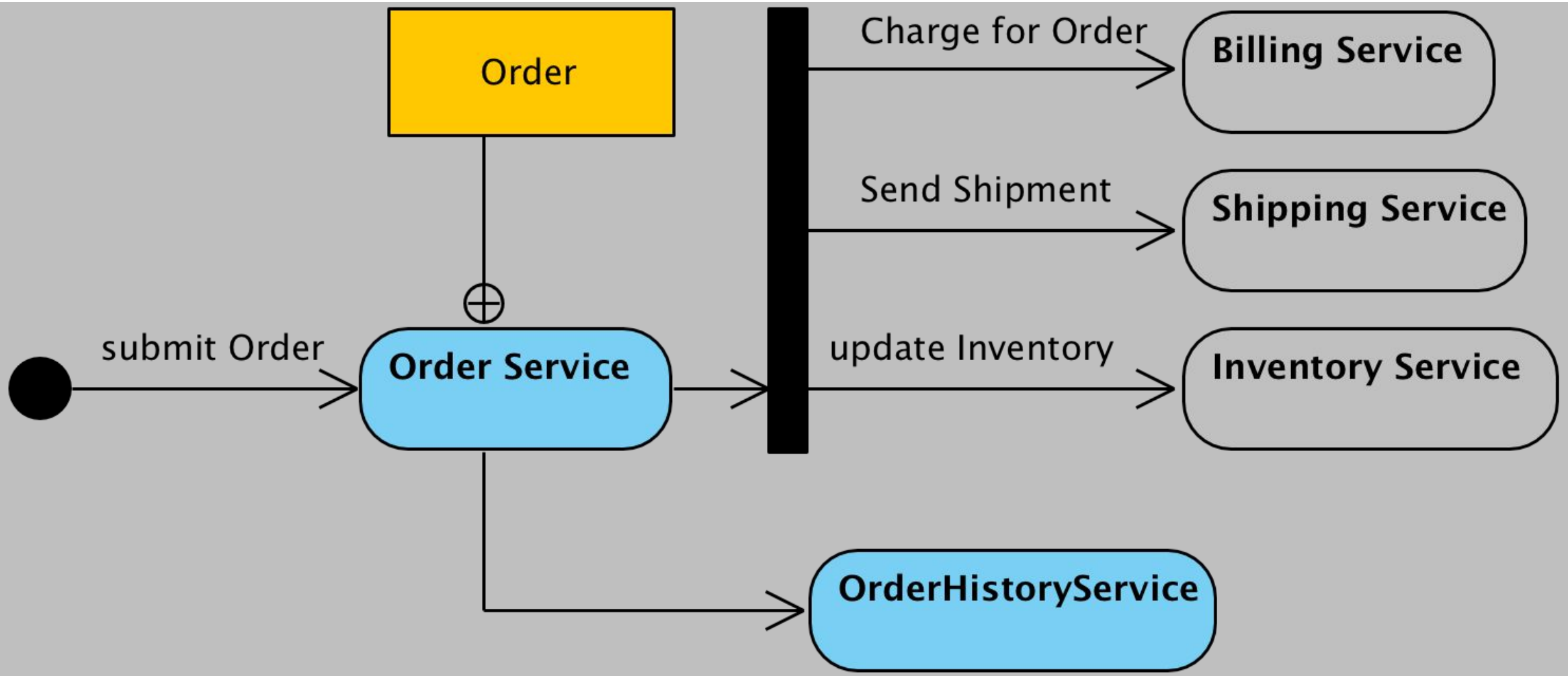
Who is in charge of composing services

- In SOA, ESB often handles
- Service provider may handle
    - Orchestration uses a central service acts as the controller
        - Orchestrates the other services based on decision flow
    - Choreography has no central controller
        - A single event is published and sent to consumers
        - Each consumer of the event performs is function

From microservices perspective, prefer choreography

- Controller as an external component or another microservice is counter to the autonomy principle

There will be situations where some degree of orchestration is required



SERVICE ORCHESTRATION

ORCHESTRATION CASES

Issue

When orchestration is required, choices become more difficult and need to be examined carefully.

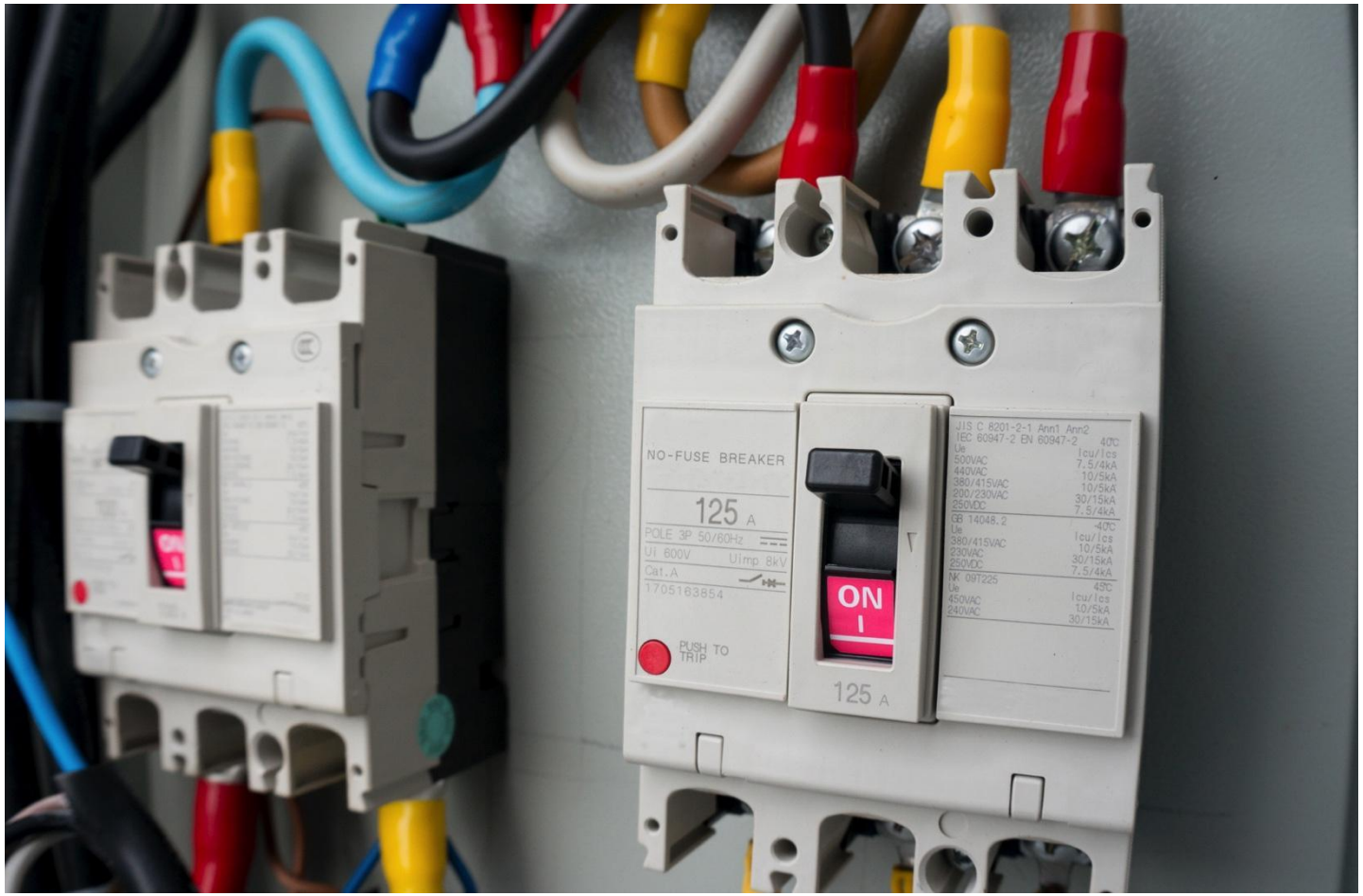
ORCHESTRATION CASES

Options include:

- Adding in a synchronous call to each external service and wait for result
- Creating a specific microservice whose job it is to compose the component services
- Replicate the one or more of the component services into one of the higher level services

CIRCUIT BREAKER

How to prevent a network or service failure from cascading to other services?



CIRCUIT BREAKER

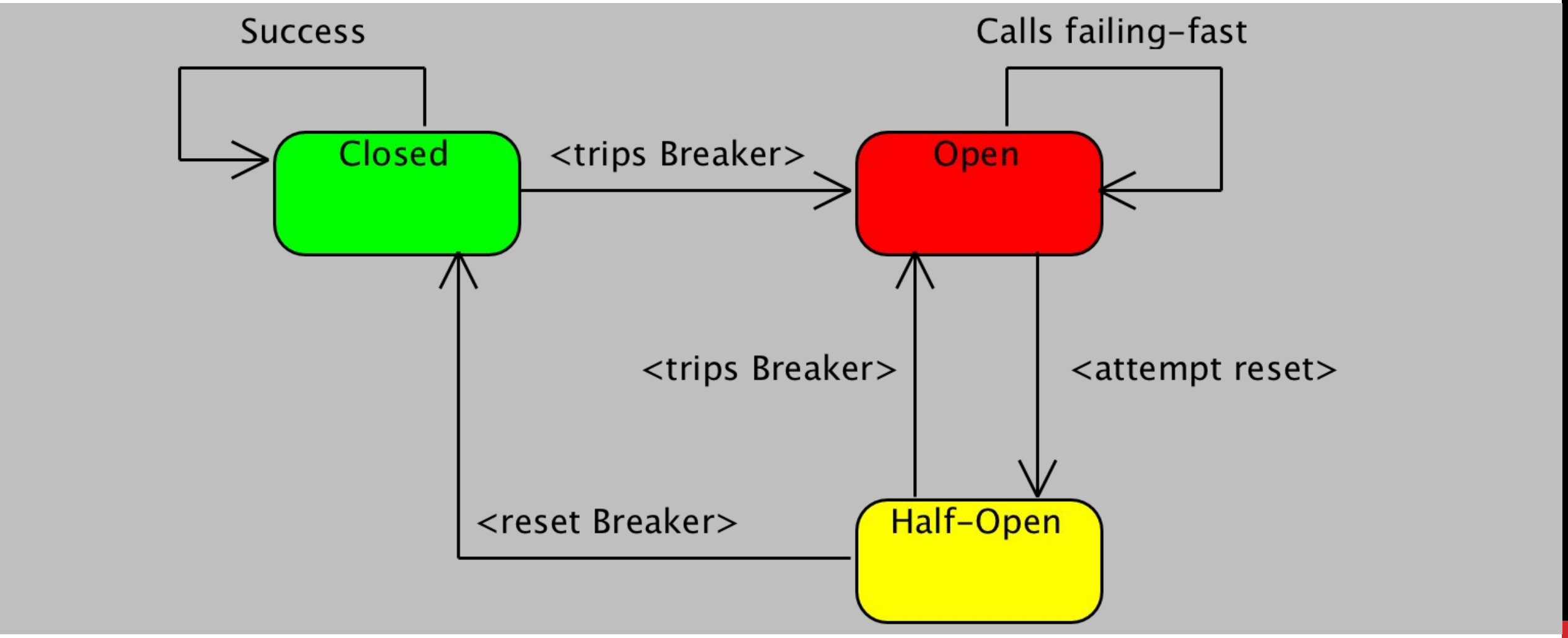
Solution

- A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.
- When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

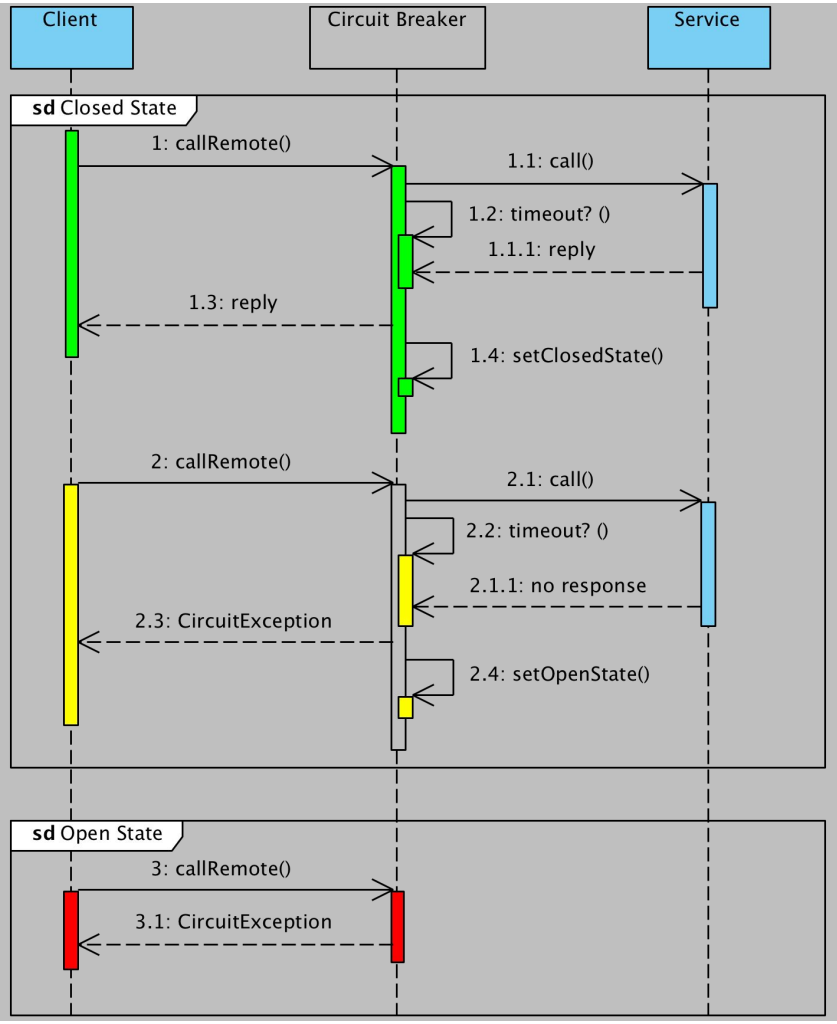
CIRCUIT BREAKER (CONT'D)

Solution

- After the timeout expires the circuit breaker allows a limited number of test requests to pass through.
- If those requests succeed the circuit breaker resumes normal operation.
- Otherwise, if there is a failure the timeout period begins again.



CIRCUIT BREAKER



CIRCUIT BREAKER

THANK YOU FOR ATTENDING