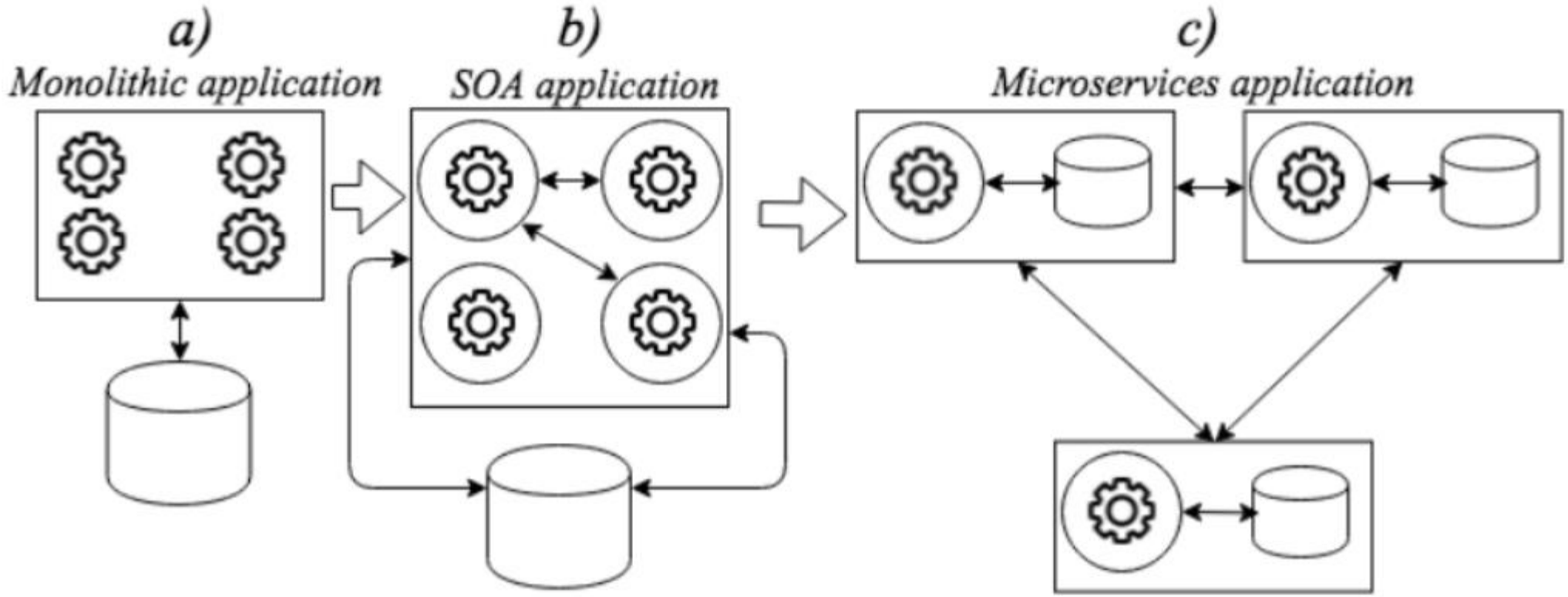
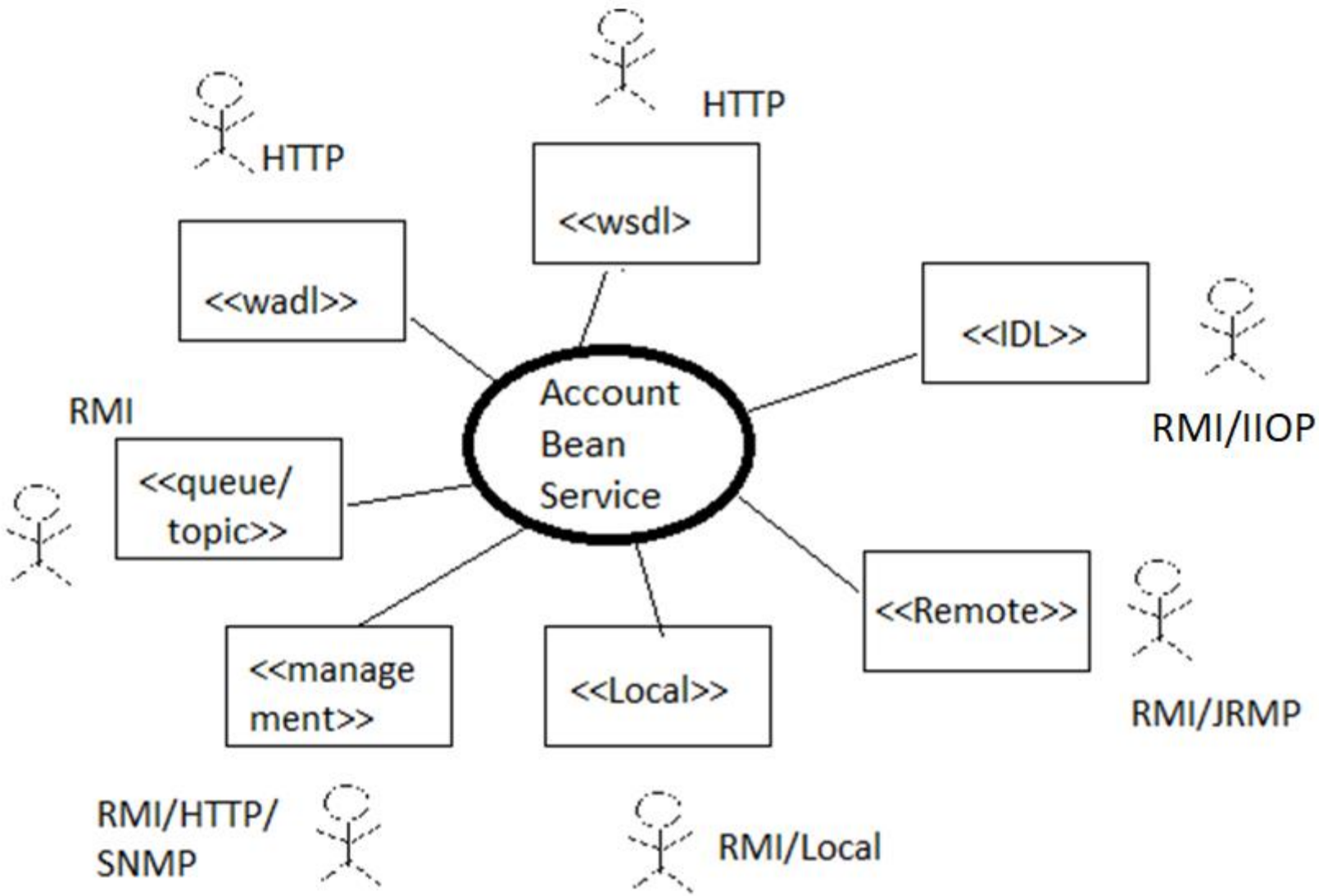
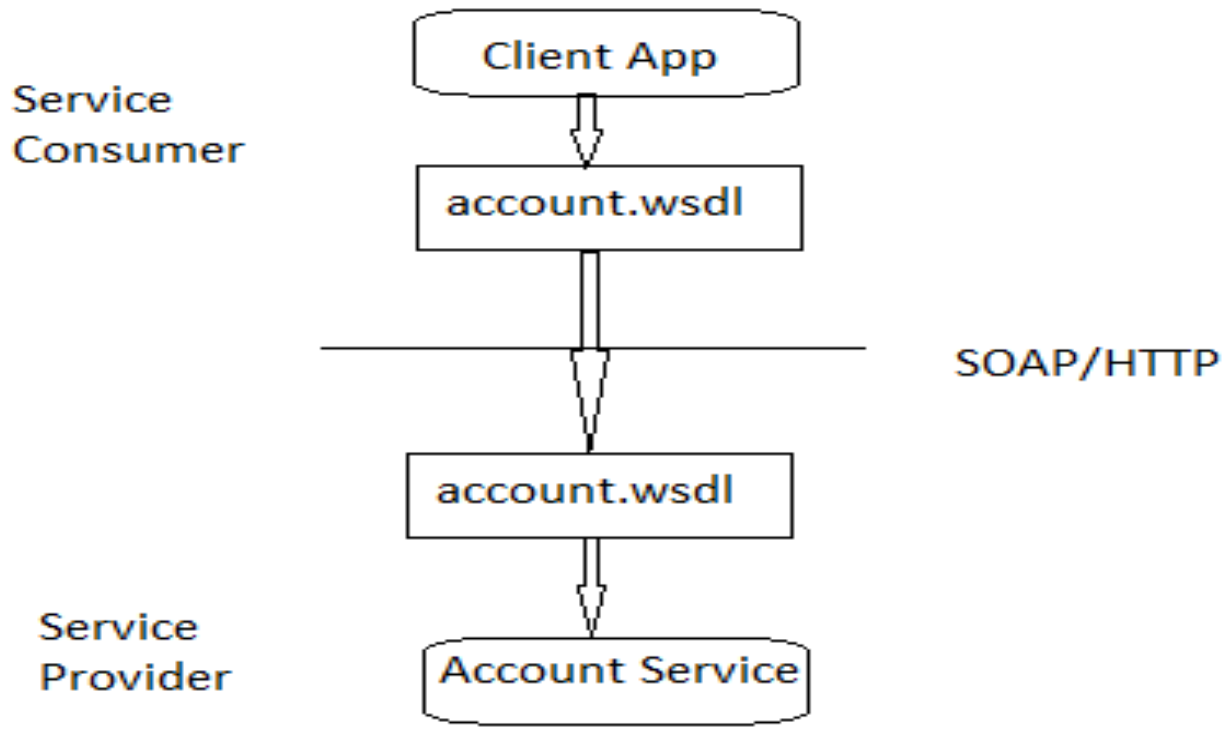
Introduction To Microservice



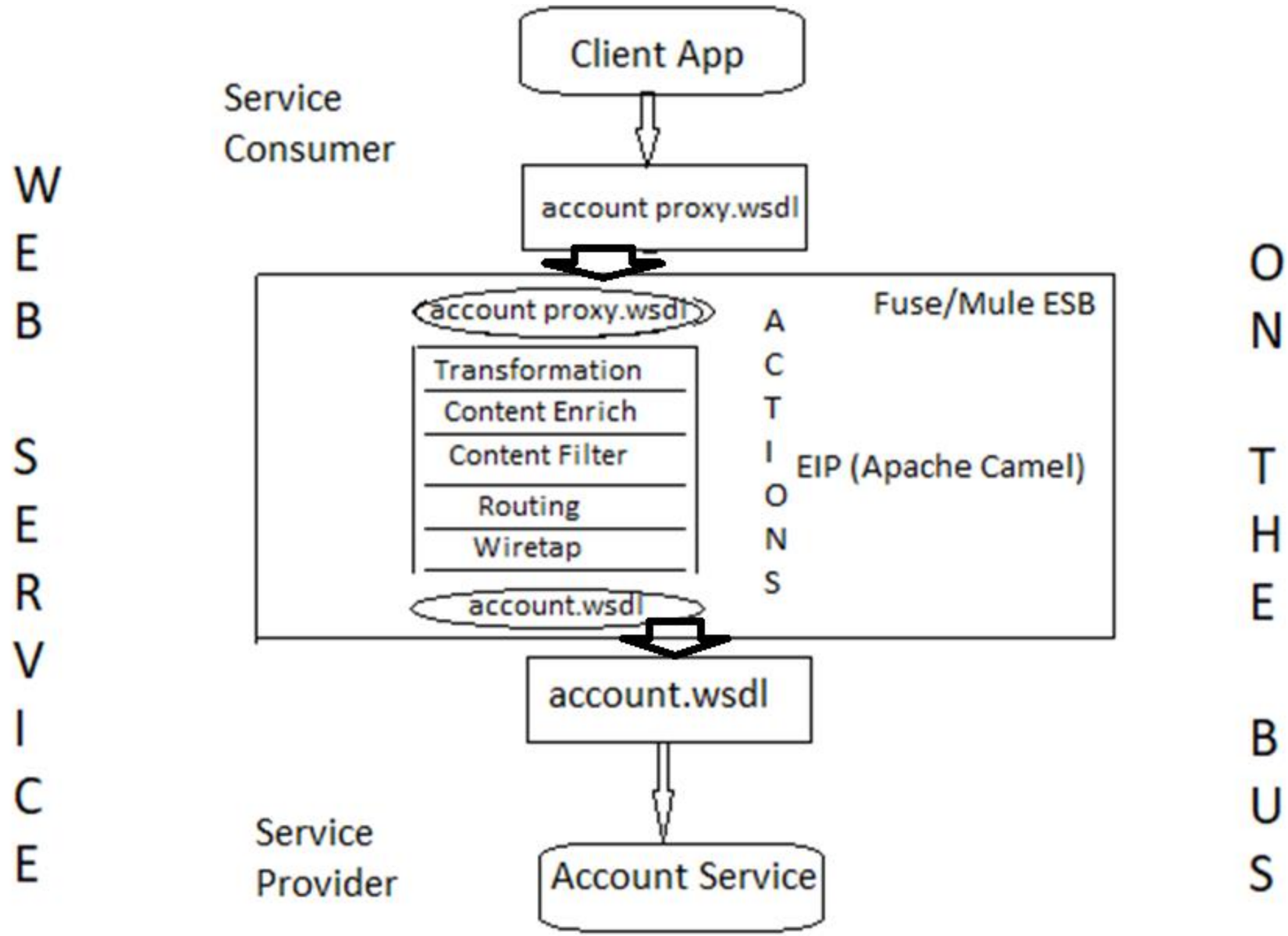
Evolution from monolithic applications to microservices-based systems

Distributed computing and web services





Webserivce OFF the BUS


WEB SERVICE
Service Consumer
Client App
account proxy.wsdl
account proxy.wsdl
Transformation
Content Enrich
Content Filter
Routing
Wiretap
account.wsdl
ACTIONS
Fuse/Mule ESB
EIP (Apache Camel)
account.wsdl
Service Provider
Account Service
ON THE BUS

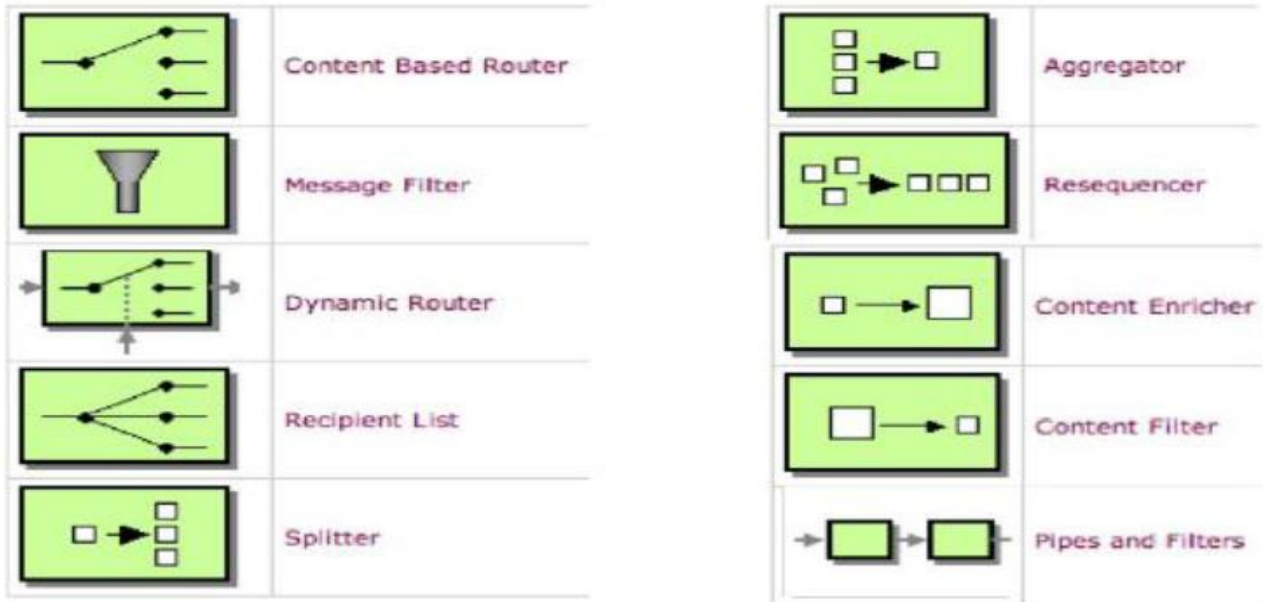
SOA

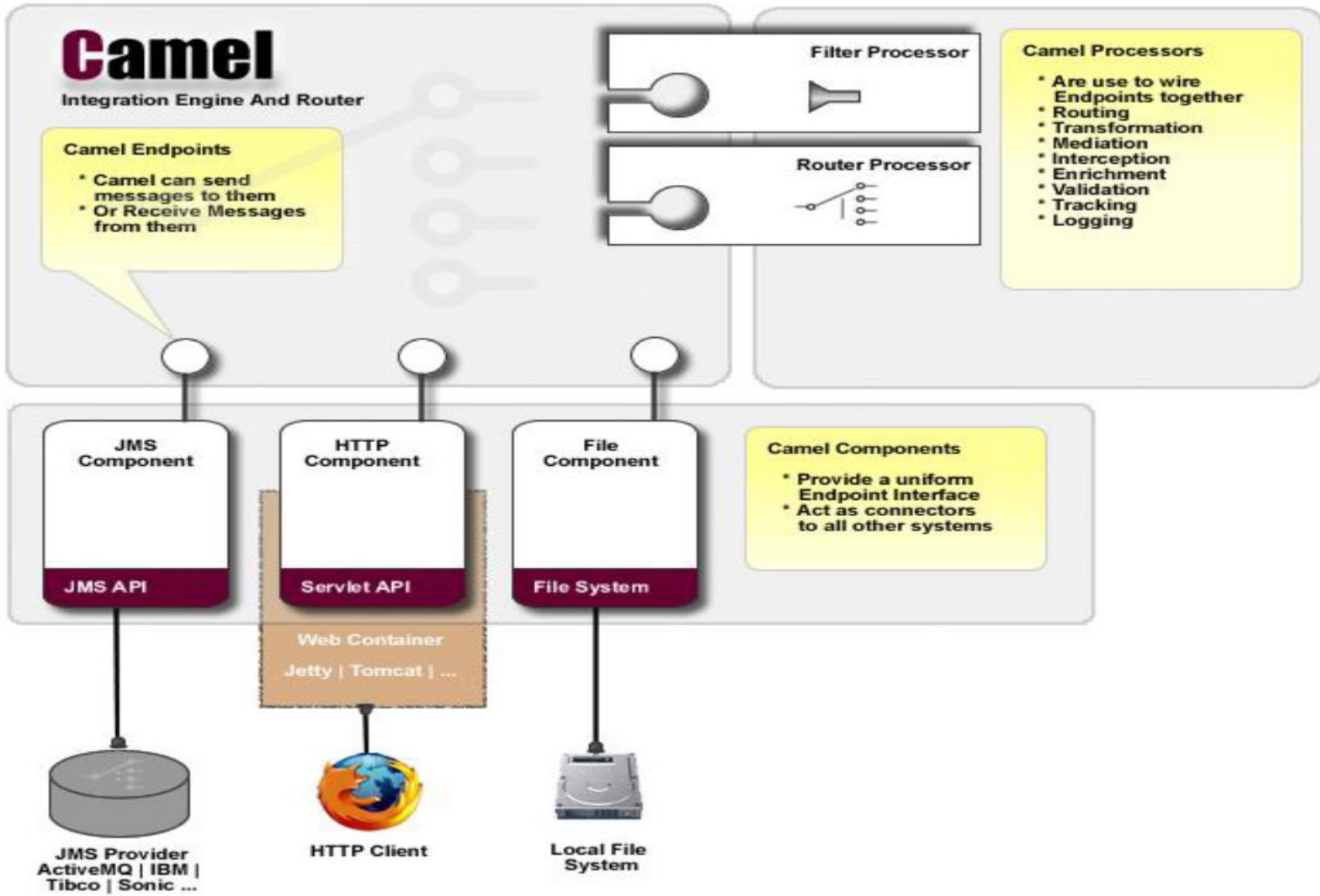
Service Oriented Architecture (SOA) is an architectural style for implementing business processes as a set of loosely-coupled services.

Apache Camel is a
powerful Open Source
Integration Framework
based on known
Enterprise Integration Patterns

What is Apache Camel?

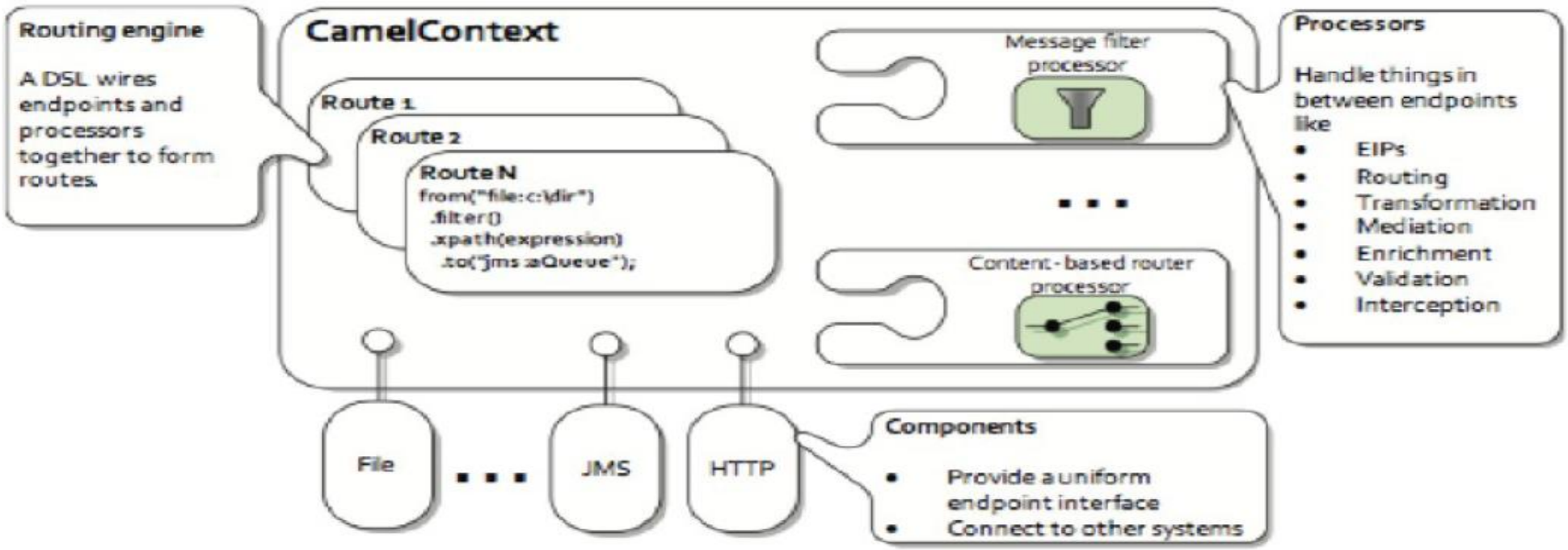
- Enterprise Integration Patterns




Camel
Integration Engine And Router
Camel Endpoints
* Camel can send messages to them
* Or Receive Messages from them
Filter Processor
Router Processor
Camel Processors
* Are use to wire Endpoints together
* Routing
* Transformation
* Mediation
* Interception
* Enrichment
* Validation
* Tracking
* Logging
JMS Component
HTTP Component
File Component
Camel Components
* Provide a uniform Endpoint Interface
* Act as connectors to all other systems
JMS API
Servlet API
File System
Web Container
Jetty | Tomcat | ...
JMS Provider
ActiveMQ | IBM | Tibco | Sonic ...
HTTP Client
Local File System

What is Apache Camel?

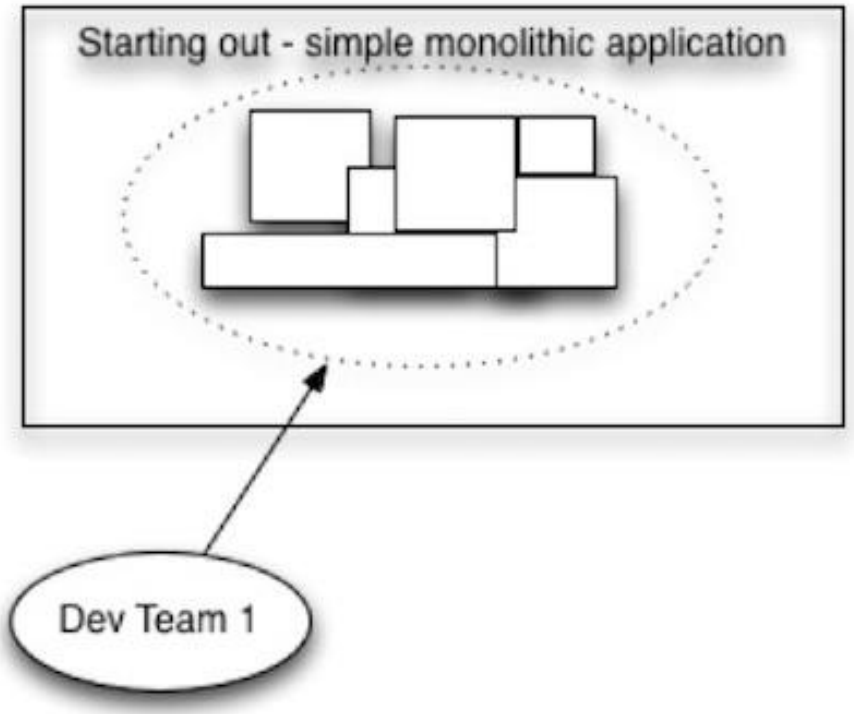
- Camel's Architecture



Monolith Application:

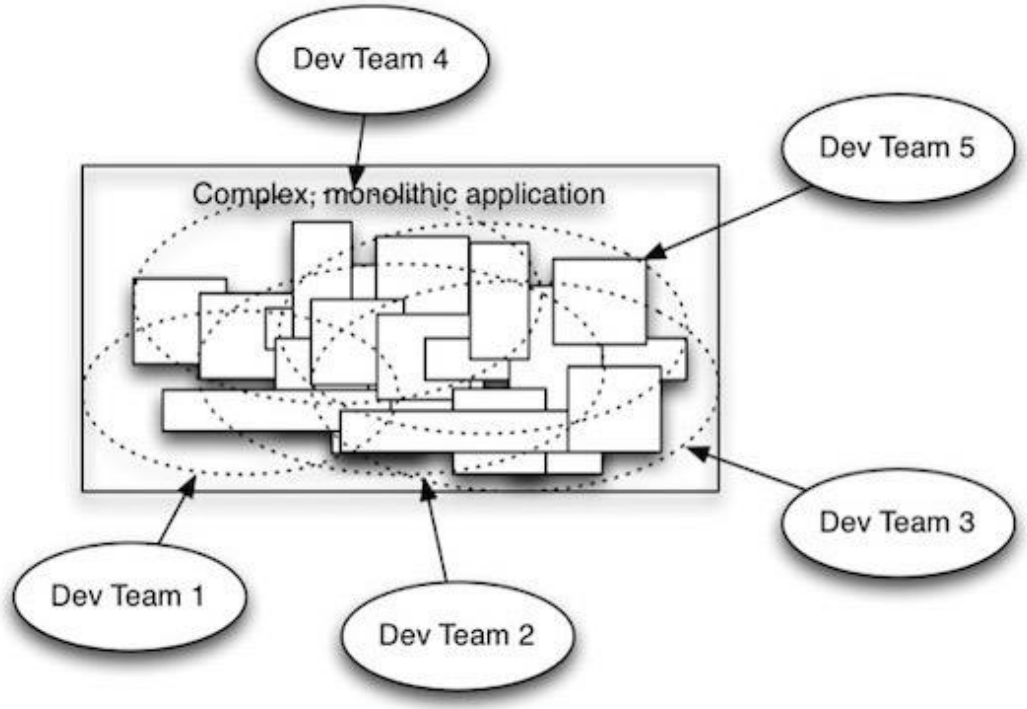
A single monolith would contain all the code for all the business activities an application performed.

We write a simple application, which is developed and managed by a single team.



But what happens if your application turns out to be successful? Users like it and begin to depend on it.

Traffic increases dramatically. And almost inevitably, users request improvements and additional features, so more developers are roped in to work on the growing application. Before too long, your application looks more like this:



What is a Microservices Architecture in a Nutshell?

“single responsibility principle” which states “gather together those things that change for the same reason, and separate those things that change for different reasons.”

A microservices architecture takes this same approach and extends it to the loosely coupled services which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.

Example

A class or module should have one, and only one, reason to be changed (i.e. rewritten).

As an example, consider a module that compiles and prints a report.

Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change.

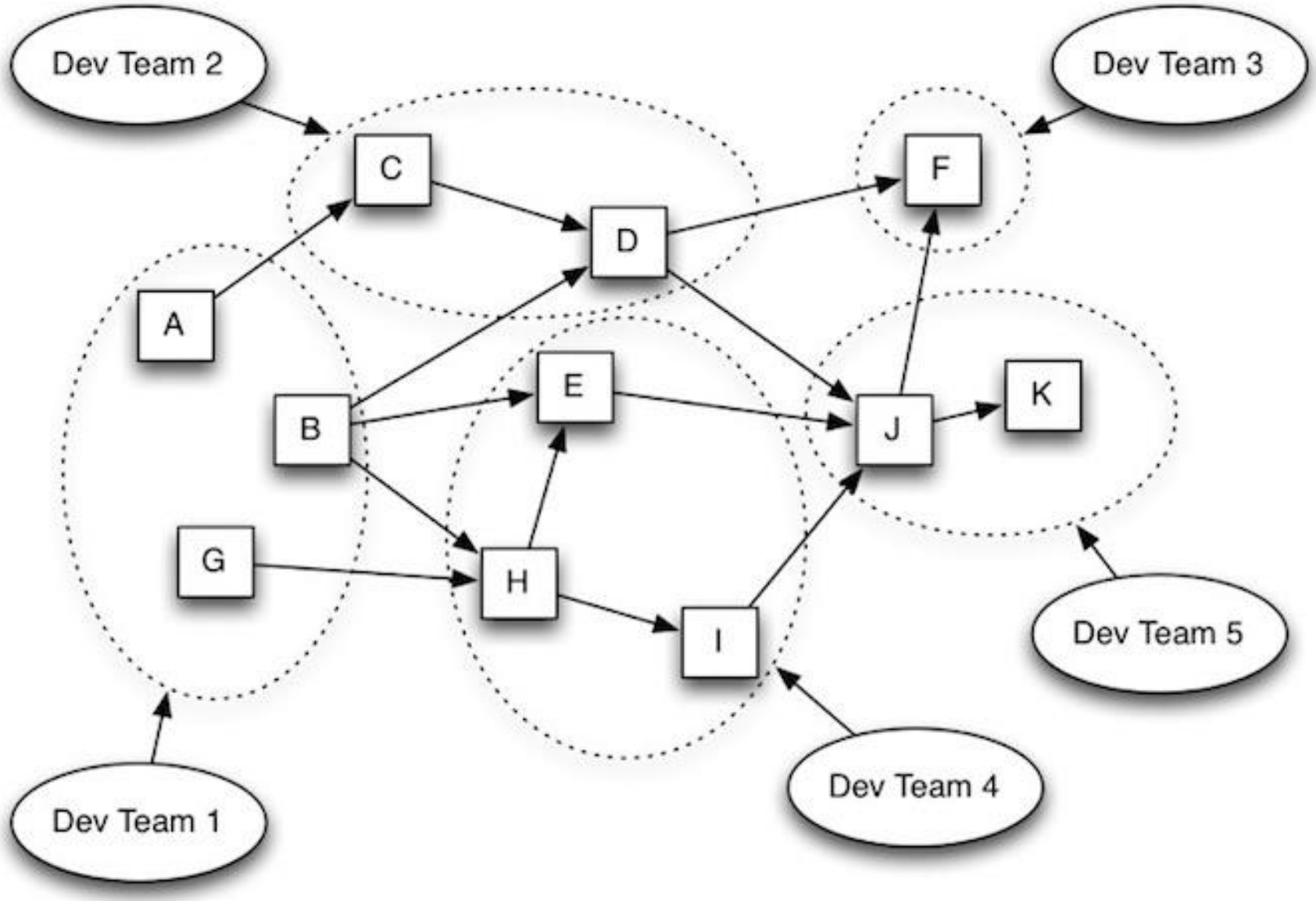
The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

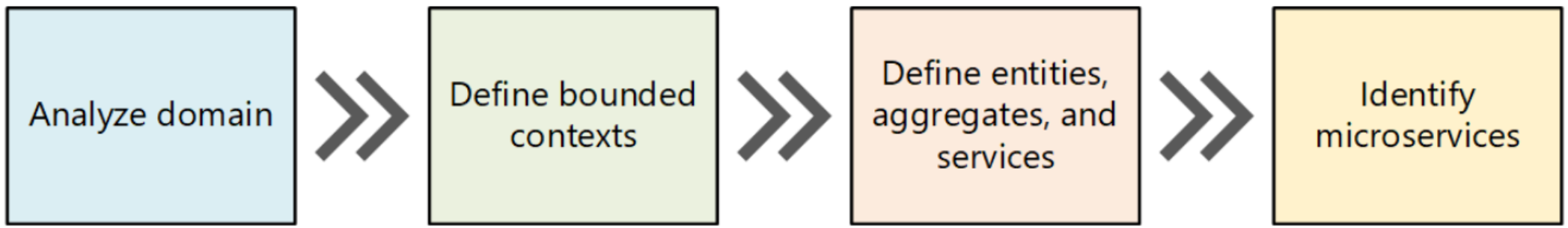
Applications built using microservices possess certain characteristics.

In particular, they:

- Are fragmented into multiple modular, loosely coupled components, each of which performs a discrete function
- Have those individual functions built to align to business capabilities
- Can be distributed across clouds and data centres
- Treat each function as an independent service that can be changed, updated, or deleted without disrupting the rest of the application


Dev Team 2
Dev Team 3
C
F
D
A
E
K
B
J
G
H
I
Dev Team 5
Dev Team 1
Dev Team 4


push state change with
DOMAIN EVENTS
SERVICES
REPOSITORIES
access with
access with
express model with
express change with
ENTITIES
encapsulate with
express identity with
act as root of
AGGREGATES
encapsulate with
express state & computation with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
model gives structure to
isolate domain expressions with
encapsulate with
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
define model within
UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
names enter
SHARED KERNEL
SHARED KERNEL
BOUNDED CONTEXT
CORE DOMAIN
interdependent contexts form
overlap allied contexts through
CUSTOMER /SUPPLIER
relate allied contexts as
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
translate and insulate unilaterally with
loosely couple contexts through
formalize
GENERIC SUBDOMAINS
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE


Analyze domain
Define bounded contexts
Define entities, aggregates, and services
Identify microservices

:

- Start by analysing the business domain to understand the application's functional requirements. The output of this step is an informal description of the domain, which can be refined into a more formal set of domain models.

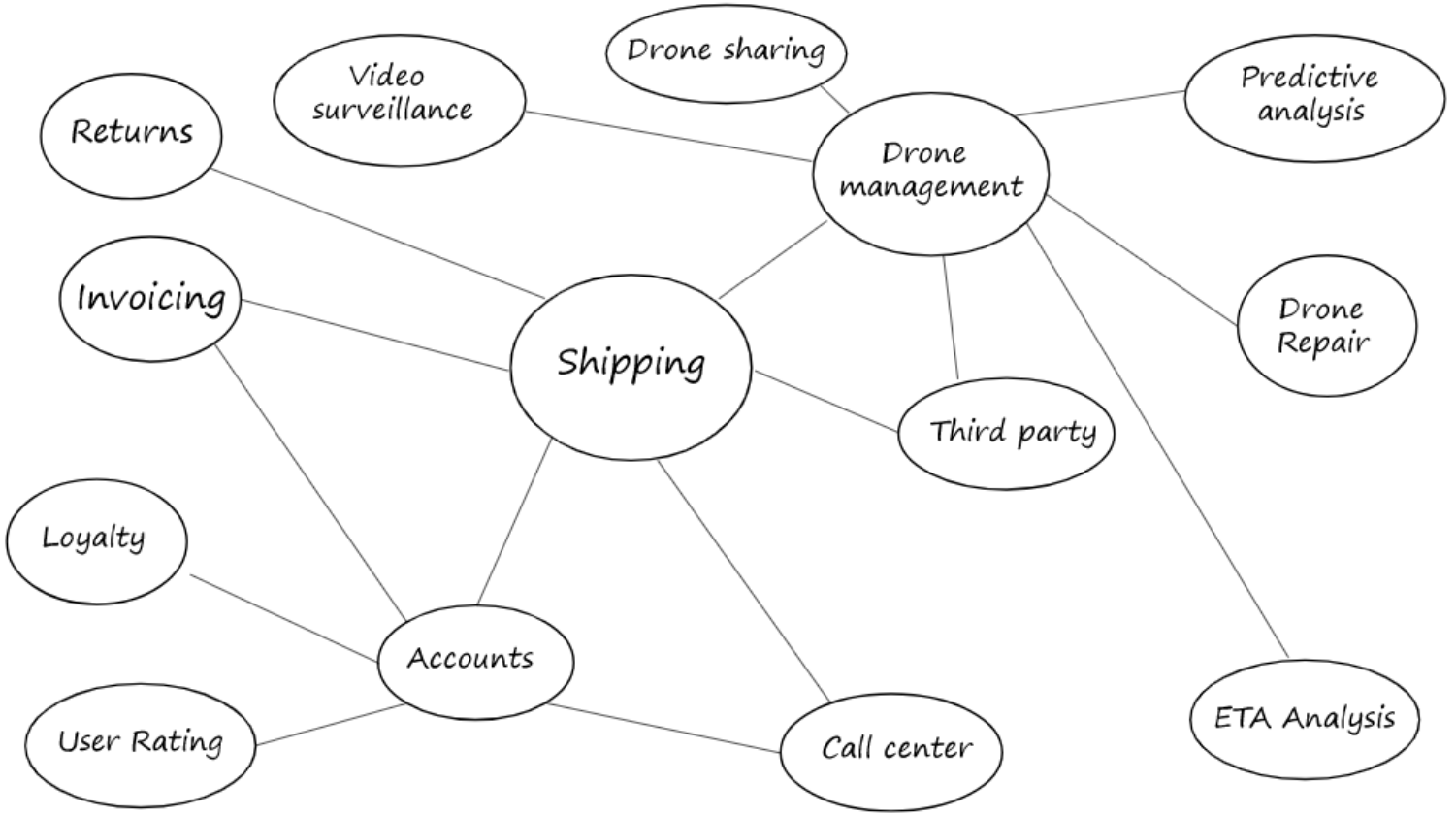
- Next, define the bounded contexts of the domain. Each bounded context contains a domain model that represents a particular subdomain of the larger application.

- Within a bounded context, apply tactical DDD patterns to define entities, aggregates, and domain services.

- Use the results from the previous step to identify the microservices in your application.

Example: Drone delivery application

After some initial domain analysis, the team came up with a rough sketch that depicts the Drone Delivery domain.



Define bounded contexts

The domain model will include representations of real things in the world — users, drones, packages, and so forth. But that doesn't mean that every part of the system needs to use the same representations for the same things.

For example, drone **repair and predictive analysis subsystems** will need to represent many physical characteristics of drones, such as their maintenance history, mileage, age, model number, performance characteristics, and so on.

The **Scheduling subsystem** only needs to know whether a drone is available.

ETA subsystem for pickup and delivery.

If we tried to create a single model for both of these subsystems, it would be unnecessarily complex.

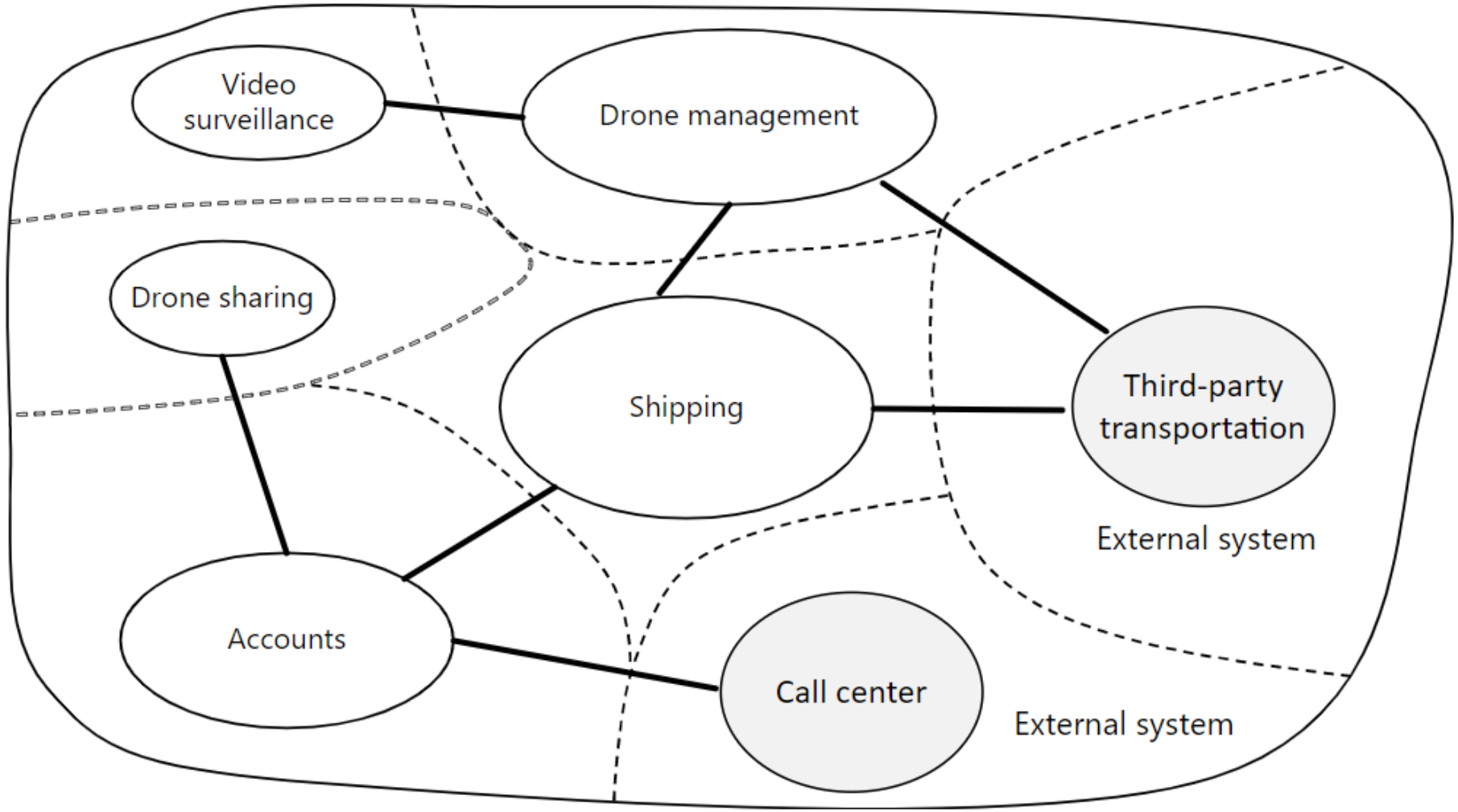
It would also become harder for the model to evolve over time, because any changes will need to satisfy multiple teams working on separate subsystems.

Therefore, it's often better to design separate models that represent the same real-world entity (in this case, a drone) in two different contexts. Each model contains only the features and attributes that are relevant within its particular context.

This is where the DDD concept of bounded contexts comes into play.

A bounded context is simply the boundary within a domain where a particular domain model applies.

Looking at the previous diagram, we can group functionality according to whether various functions will share a single domain model.


Video surveillance
Drone management
Drone sharing
Shipping
Third-party transportation
External system
Accounts
Call center
External system

Microservices

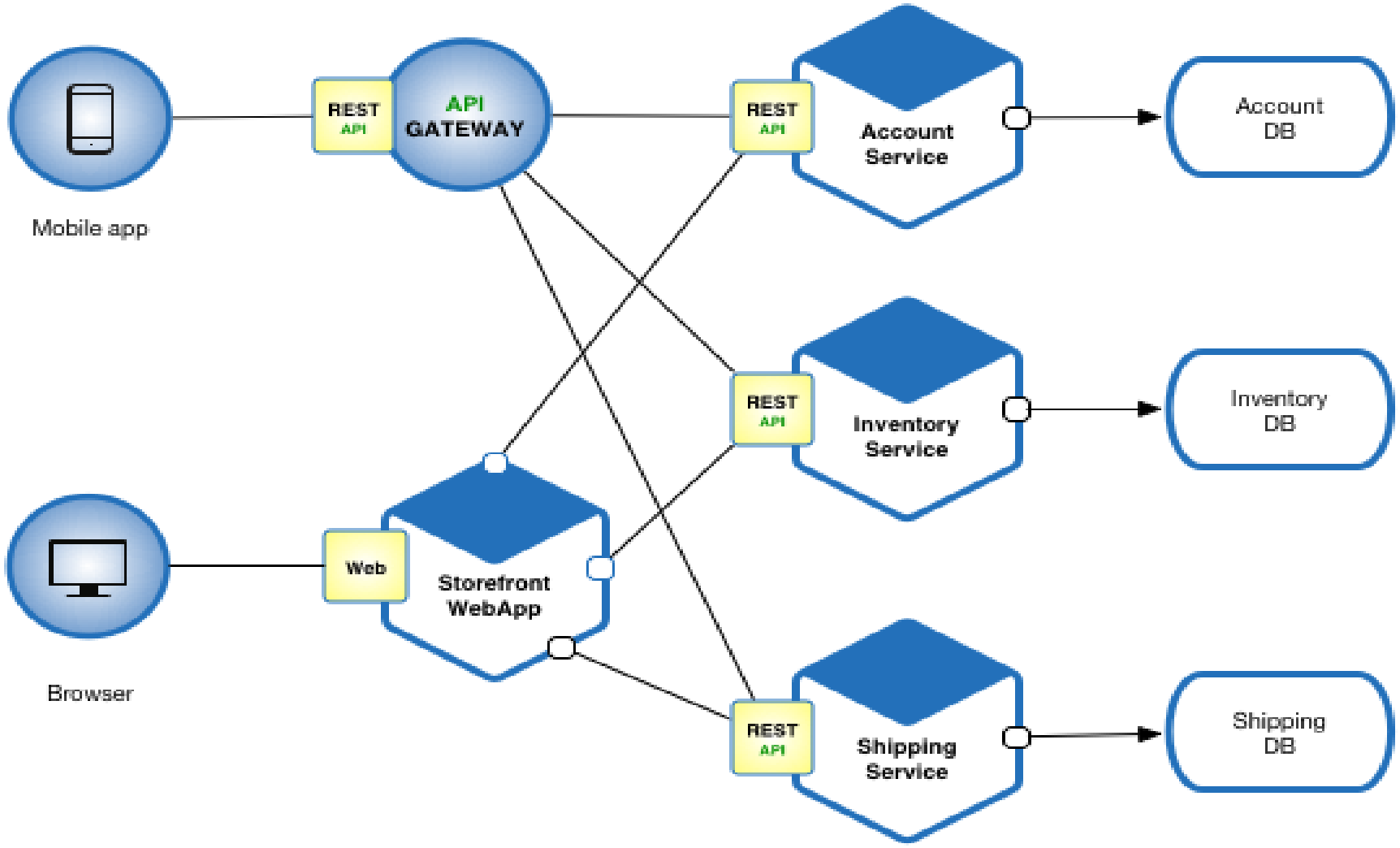
So what exactly is a microservice architecture?

<u>According to Martin Fowler:</u>

A microservice architecture consists of “set of independently deployable services” organized “around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.”

Microservice architecture is a method of developing software applications as a set of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

Microservice Architecture



Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

The microservices style is usually organized around ***business capabilities and priorities.***

Unlike a traditional monolithic development approach—where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic—microservice architecture utilizes cross-functional teams.

The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. That means that when changes are required, there won't necessarily be any reason for the project, as a whole, to take more time or for developers to have to wait for budgetary approval before individual services can be improved.

Microservices enable continuous delivery/deployment

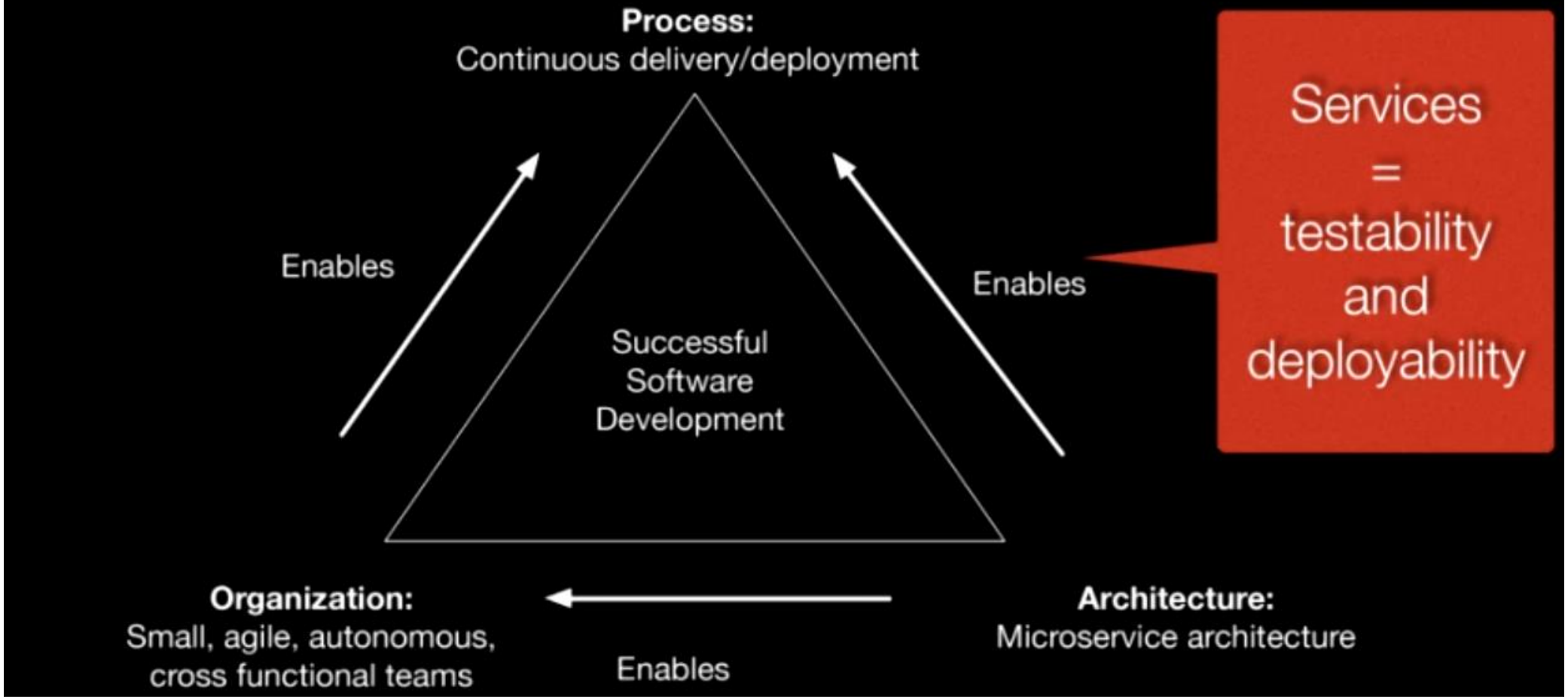
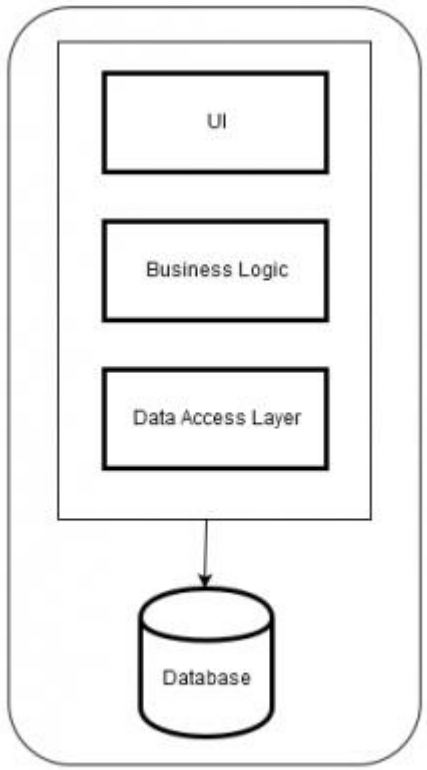
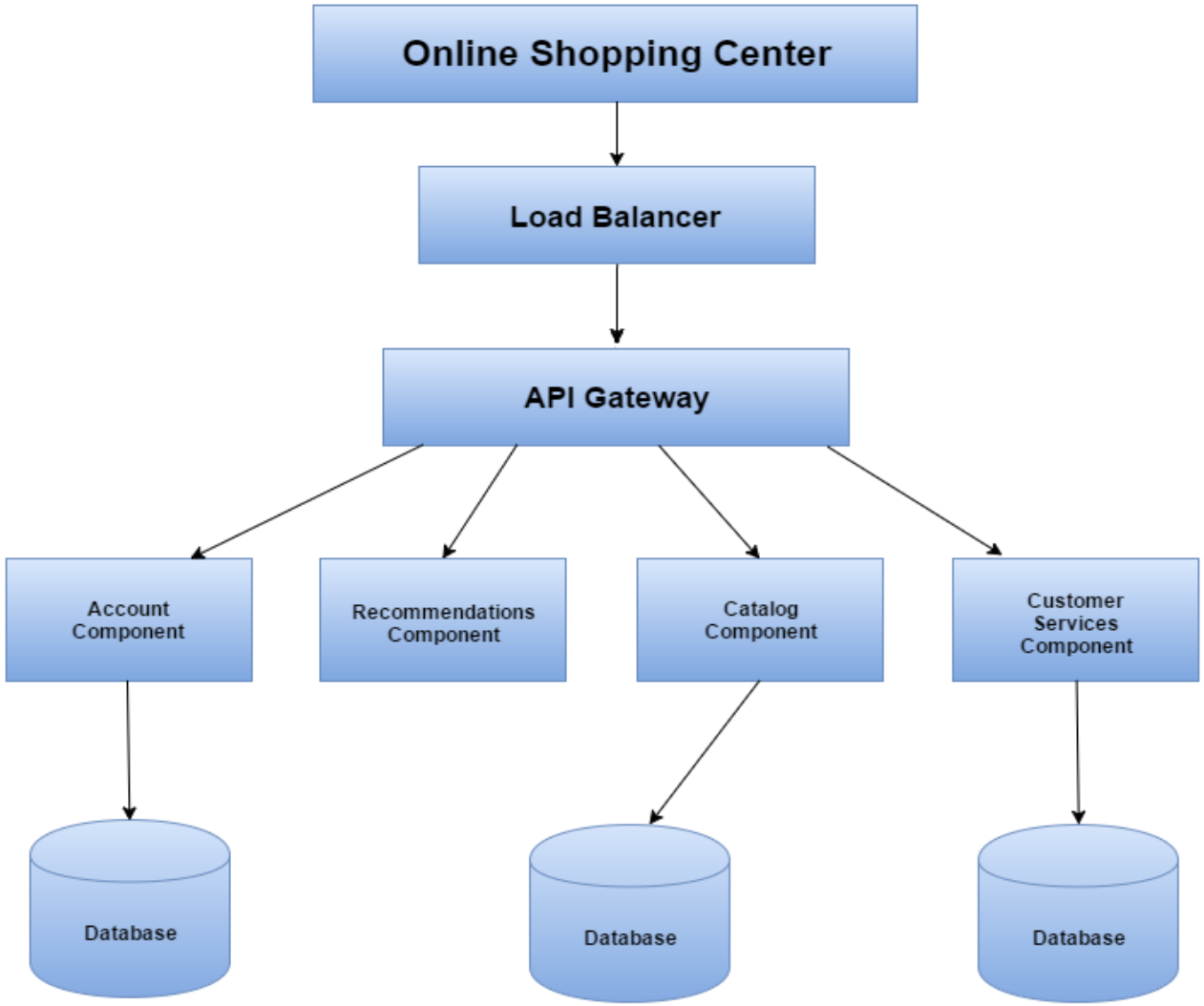
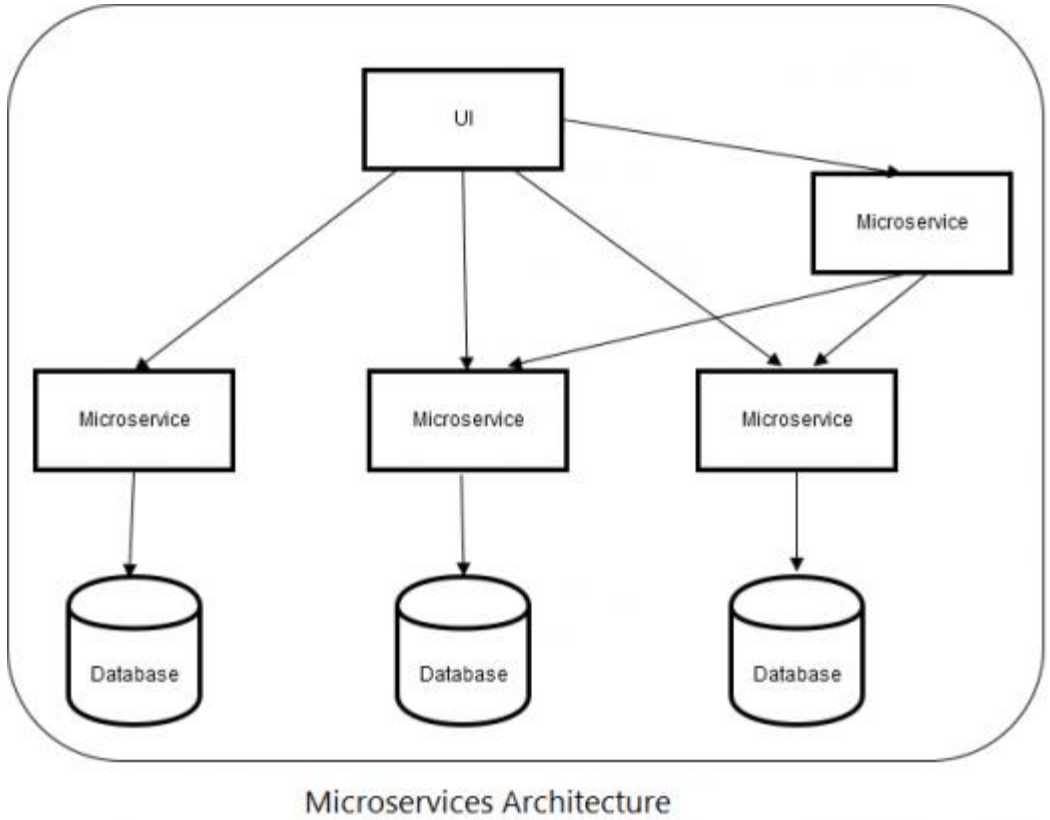


Figure 1: Conventional Approach



Monolithic Architecture

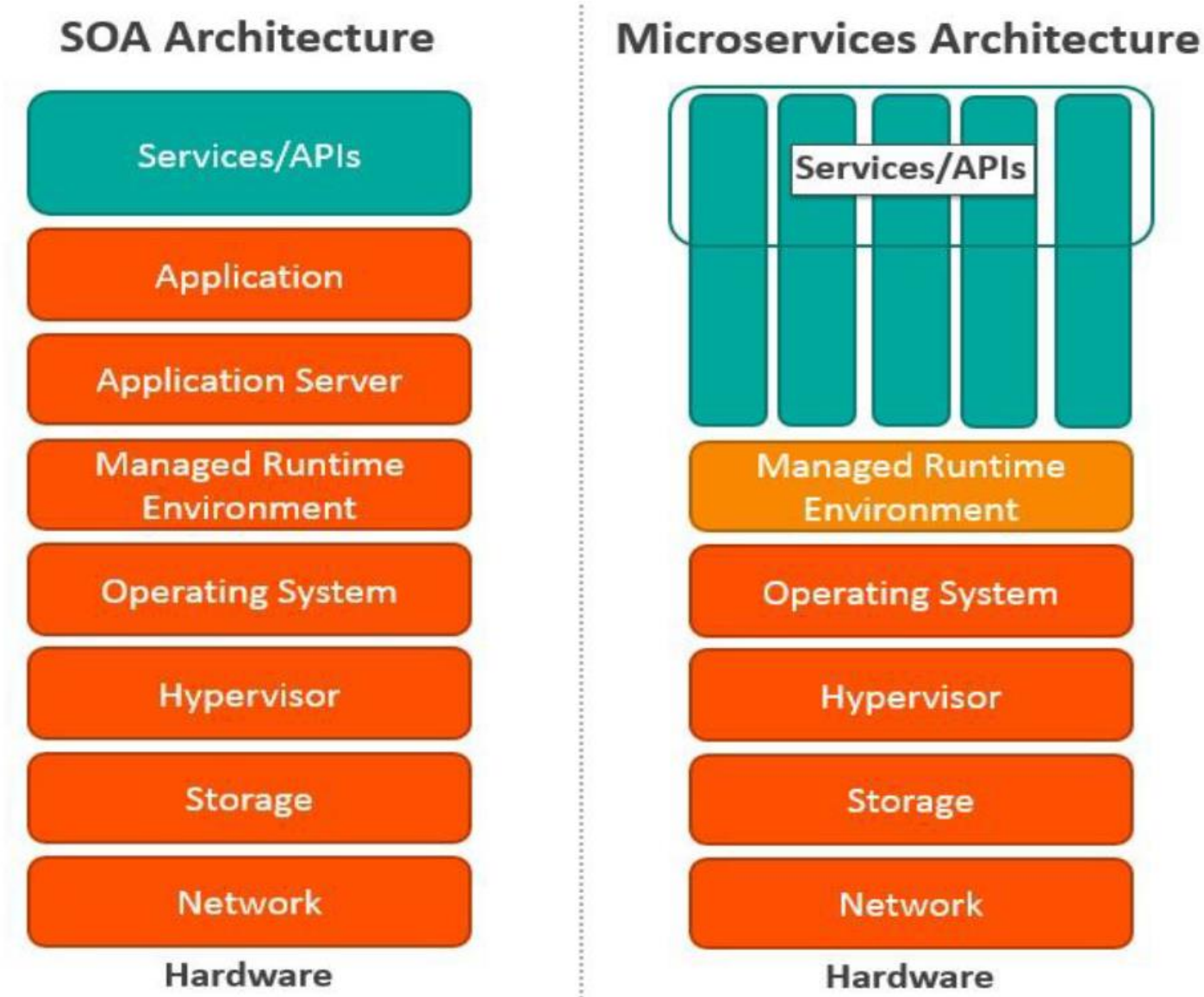




Microservices Architecture

Figure 2: Micro Services Approach

SOA Vs Microservices



SOA Vs Microservices

SOA commonly relies on a shared data model with multiple hierarchies and complex relationships between dozens or hundreds of data structures and services.

It uses a tiered organizational architecture that contains a centralized messaging middleware for service coordination plus additional messaging functionality.

Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

Microservices Benefits

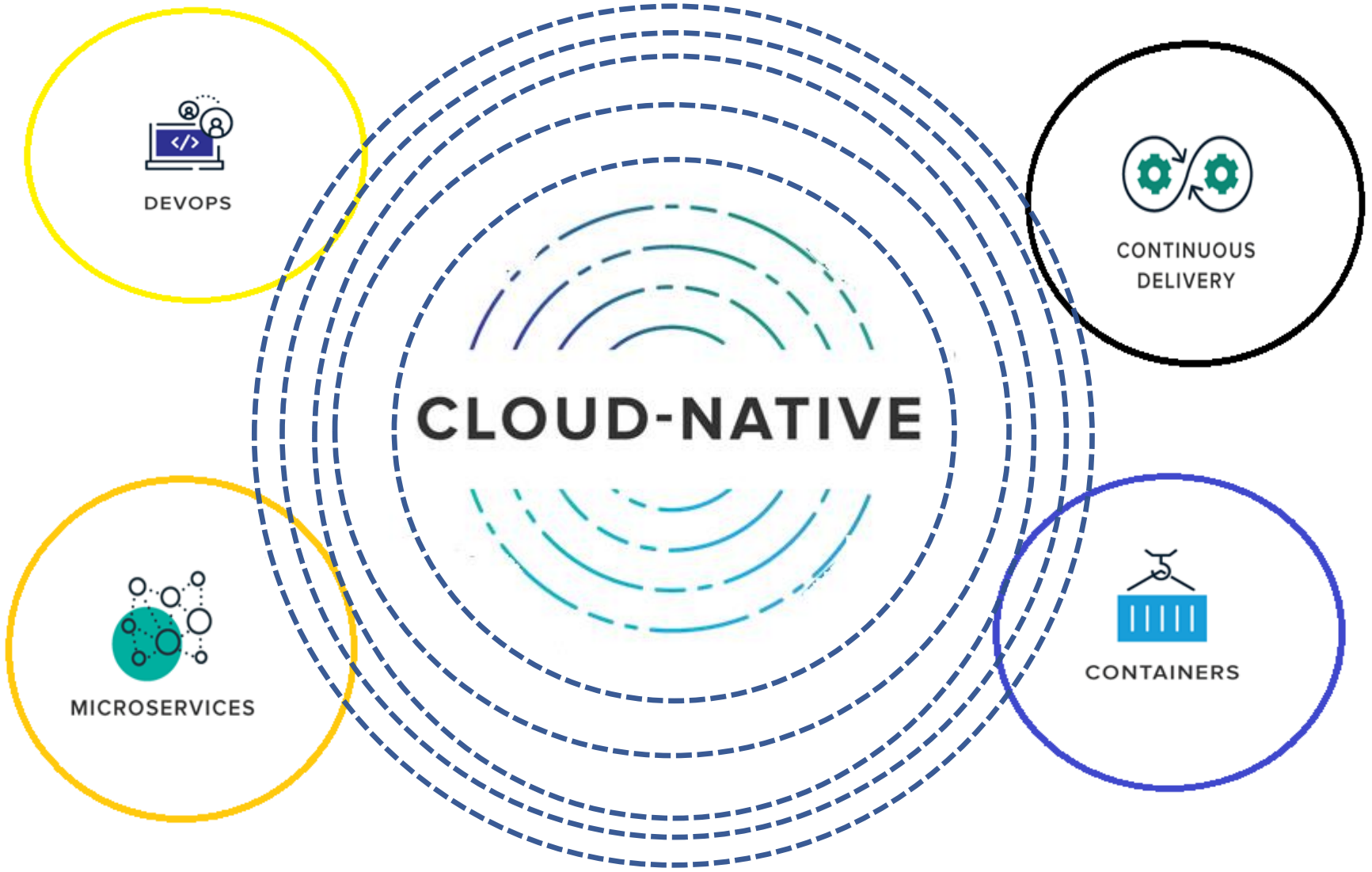
- Smaller code base is easy to maintain.
- Easy to scale as individual component.
- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

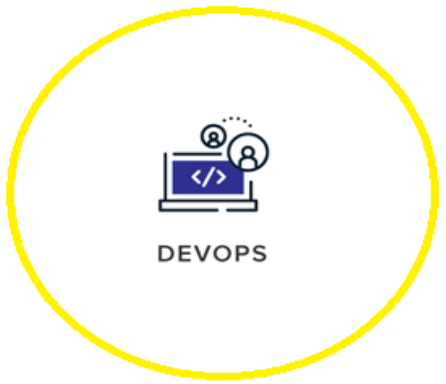
Microservices Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for end to end testing
  (JUnit, Selenium ,Arquillian, Hoverfly, AssertJ)
- Required cultural changes in across teams like Dev and Ops working together even in same team.

Cloud Native Platform

- **Cloud-native** is an approach to building and running applications that exploits the **advantages** of the **cloud computing delivery** model.
- Cloud-native applications conform to a framework or **"contract"** designed to maximize resilience through predictable behaviours.
- Organizations require a platform for building and operating cloud-native applications and services that automates and integrates on **FOUR characteristics**.

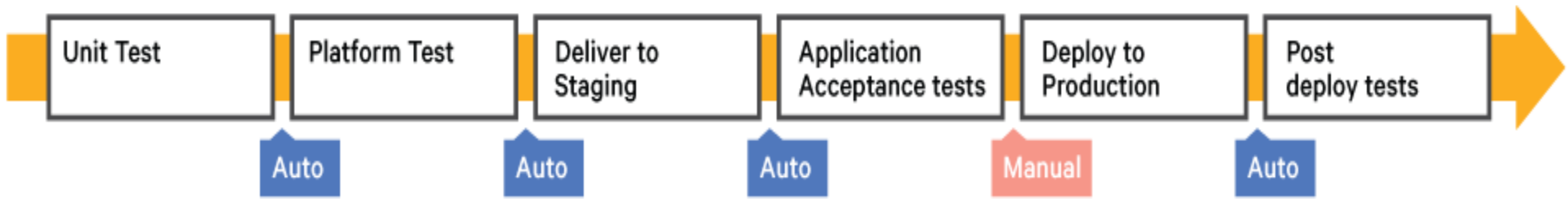

DEVOPS
CONTINUOUS DELIVERY
CLOUD-NATIVE
MICROSERVICES
CONTAINERS



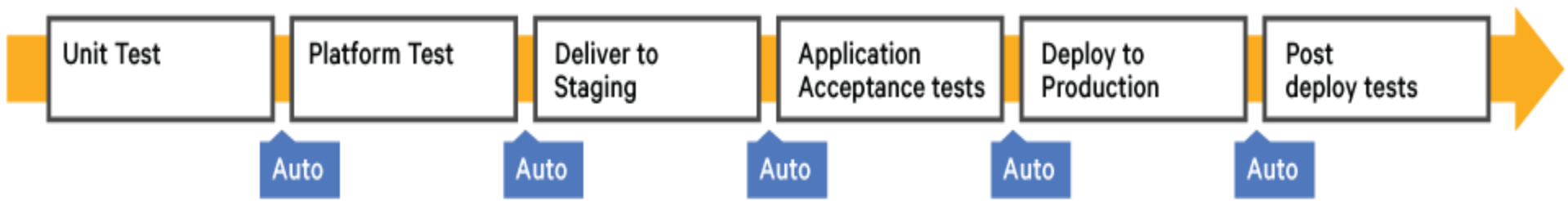
A DevOps culture helps developers and operations work together to deliver shared value to the customer.

Continuous Delivery, enabled by Agile product development practices, is about shipping small batches of software to production constantly, through automation, at less risk, and get feedback faster from end users.

Continuous Delivery



Continuous Deployment



A microservices architecture is a method of developing software applications as a suite of:

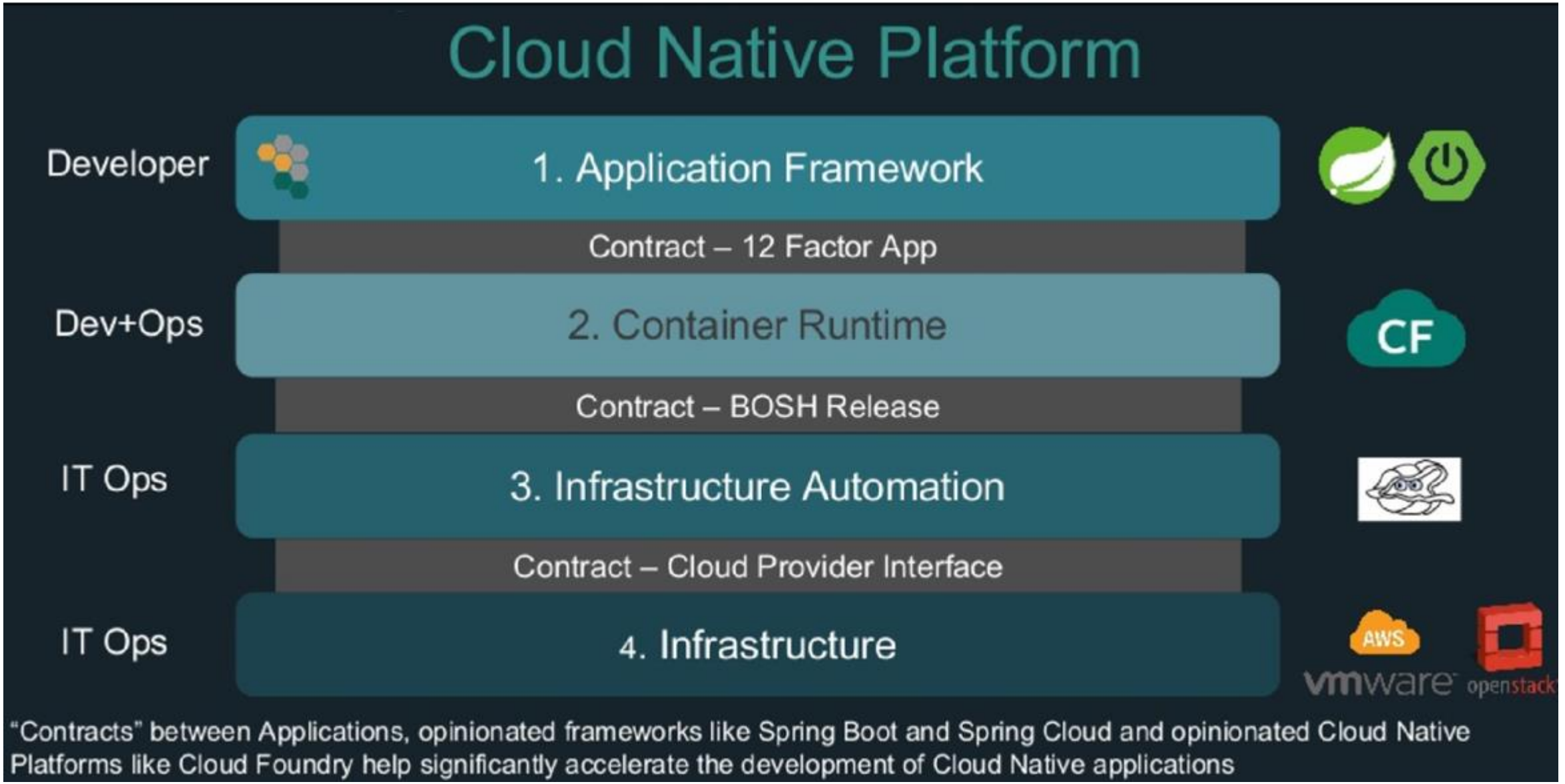
- ❑ independently deployable,
- ❑ small,
- ❑ Each service runs a unique process and
- ❑ communicates through a well-defined, lightweight mechanism to serve a business goal.

Containerization is not a virtualization technology, it's an application delivery technology.

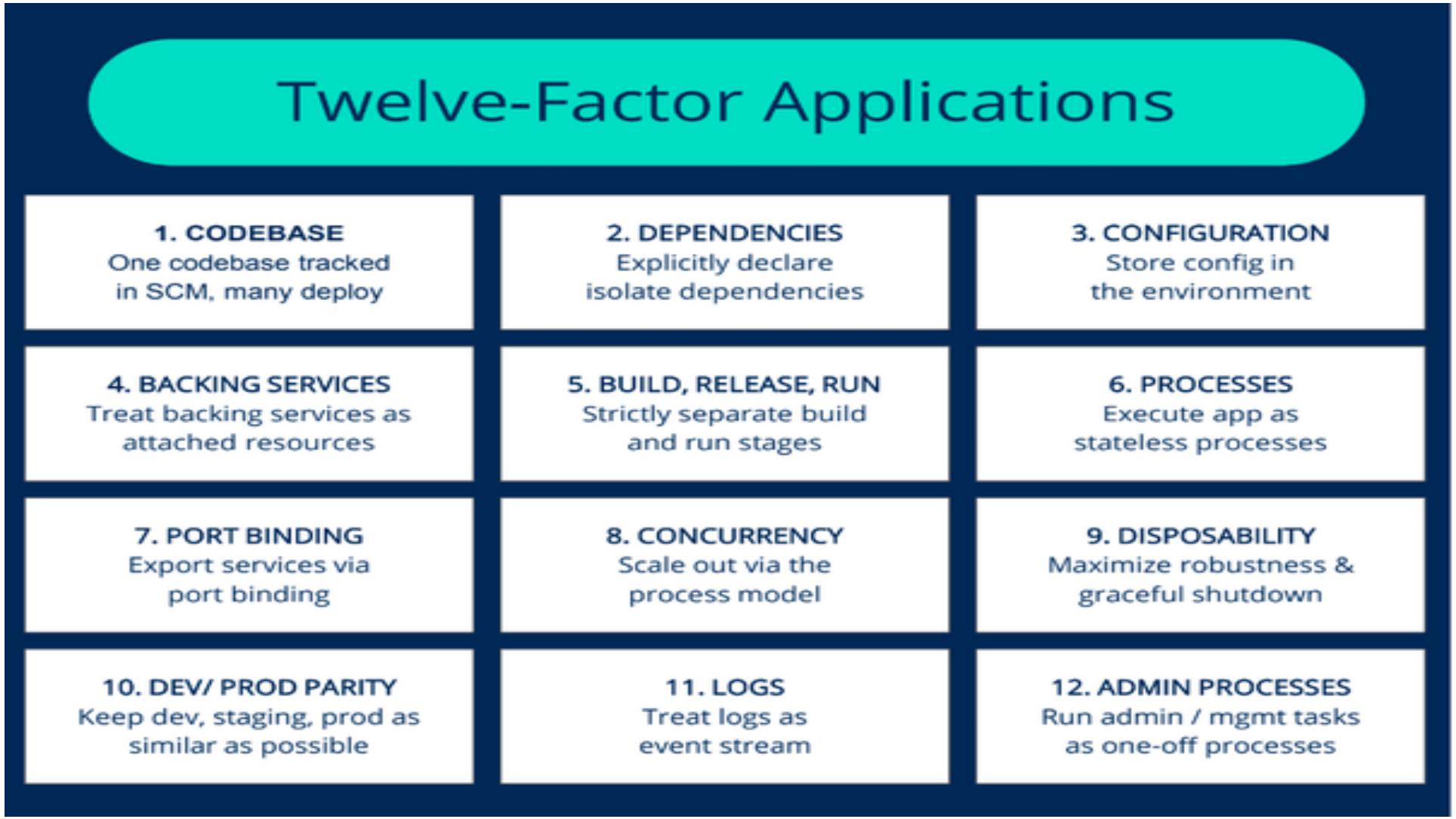
Popular Containers are:

Docker
PCF - Garden
LXD
OpenVZ
Rocket (rkt)

“Cloud-native applications conform to a framework or **“contract”** designed to maximize resilience through predictable behaviours”

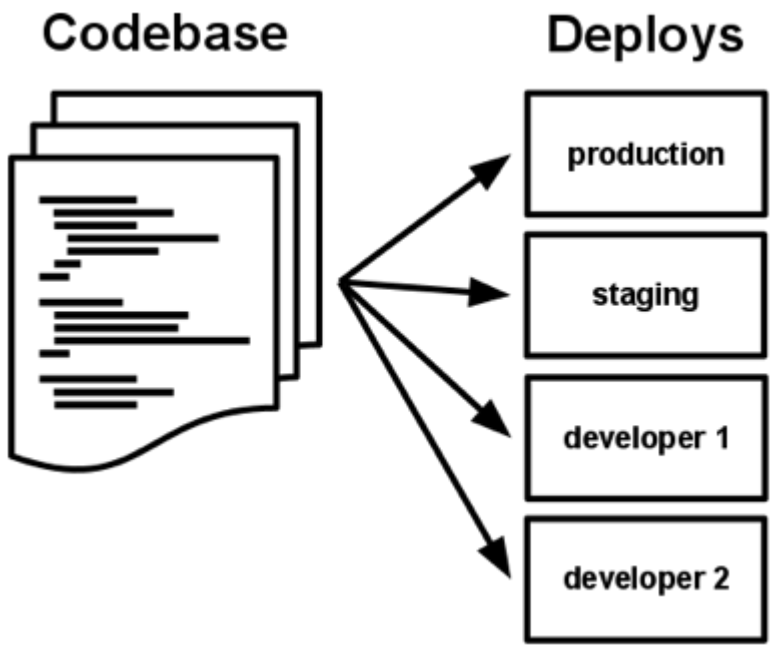

Cloud Native Platform
Developer
1. Application Framework
Contract – 12 Factor App
Dev+Ops
2. Container Runtime
CF
Contract – BOSH Release
IT Ops
3. Infrastructure Automation
Contract – Cloud Provider Interface
IT Ops
4. Infrastructure
AWS
vmware
openstack
"Contracts" between Applications, opinionated frameworks like Spring Boot and Spring Cloud and opinionated Cloud Native Platforms like Cloud Foundry help significantly accelerate the development of Cloud Native applications

The twelve-factor app is a methodology for building software-as-a-service apps



1. CODEBASE

One codebase tracked in SCM, many deploy



GitHub

2. DEPENDENCIES
Explicitly declare
isolate dependencies

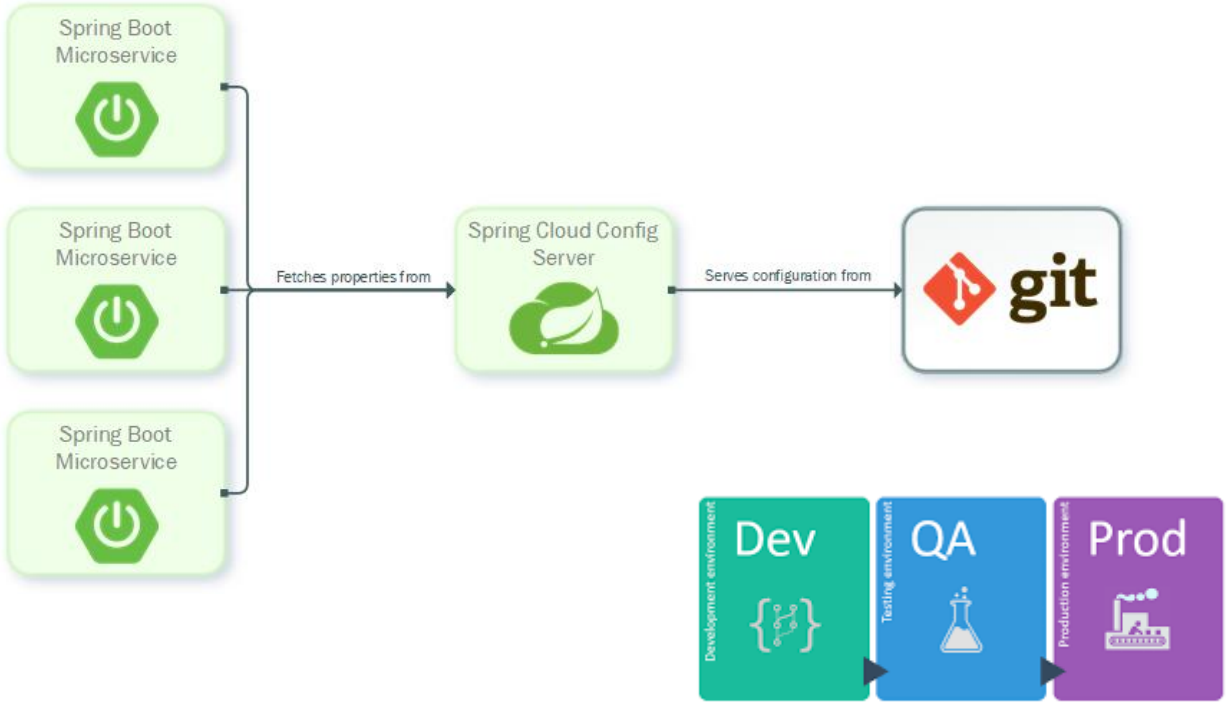
pom.xml

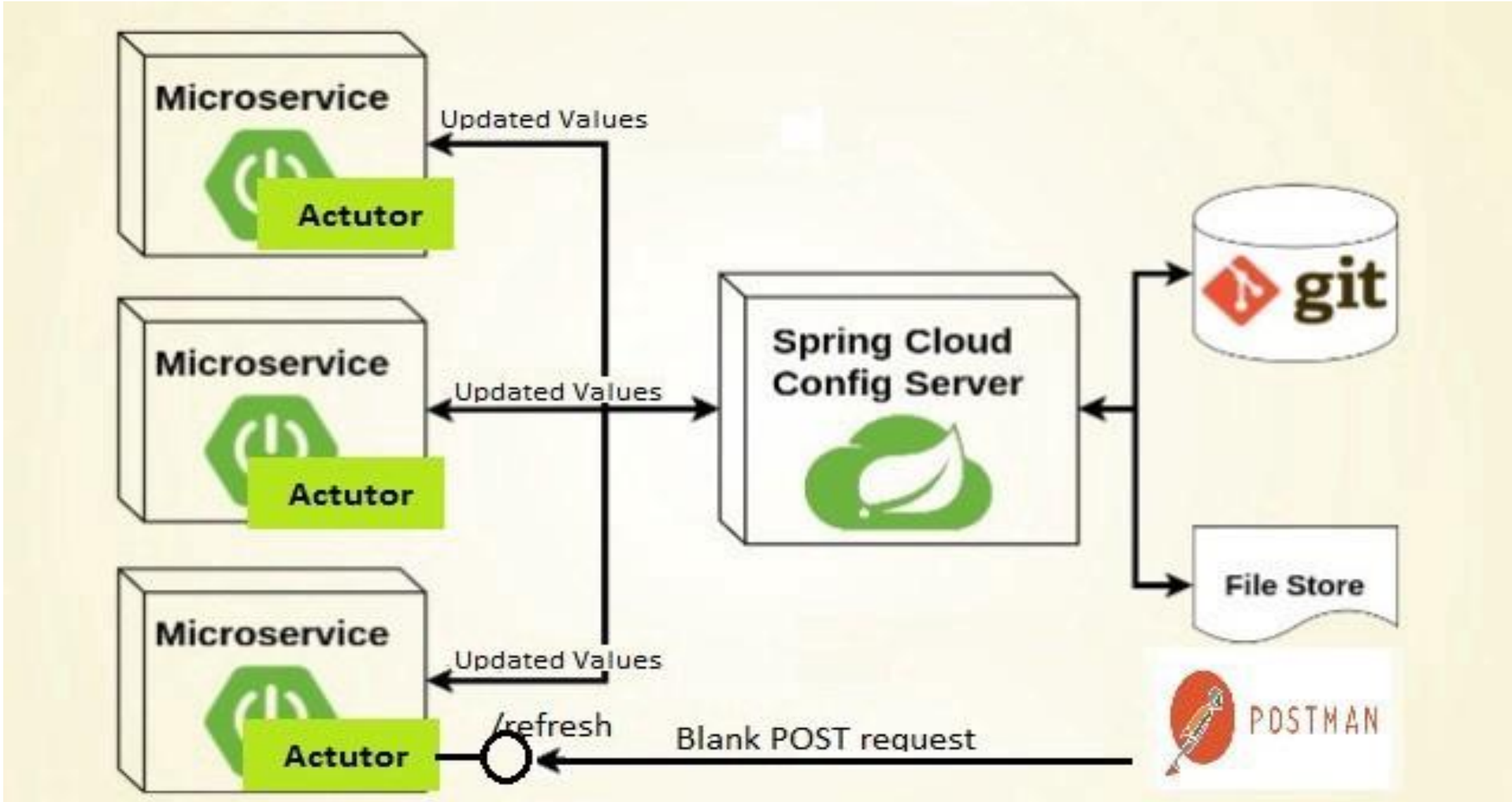
Dockerfile

manifest.yml

3. CONFIGURATION

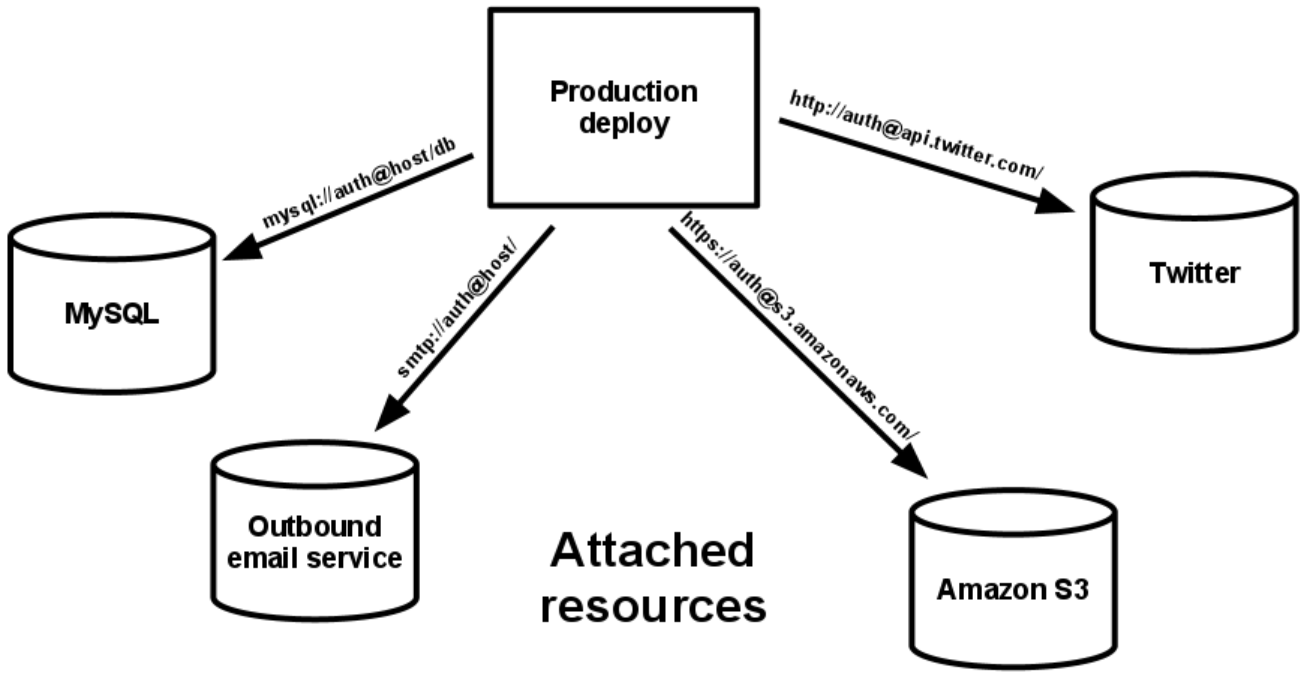
Store config in the environment




Microservice
Actutor
Microservice
Actutor
Microservice
Actutor
Updated Values
Updated Values
Updated Values
/refresh
Blank POST request
Spring Cloud
Config Server
git
File Store
POSTMAN

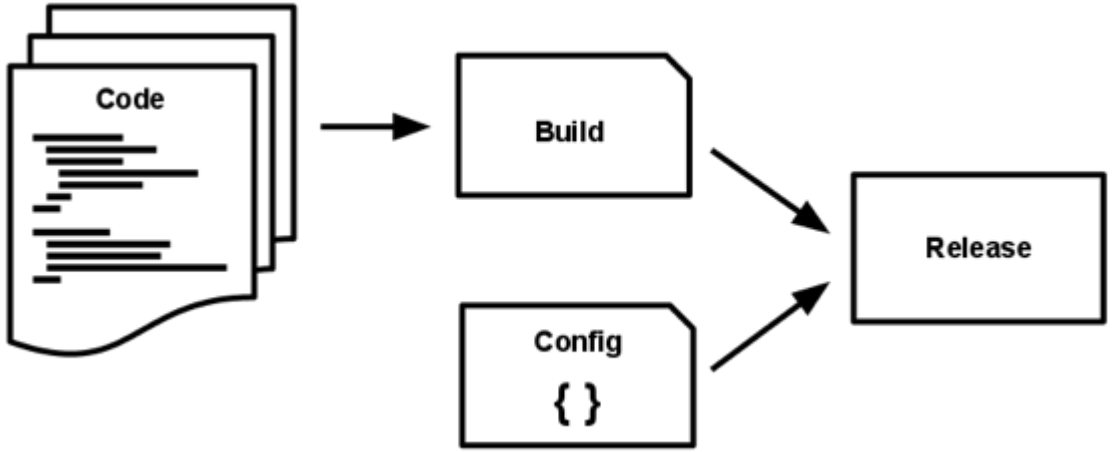
4. BACKING SERVICES

Treat backing services as attached resources



5. BUILD, RELEASE, RUN

Strictly separate build and run stages

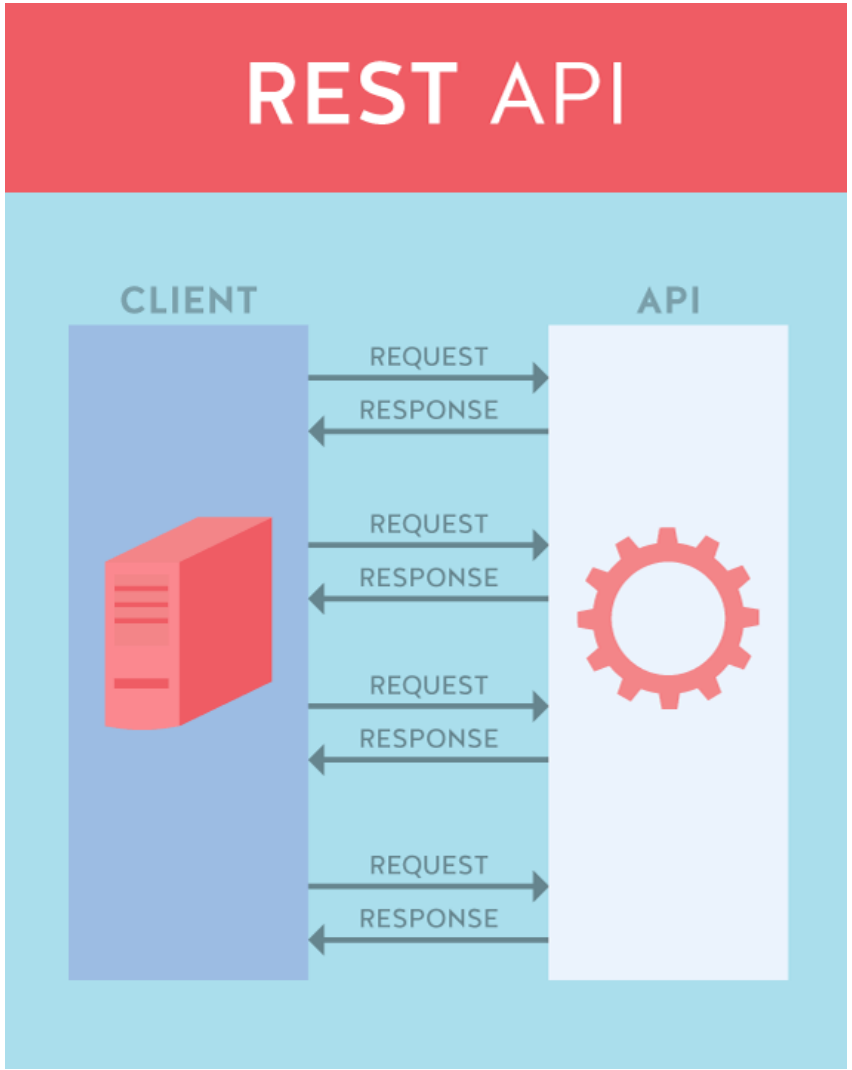


Jenkins

concourse

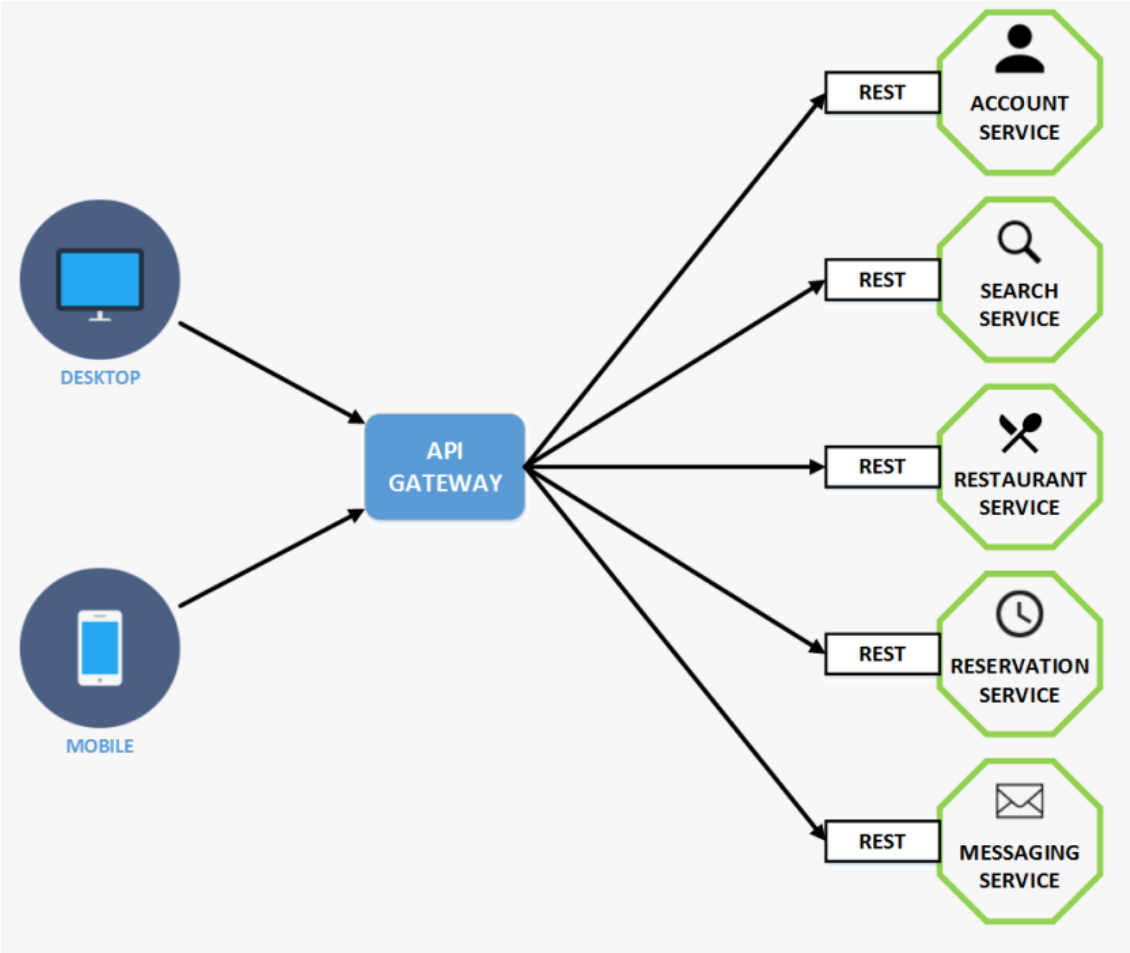
6. PROCESSES

Execute app as
stateless processes

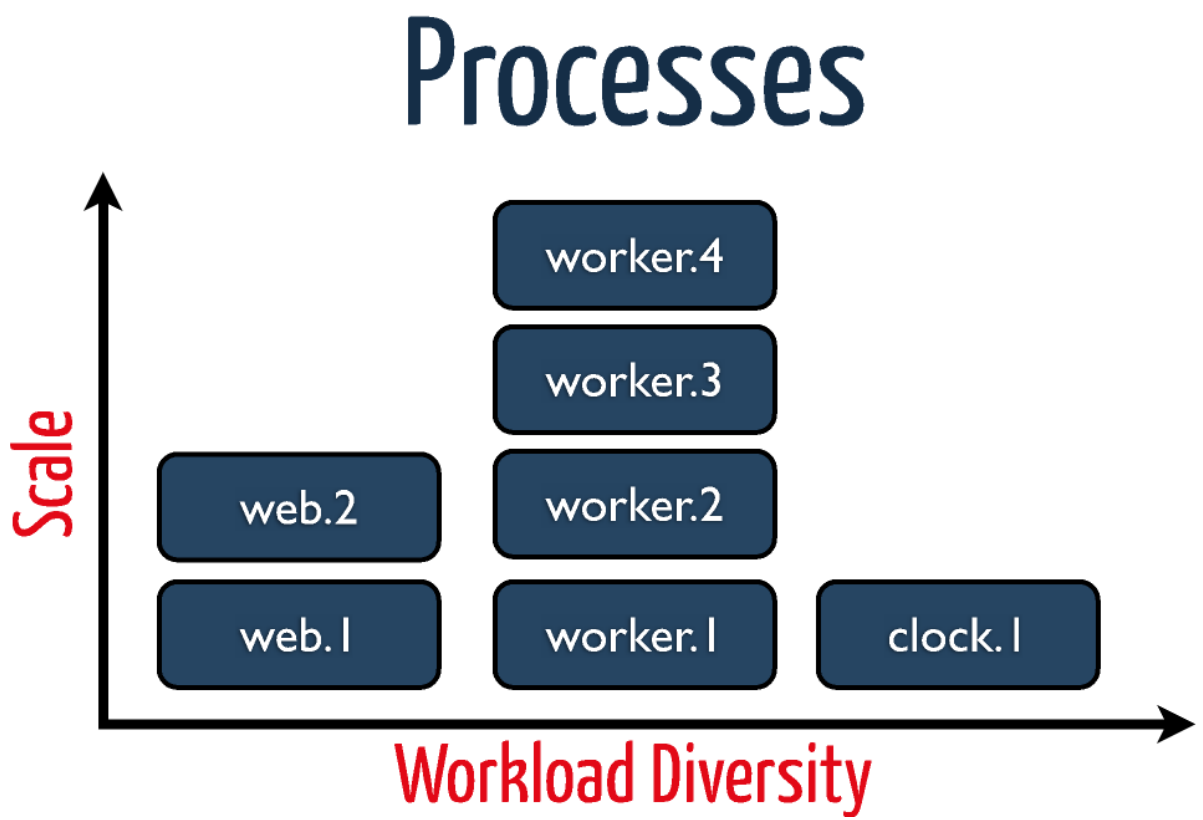


7. PORT BINDING

Export services via port binding

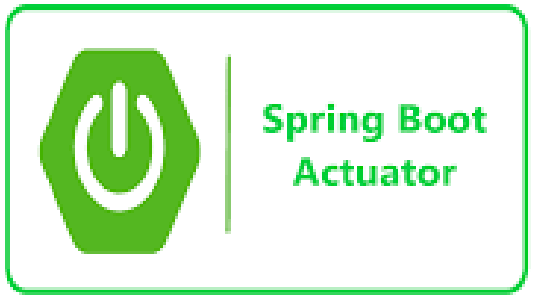


8. CONCURRENCY
Scale out via the process model



9. DISPOSABILITY

Maximize robustness
with fast startup
and graceful shutdown

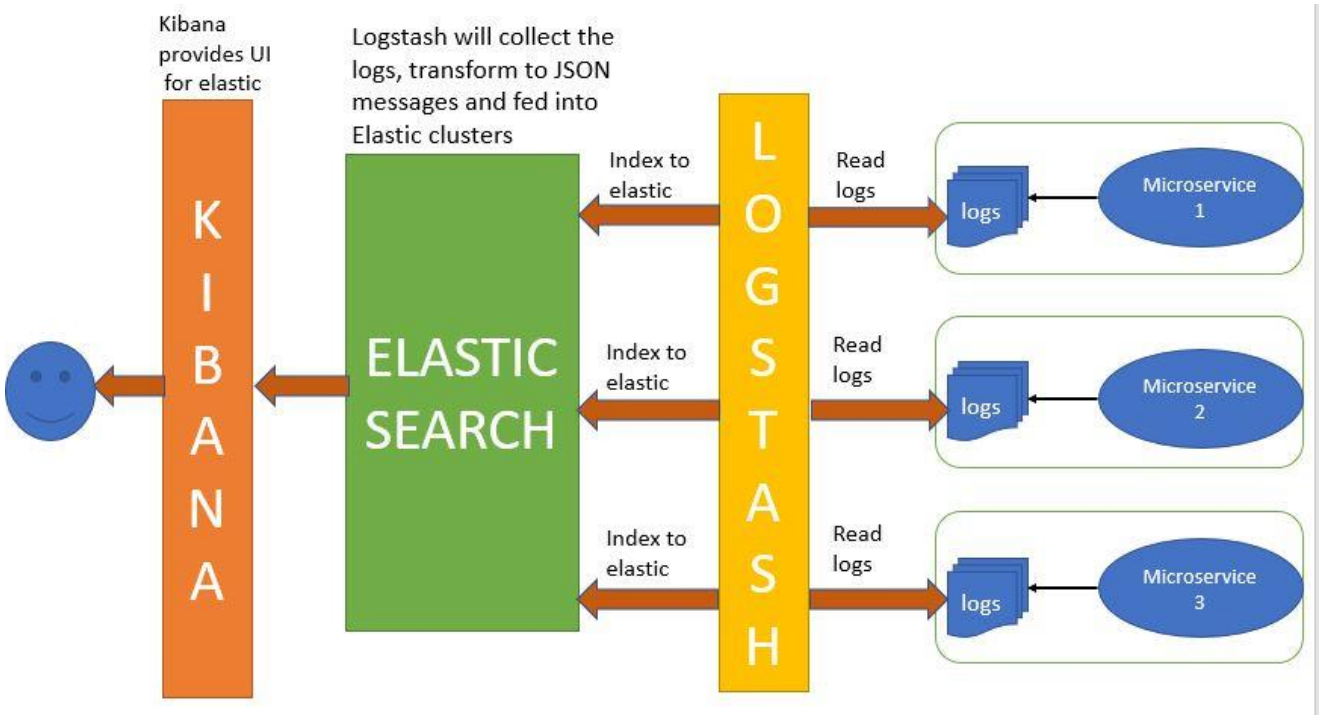


10. DEV/ PROD PARITY

Keep dev, staging, prod as similar as possible

11. LOGS

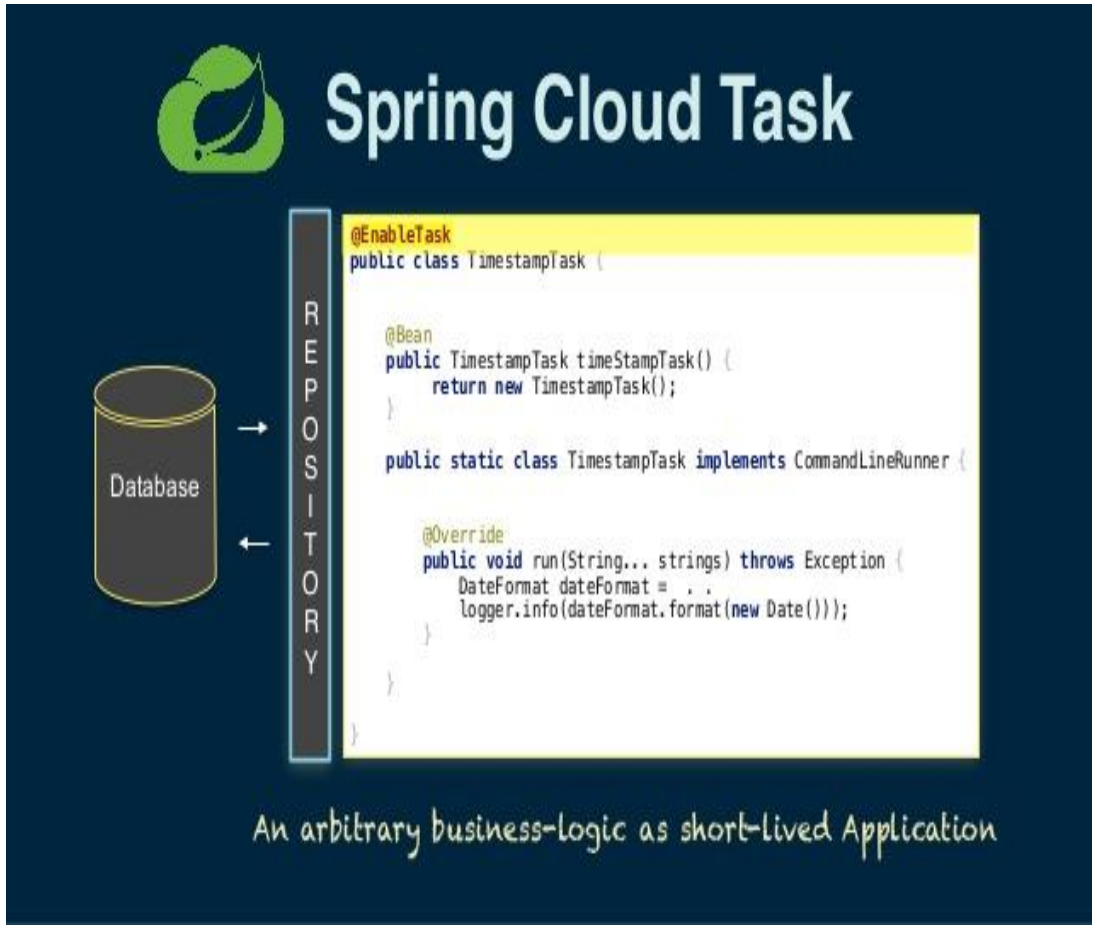
Treat logs as event stream

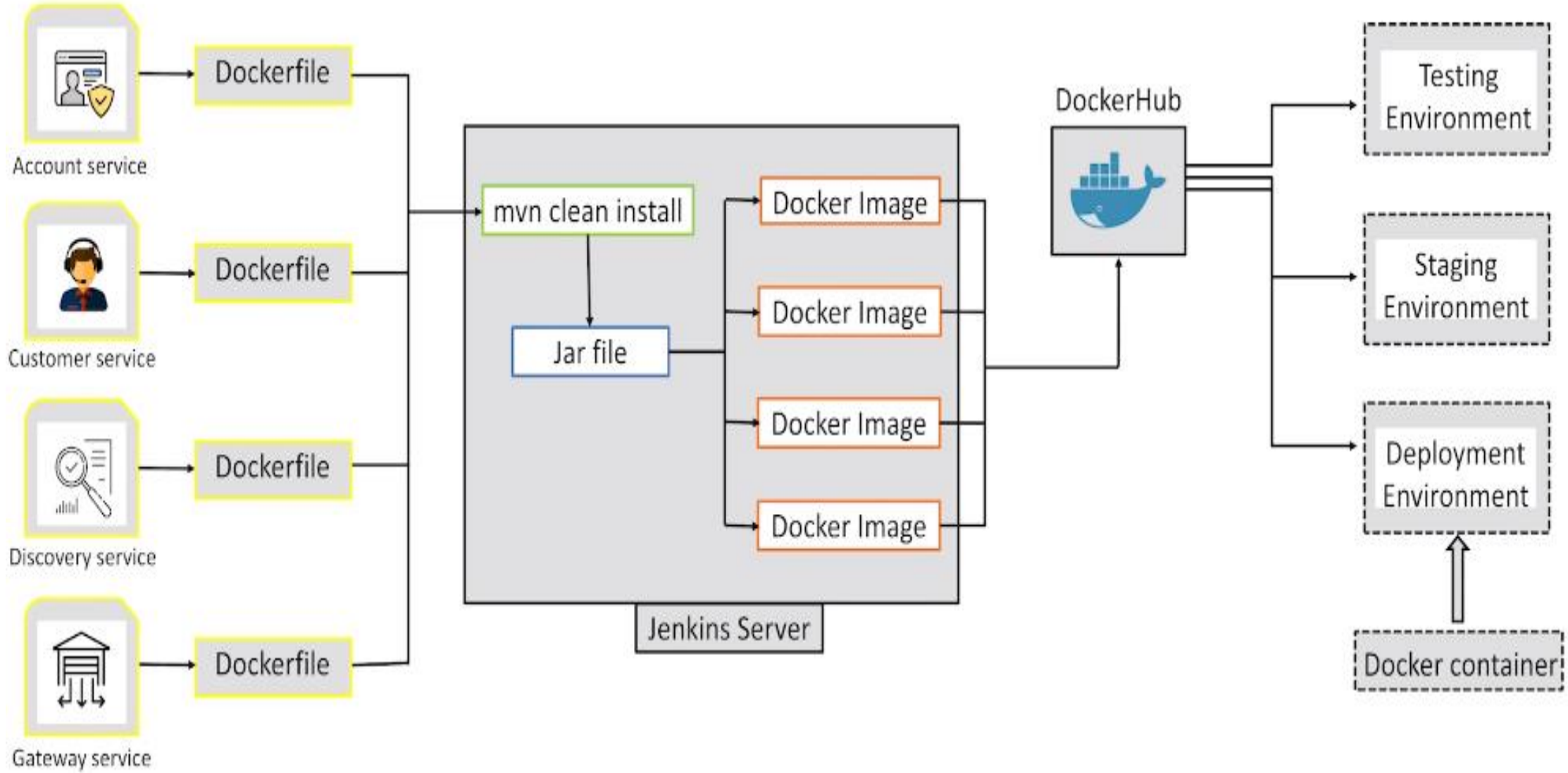


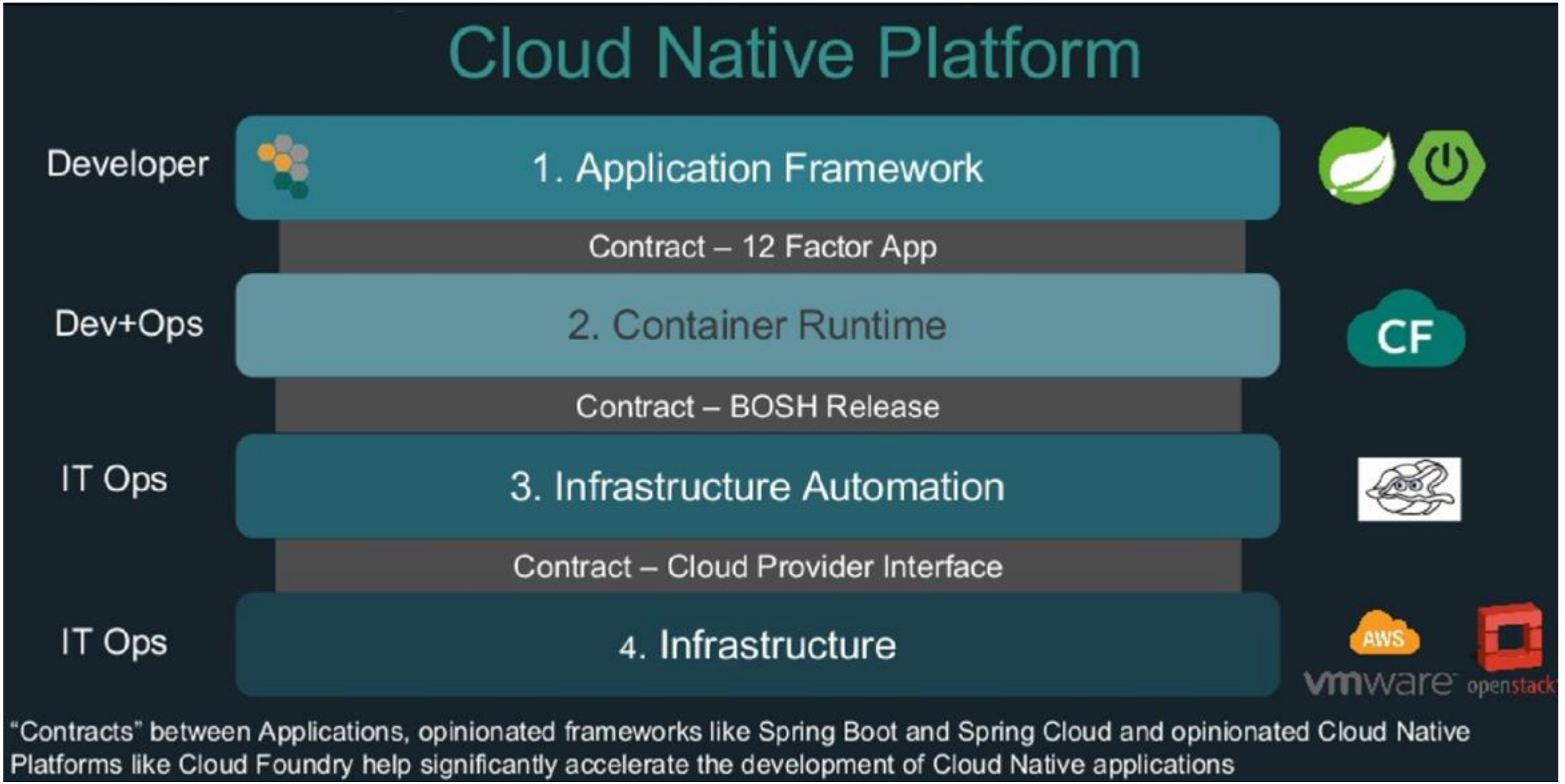
12. ADMIN PROCESSES

Run admin / mgmt tasks
as one-off processes

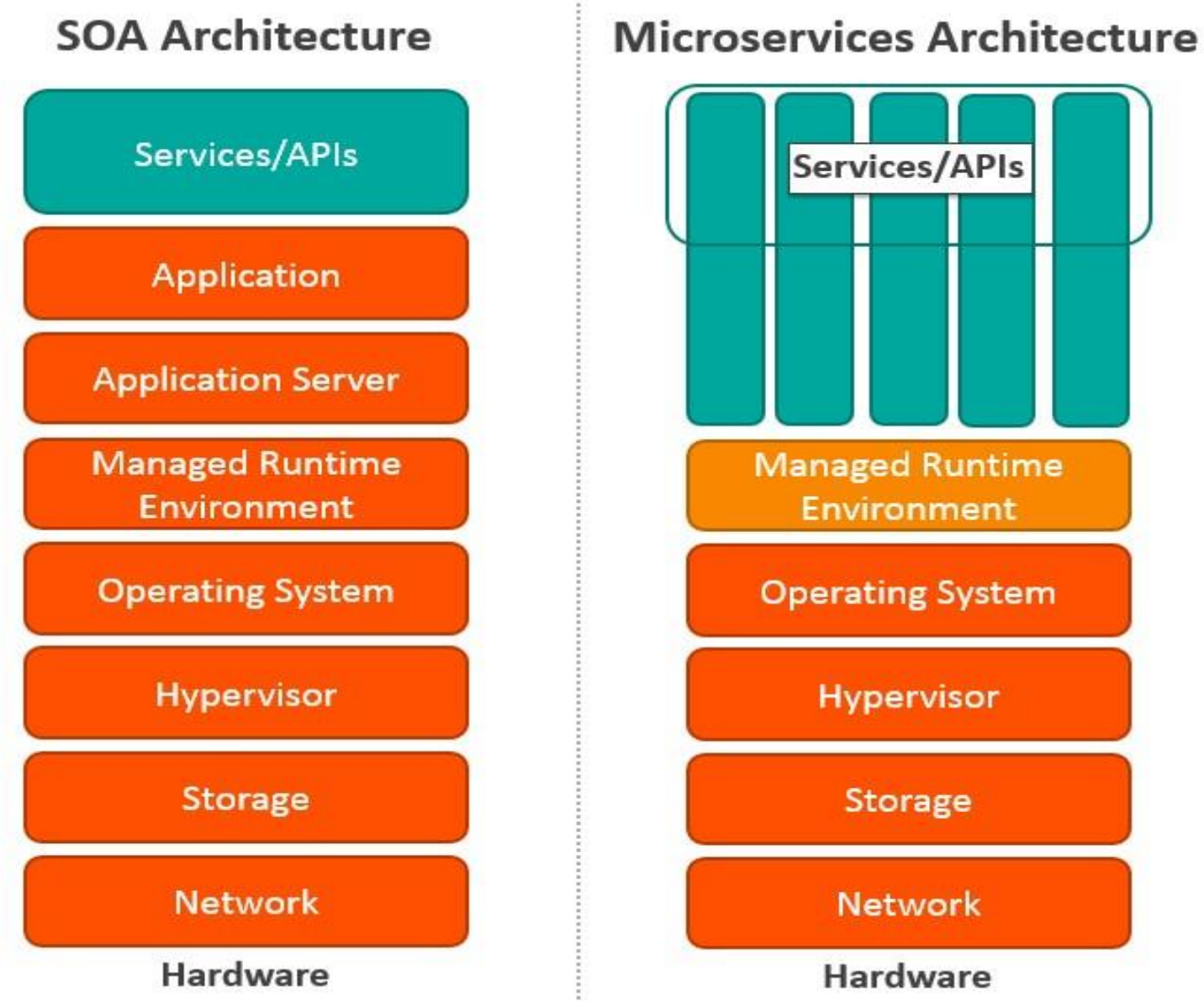
- ❑ Running database migrations
- ❑ Fetch analytical data to gather business insights
- ❑ Batch processing tasks




Account service
Dockerfile
Customer service
Dockerfile
Discovery service
Dockerfile
Gateway service
Dockerfile
mvn clean install
Jar file
Docker Image
Docker Image
Docker Image
Docker Image
Jenkins Server
DockerHub
Testing Environment
Staging Environment
Deployment Environment
Docker container


Cloud Native Platform
Developer
1. Application Framework
Contract – 12 Factor App
Dev+Ops
2. Container Runtime
CF
Contract – BOSH Release
IT Ops
3. Infrastructure Automation
Contract – Cloud Provider Interface
IT Ops
4. Infrastructure
AWS
vmware
openstack
"Contracts" between Applications, opinionated frameworks like Spring Boot and Spring Cloud and opinionated Cloud Native Platforms like Cloud Foundry help significantly accelerate the development of Cloud Native applications

SOA Vs Microservices



SOA Vs Microservices

SOA commonly relies on a shared data model with multiple hierarchies and complex relationships between dozens or hundreds of data structures and services.

It uses a tiered organizational architecture that contains a centralized messaging middleware for service coordination plus additional messaging functionality.

Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

Microservices Infrastructure

- Platform as a Service like Pivotal Cloud Foundry help to deployment, easily run, scale, monitor etc.

- It support for continuous deployment, rolling upgrades fo new versions of code, running multiple versions of same service at same time

Virtualization ,Cloud Computing & Docker

- **Virtualization** is generally accomplished by dividing a single piece of hardware into two or more ‘segments.’

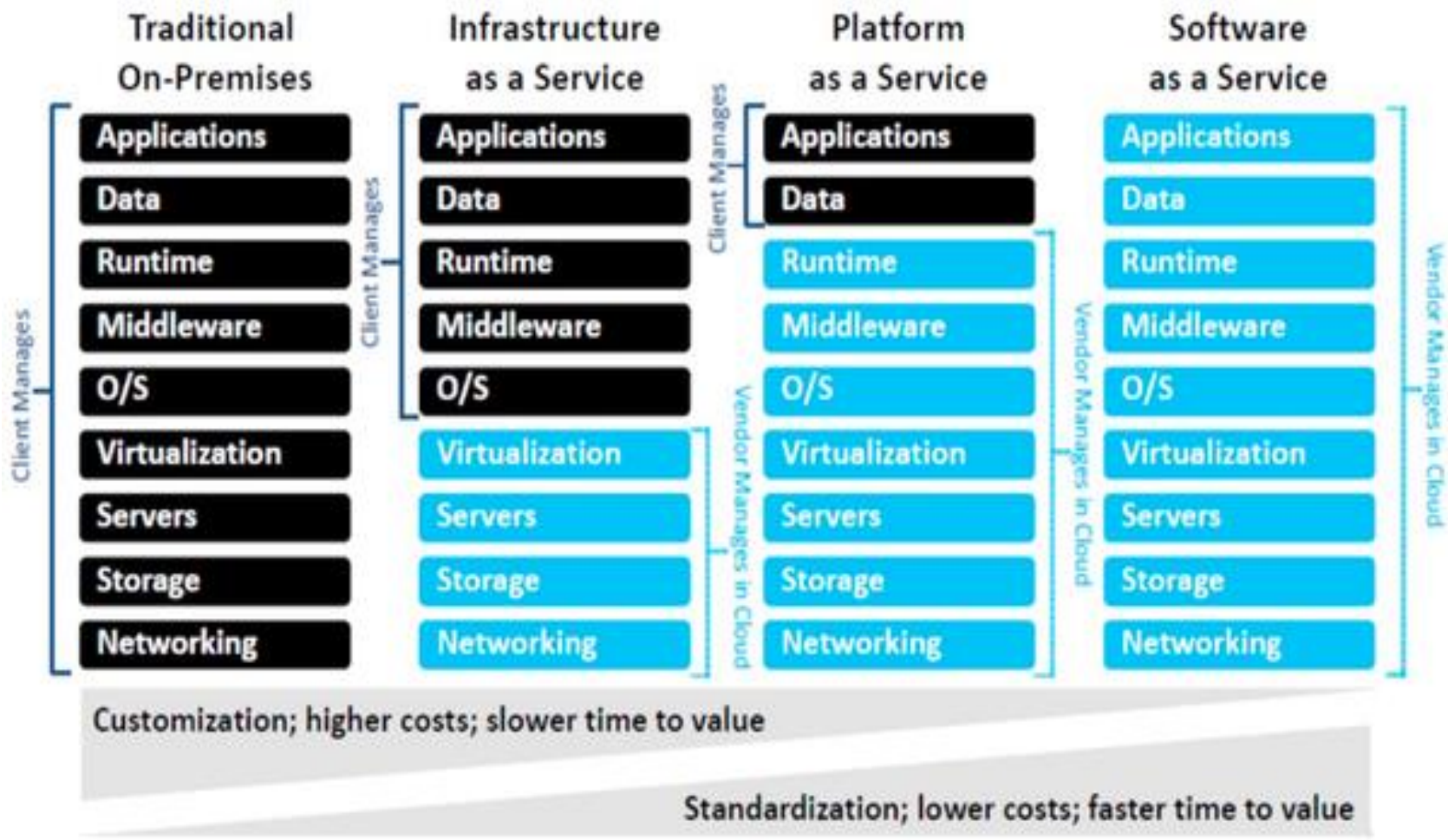
- Each segment operates as its own independent environment.

- For example, server virtualization partitions a single server into a number of smaller virtual servers.

- Essentially, virtualization serves to make computing environments independent of physical infrastructure.

CLOUD COMPUTING

It is shared computing resources, software, or data are delivered as a service and on-demand through the Internet.


Traditional On-Premises
Infrastructure as a Service
Platform as a Service
Software as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Client Manages
Client Manages
Client Manages
Vendor Manages in Cloud
Vendor Manages in Cloud
Vendor Manages in Cloud
Customization; higher costs; slower time to value
Standardization; lower costs; faster time to value

Virtualization vs Cloud Computing

- Virtualization is a technology
- Cloud computing is a service
- Virtualization can exist without the cloud, but cloud computing cannot exist without virtualization

Docker vs VM

VMs are hardware-centric and docker containers are application-centric

A **virtual machine** (**VM**) is a software implementation of a **machine** (i.e. a computer) that executes programs like a physical **machine**.

An **image** of a **virtual machine** is a copy of the **VM**, which may contain an OS, data files, and applications (just like your personal computer).

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

Docker is a bit like a virtual machine.

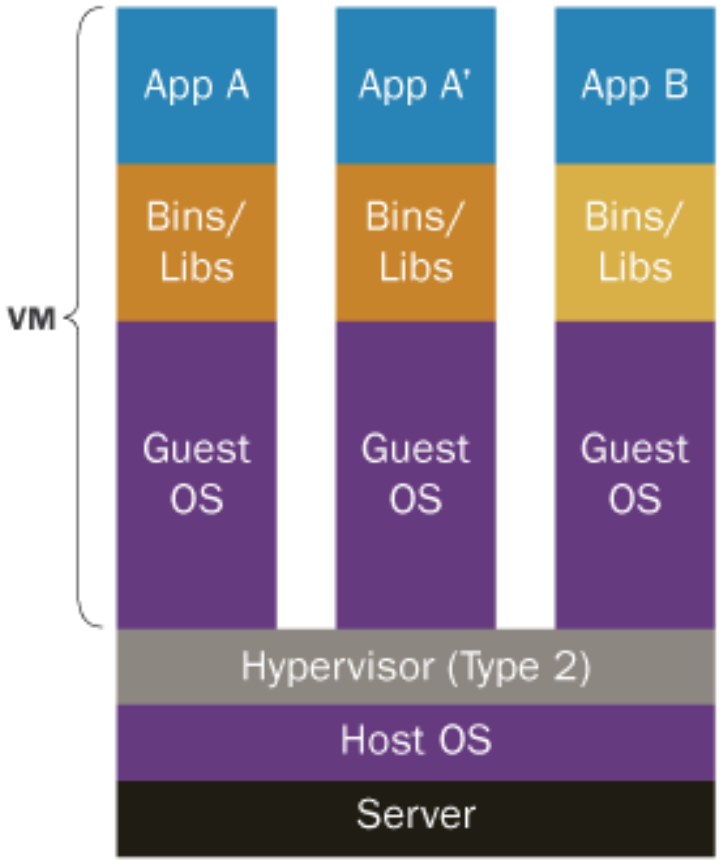
But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer.

This gives a significant performance boost and reduces the size of the application.

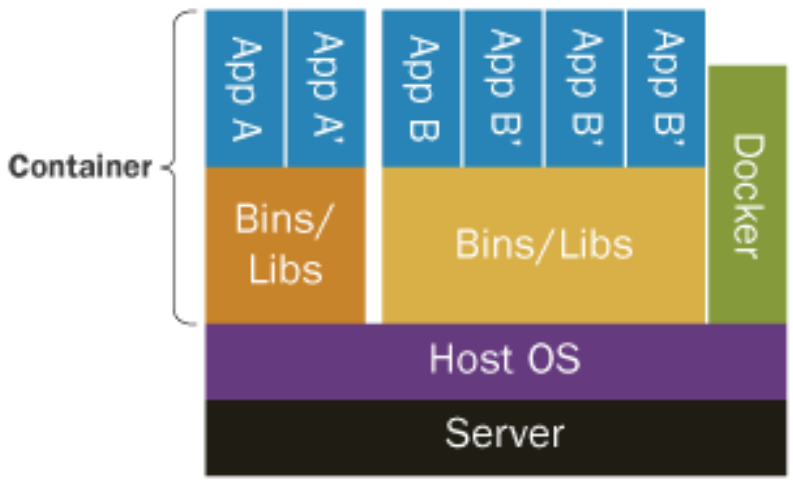
Docker is not a virtualization technology, it’s an application delivery technology

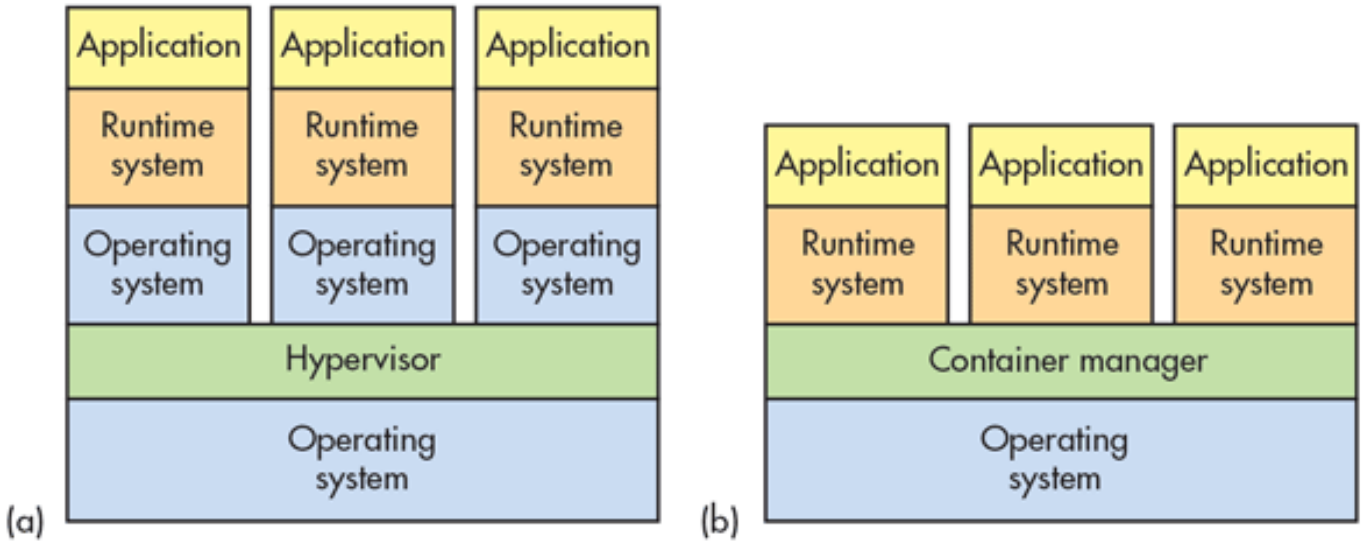
PCF uses Garden container. Docker is another option

Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries

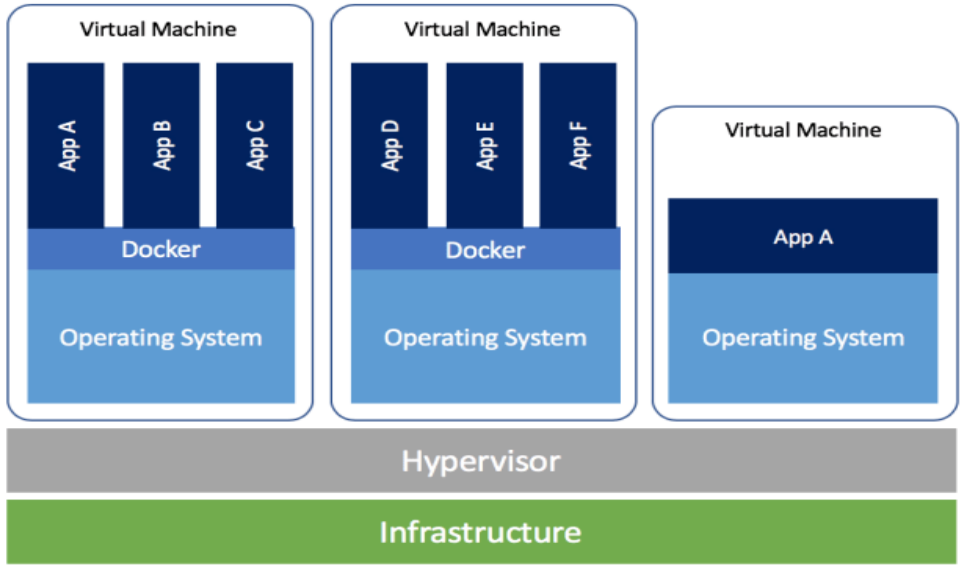




Virtual machines (VM) are managed by a hypervisor and utilize VM hardware

Container systems provide operating system services from the underlying host and isolate the applications using virtual-memory hardware.

Integrate Containers with Your Existing IT Processes



Containerized applications share common operating system and software libraries which greatly improves CPU utilization within a VM. This means an organization can reduce the overall number of virtual machines needed to operate their environment and increase the number of applications that can run on a server.

Docker primarily focuses on automating the deployment of applications inside application containers.

Application containers are designed to package and run a single service.

System containers are designed to run multiple processes, like virtual machines.

Design Patterns

RESTful API Patterns

- ❑ Statelessness
- ❑ Content Negotiation
- ❑ URI Templates
- ❑ Pagination
- ❑ Versioning
- ❑ Authorization
- ❑ API facade
- ❑ Discoverability
- ❑ Idempotent
- ❑ Circuit breaker

domain-driven design patterns

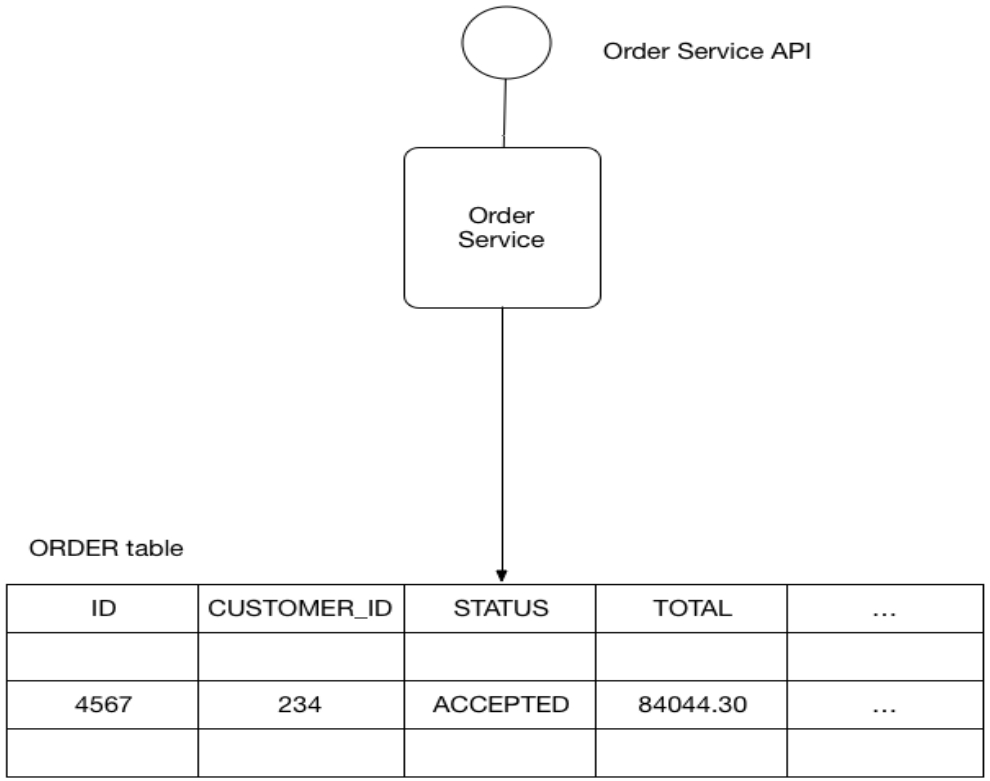
Microservice Patterns:

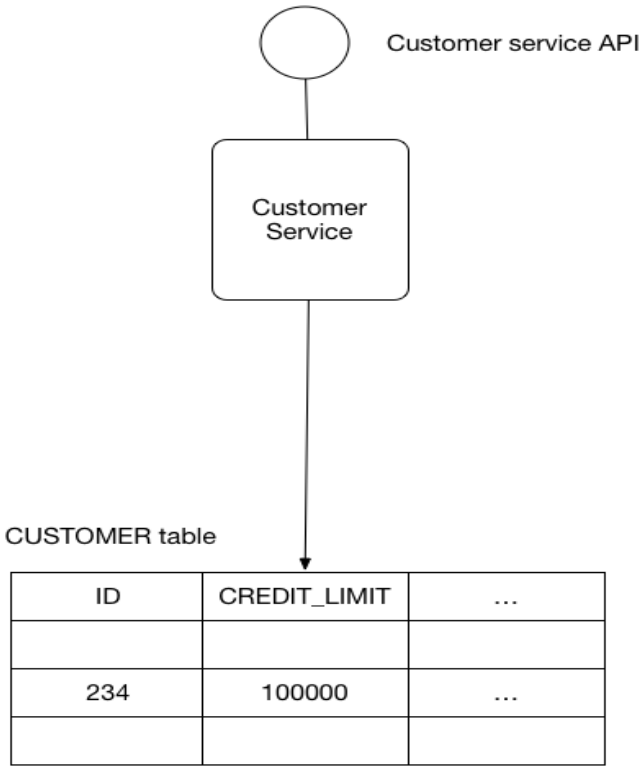
- ❑ API gateway
- ❑ Service registry
- ❑ Circuit breaker
- ❑ Messaging
- ❑ Database per Service
- ❑ Access Token
- ❑ Saga
- ❑ Event Sourcing & CQRS

Problem

What’s the database architecture in a microservices application?

Pattern: Shared database





Pattern: Shared database

The benefits of this pattern are:

A developer uses familiar and straightforward ACID transactions to enforce data consistency

A single database is simpler to operate

Pattern: Shared database

The drawbacks of this pattern are:

Development time coupling - a developer working on, for example, the OrderService will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.

Runtime coupling - because all services access the same database they can potentially interfere with one another. For example, if long running CustomerService transaction holds a lock on the ORDER table then the OrderService will be blocked.

Single database might not satisfy the data storage and access requirements of all services.

Pattern: Database per service

Resulting context
Using a database per service has the following benefits:

Helps ensure that the services are loosely coupled.

Changes to one service’s database does not impact any other services.

Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo

Using a database per service has the following drawbacks:

Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided. Moreover, many modern (NoSQL) databases don’t support them.

Implementing queries that join data that is now in multiple databases is challenging.

Complexity of managing multiple SQL and NoSQL databases

There are various patterns/solutions for implementing transactions and queries that span services:

Implementing transactions that span services - use the **Saga** pattern.

Implementing queries that span services:

API Composition - the application performs the join rather than the database.

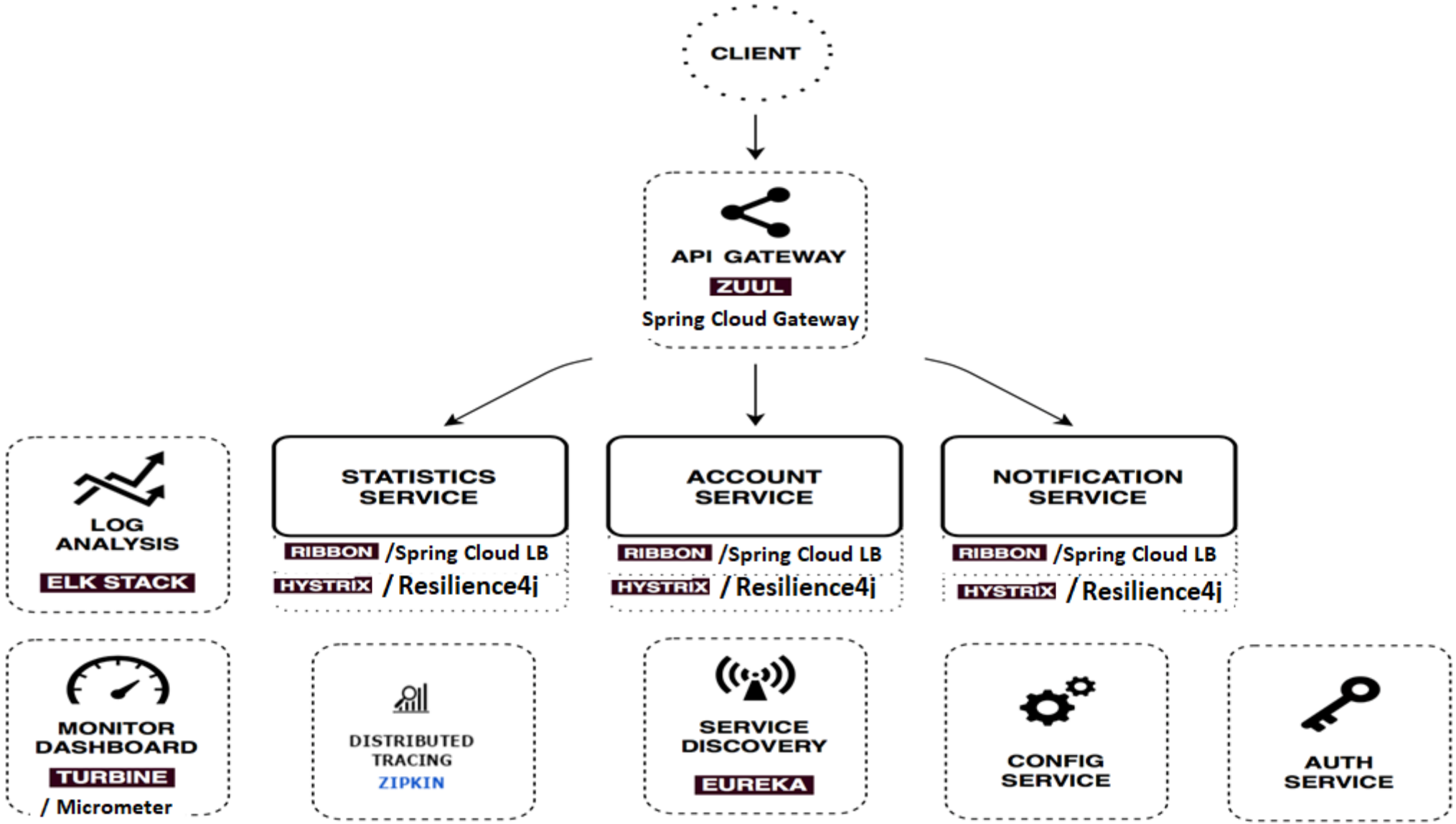
Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data.

A microservices architecture is a method of developing software applications as a suite of:

- ❑ independently deployable,
- ❑ small,
- ❑ Each service runs a unique process and
- ❑ communicates through a well-defined, lightweight mechanism to serve a business goal.

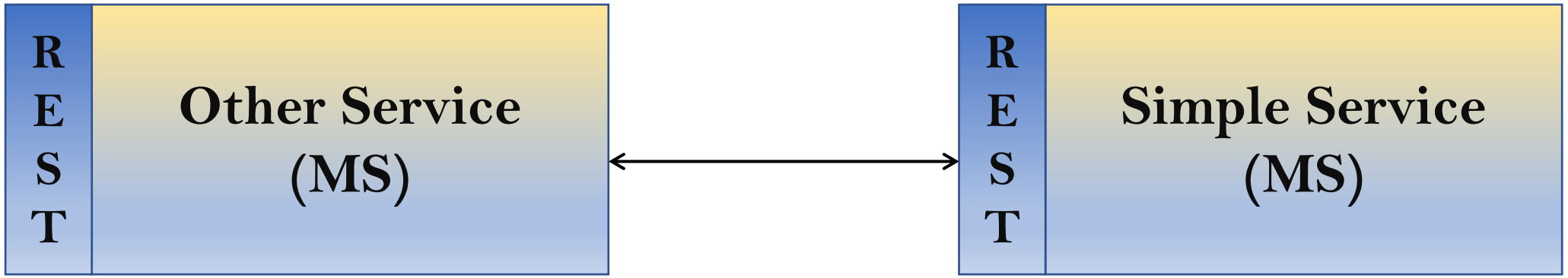
Microservice Patterns:

- ❑ API gateway
- ❑ Service registry
- ❑ Circuit breaker
- ❑ Messaging
- ❑ Database per Service
- ❑ Access Token
- ❑ Saga
- ❑ Event Sourcing & CQRS


CLIENT
API GATEWAY
ZUUL
Spring Cloud Gateway
STATISTICS SERVICE
ACCOUNT SERVICE
NOTIFICATION SERVICE
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
LOG ANALYSIS
ELK STACK
MONITOR DASHBOARD
TURBINE
/ Micrometer
DISTRIBUTED TRACING
ZIPKIN
SERVICE DISCOVERY
EUREKA
CONFIG SERVICE
AUTH SERVICE

Problem

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?



Solution

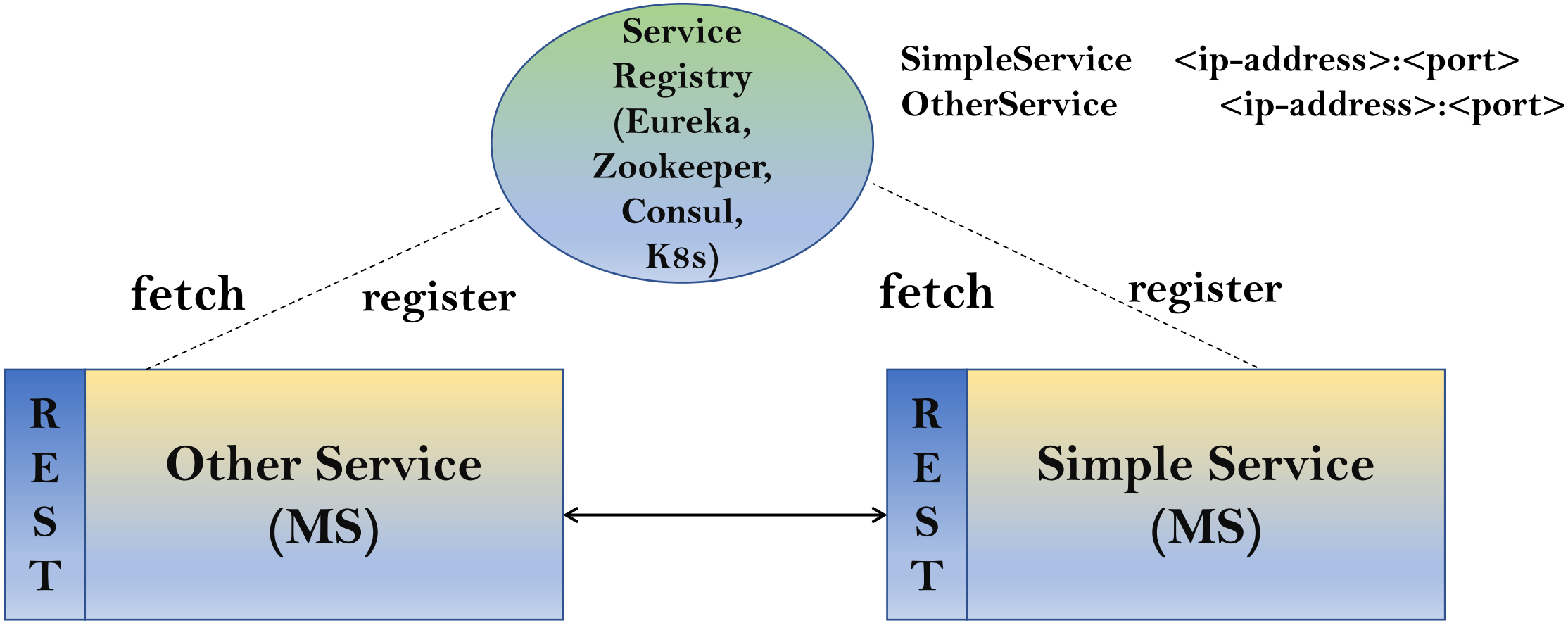
Implement a service registry, which is a database of services, their instances and their locations.

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.

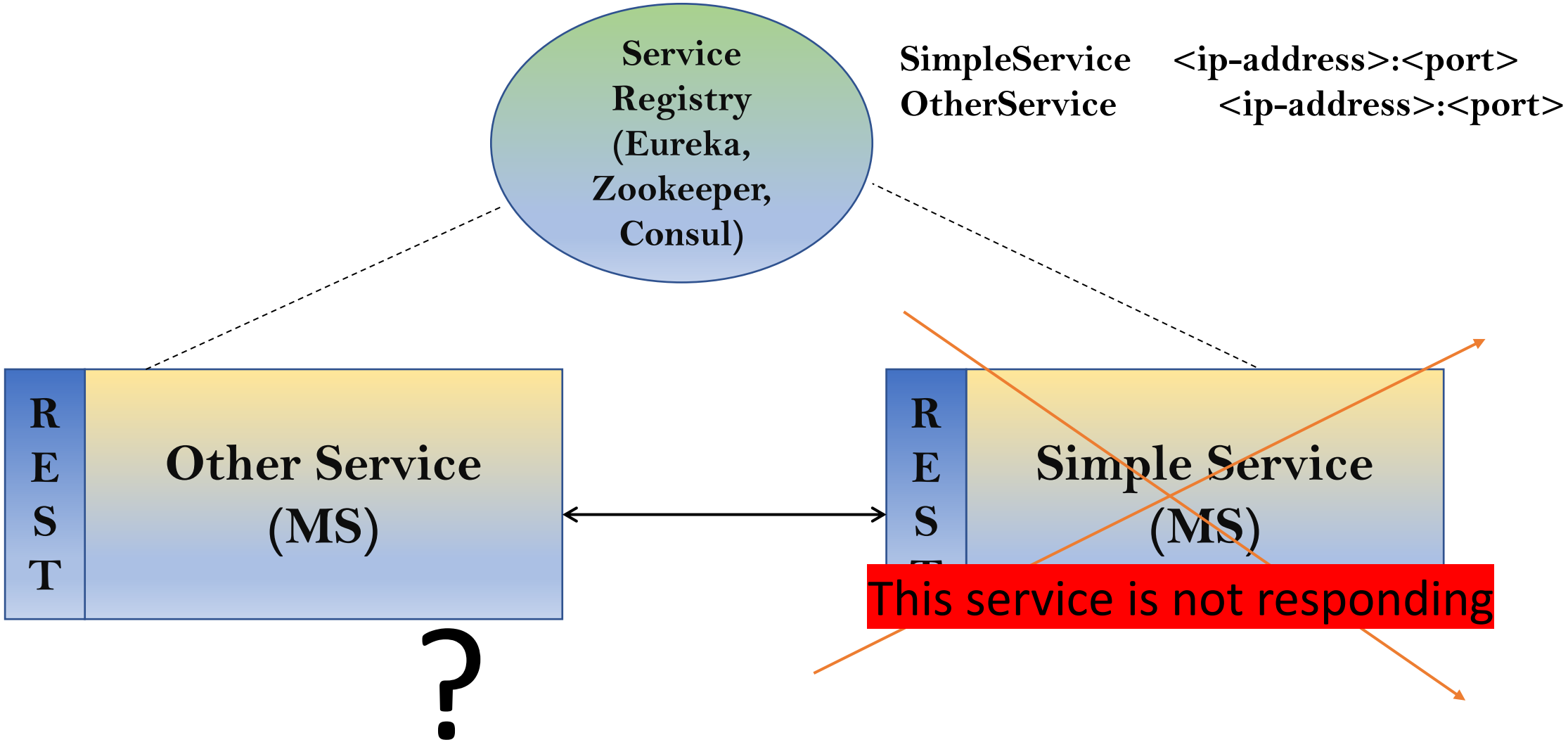
A service registry might invoke a service instance’s health check API to verify that it is able to handle requests

Service registry Pattern



Problem

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



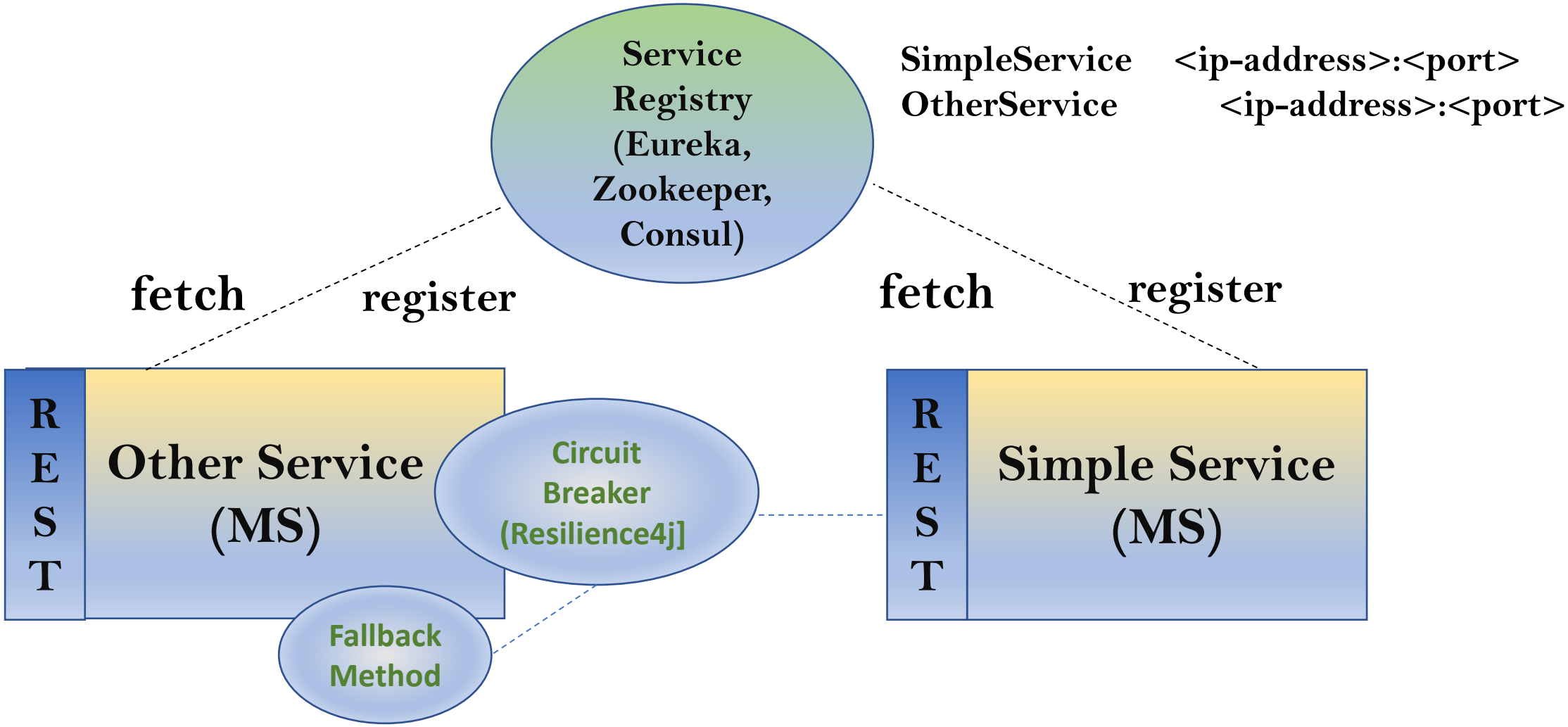
Solution

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.

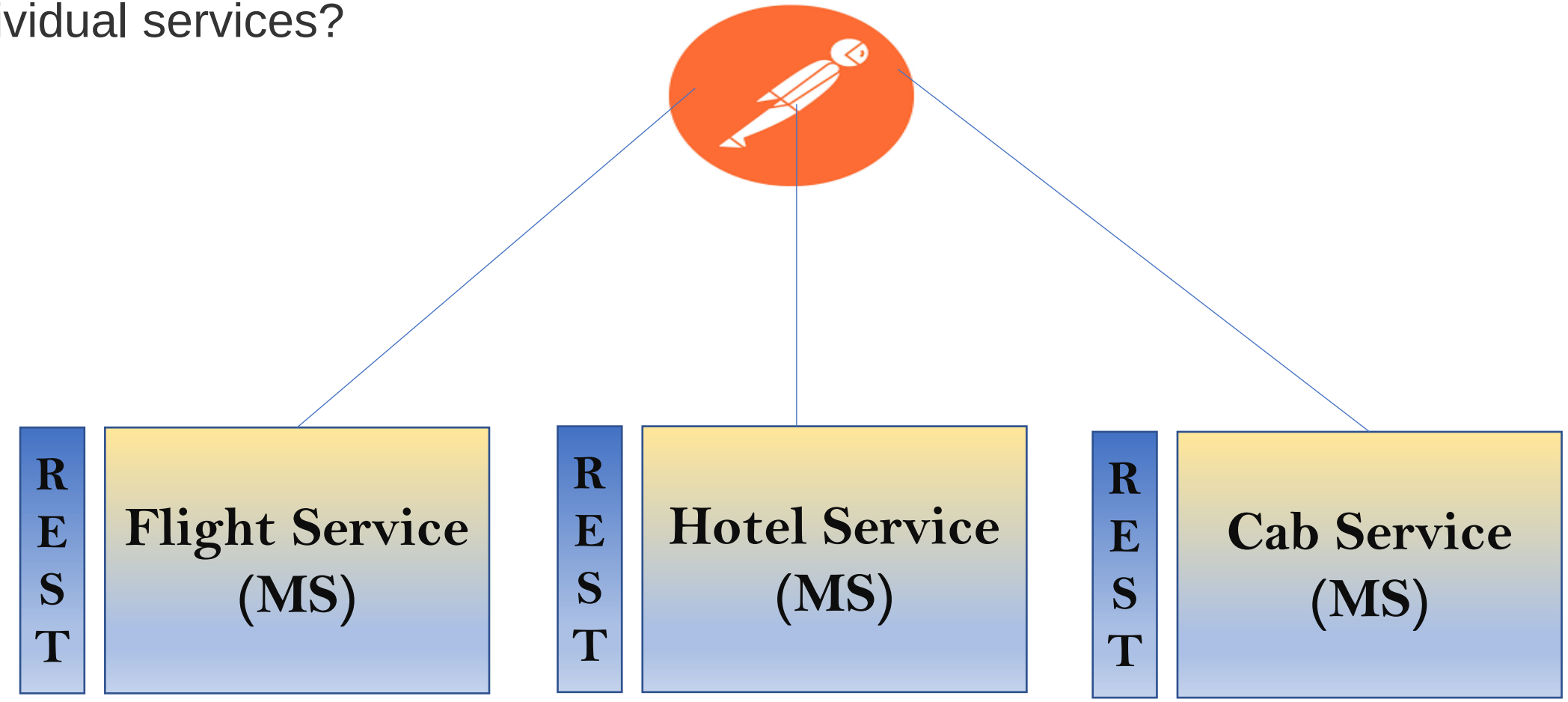
After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

Circuit Breaker Pattern



Problem

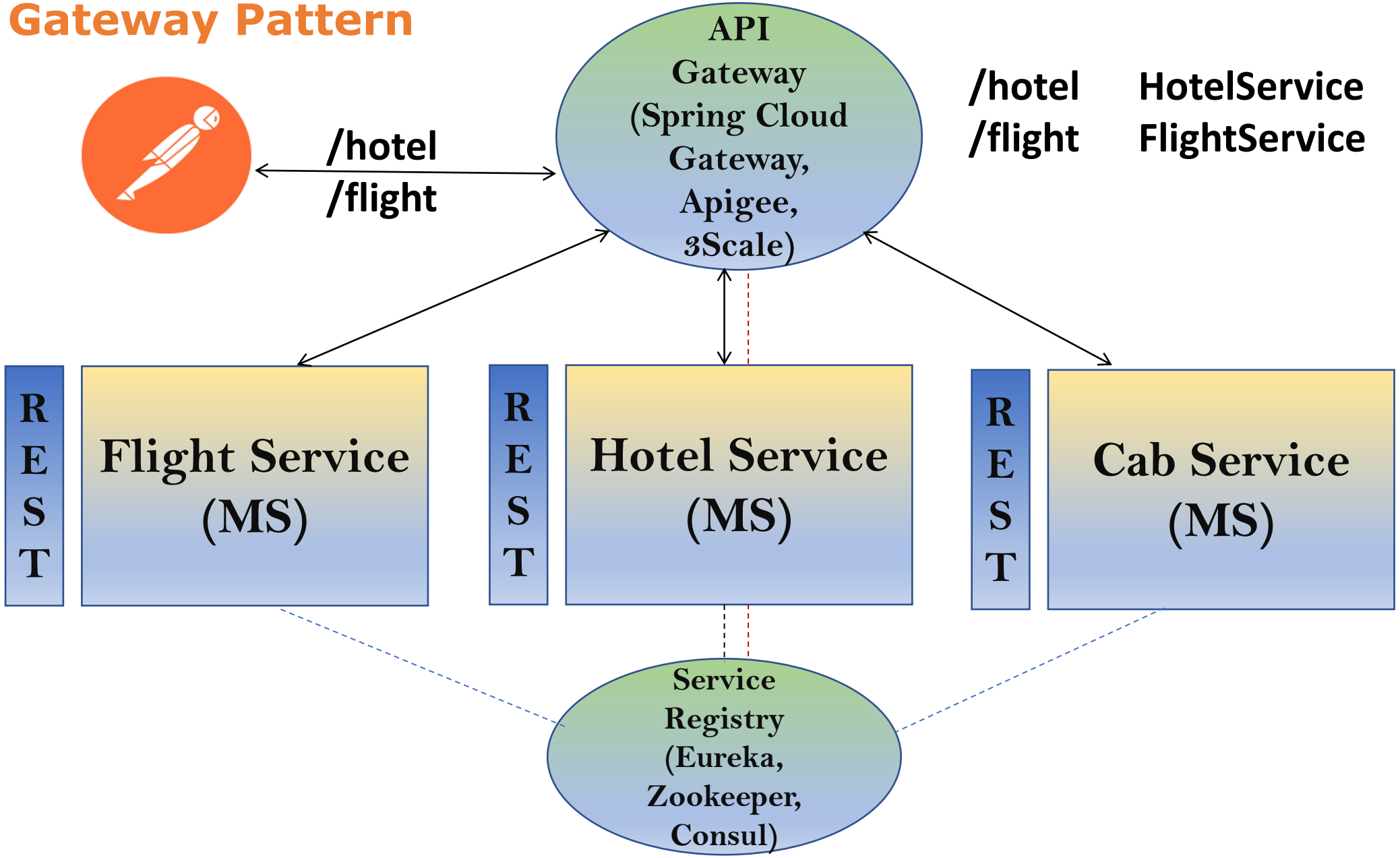
How do the clients of a Microservices-based application access the individual services?



Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles

API Gateway Pattern



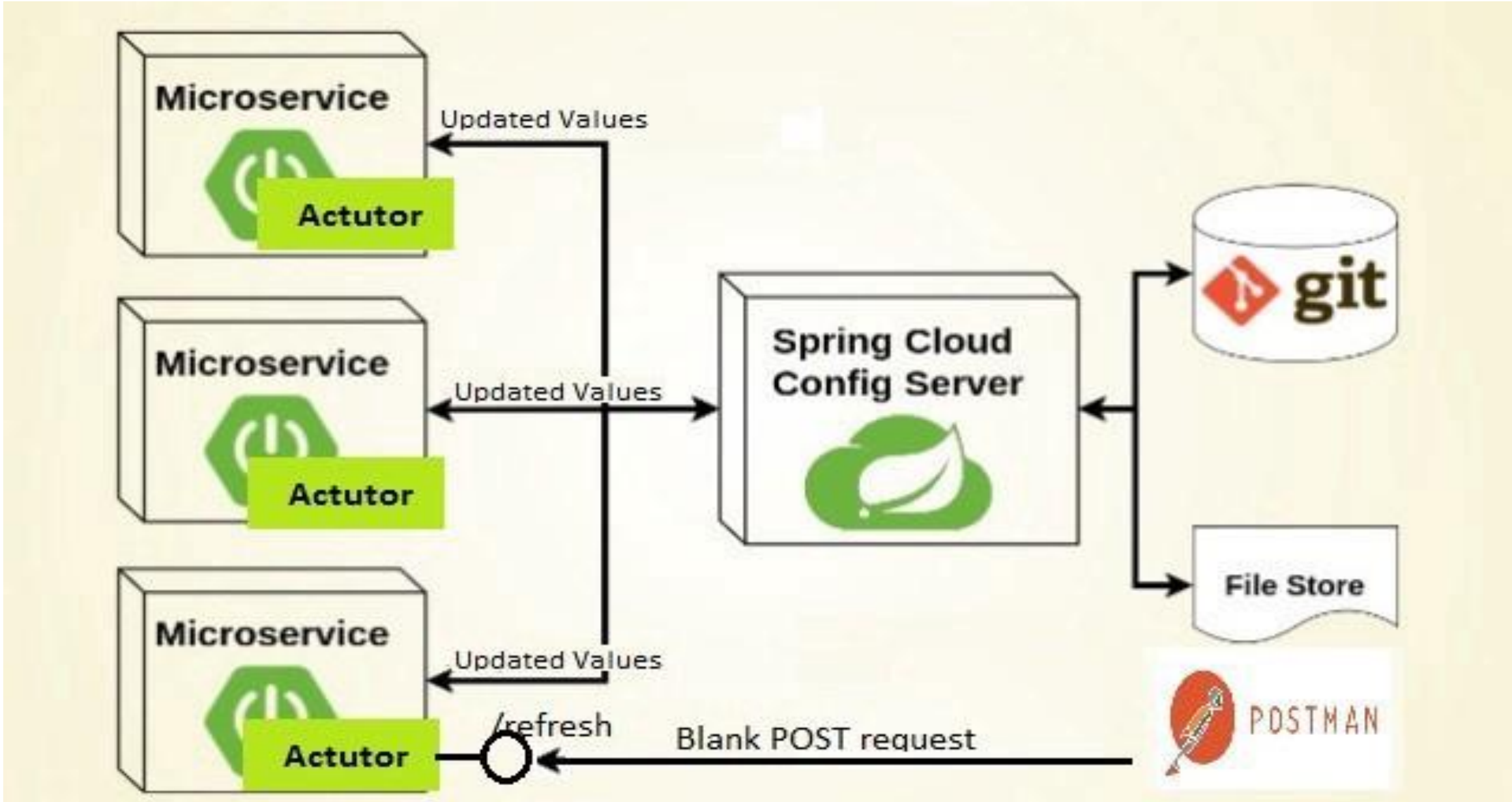
Pattern: Externalized configuration

Problem

How to enable a service to run in multiple environments without modification?

Solution

Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.


Microservice
Actutor
Microservice
Actutor
Microservice
Actutor
Updated Values
Updated Values
Updated Values
/refresh
Blank POST request
Spring Cloud
Config Server
git
File Store
POSTMAN

Container design patterns (Distributed Systems)

- The sidecar design pattern
- The ambassador design pattern
- The adapter design pattern
- The leader election design pattern
- The work queue design pattern
- The scatter/gather design pattern

Microservices Tooling Supports

1. Using Spring for creating Microservices

- ❑ Setup new service by using Spring Boot
- ❑ Expose resources via a RestController
- ❑ Consume remote services using RestTemplate/Feign

2. Adding Spring Cloud and Discovery server

- It is building blocks for Cloud and Microservices
- It provides microservices infrastructure like provide use services such as Service Discovery, Configuration server and Monitoring.
- It provides several other open source projects like Netflix OSS

.

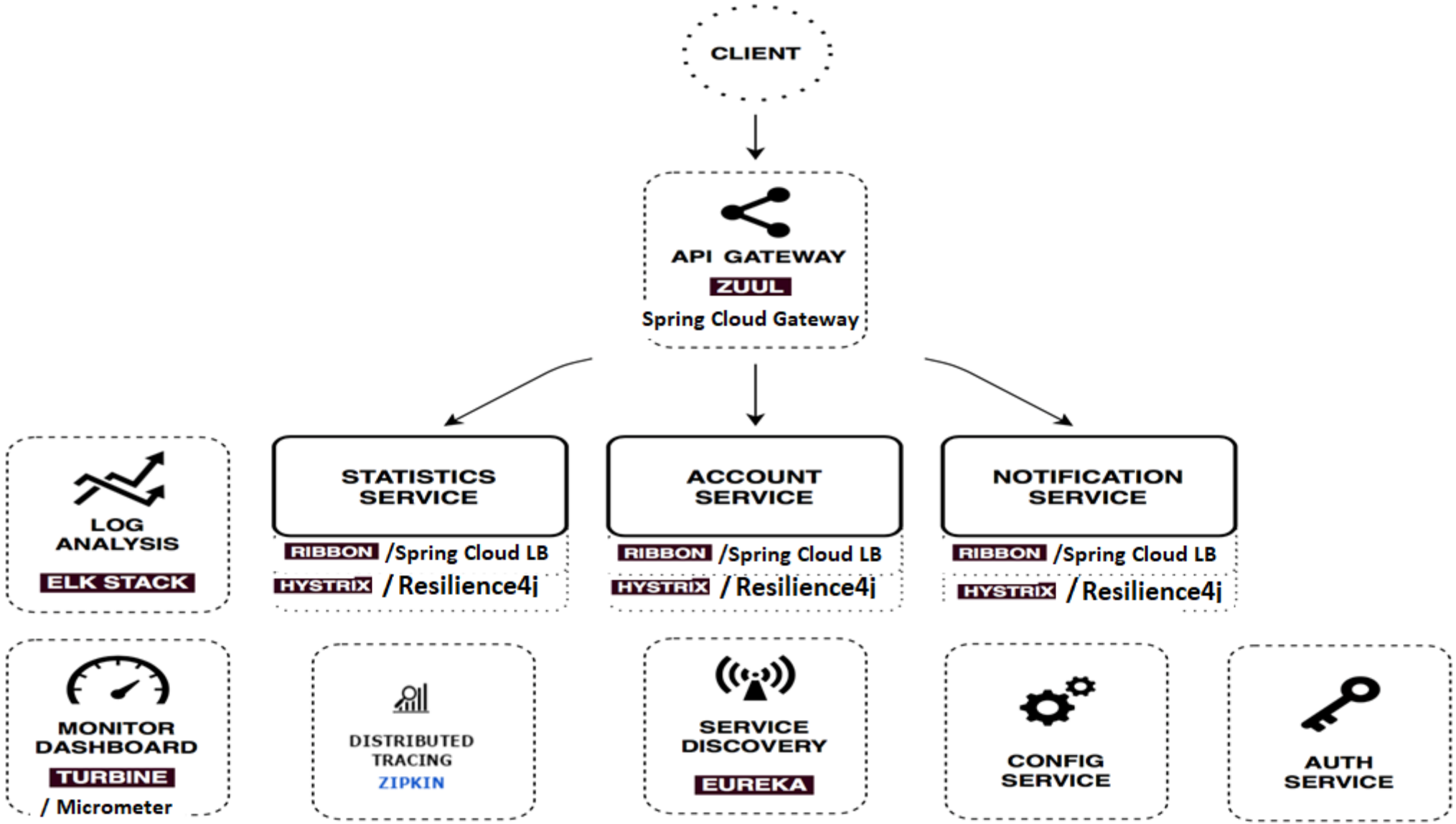
- It provides PaaS like Cloud Foundry, AWS etc.,
- It uses Spring Boot style starters

spring-cloud-netflix

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.

The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon)


CLIENT
API GATEWAY
ZUUL
Spring Cloud Gateway
STATISTICS SERVICE
ACCOUNT SERVICE
NOTIFICATION SERVICE
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
RIBBON /Spring Cloud LB
HYSTRIX / Resilience4j
LOG ANALYSIS
ELK STACK
MONITOR DASHBOARD
TURBINE
/ Micrometer
DISTRIBUTED TRACING
ZIPKIN
SERVICE DISCOVERY
EUREKA
CONFIG SERVICE
AUTH SERVICE

Service Discovery:

Eureka instances can be registered and clients can discover the instances using Spring-managed beans

Circuit Breaker:

Hystrix/Resilience4j clients can be built with a simple annotation-driven method decorator
Embedded Hystrix/Micrometer dashboard with declarative Java configuration

Declarative REST Client:

Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations

Client Side Load Balancer:

Ribbon / Spring Cloud Load Balancer

External Configuration:

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Router and Filter:

Automatic registration of **Zuul/Spring Cloud Gateway** filters, and a simple convention over configuration approach to reverse proxy creation

ELK – Elasticsearch, Logstash, Kibana:

three different tools usually used together.

They are used for searching, analyzing, and visualizing log data in a real time.

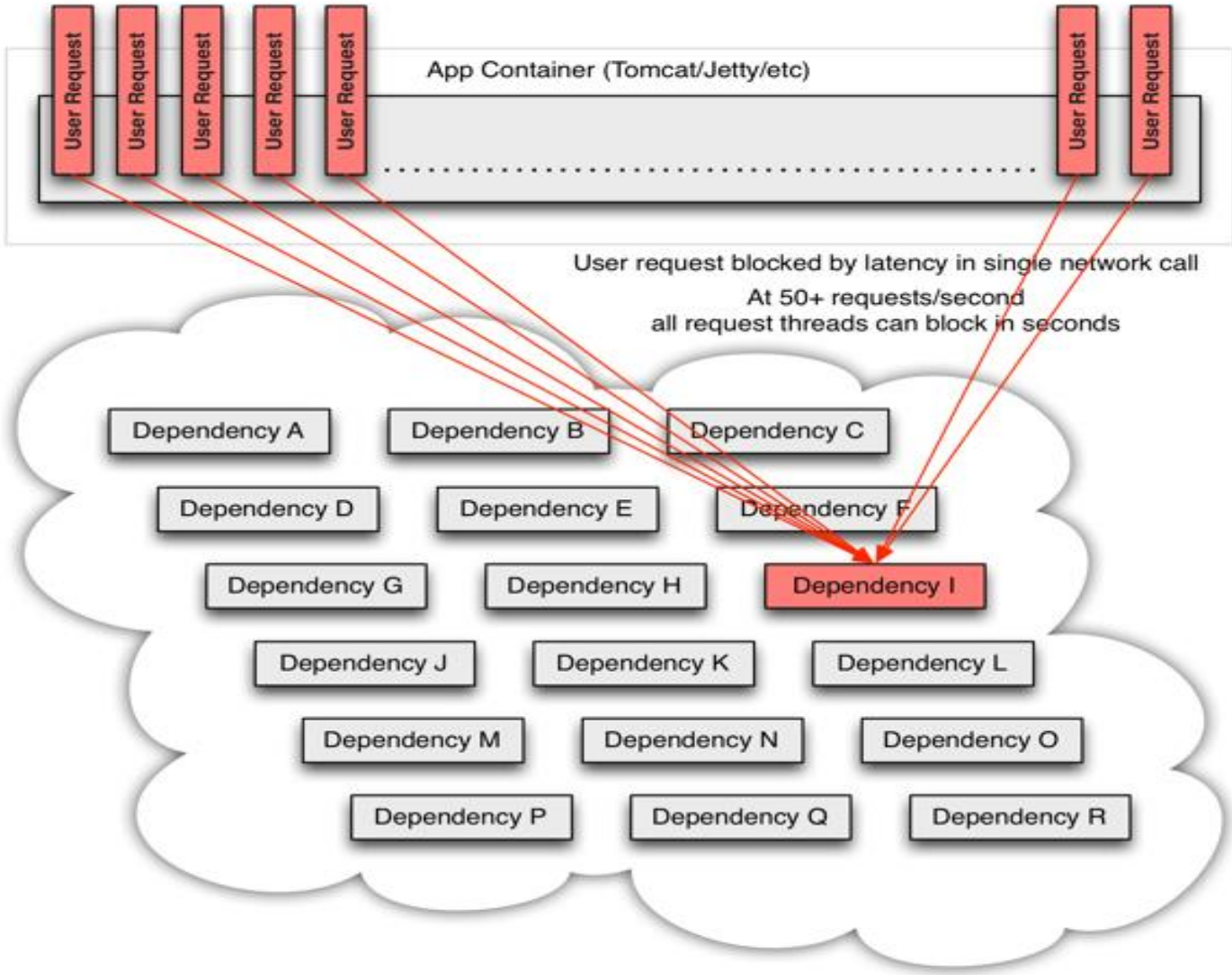
Zipkin is a distributed tracing system

It helps gather timing data needed to troubleshoot latency problems in microservice architectures.

It manages both the collection and lookup of this data.

Latency and Fault Tolerance for Distributed Systems
(Circuit Breaker Pattern – Hystrix / Resilience4j)

With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.



These issues are even worse when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.

All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

Pattern: Circuit Breaker

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

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.

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.

CURRENT	REPLACEMENT
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

Resilience4j

Resilience4j has been inspired by Netflix Hystrix but is designed for Java 8 and functional programming.

It is lightweight compared to Hystrix as it has the Vavr library as its only dependency.

Vavr (Javaslang) core is a functional library for Java. It helps to reduce the amount of code and to increase the robustness.

A first step towards functional programming is to start thinking in immutable values.

Vavr provides immutable collections and the necessary functions and control structures to operate on these values.


```java
@Bulkhead(name = "bulkHeadPostToES")
public String postCallProxyES (String endpoint, String payload) {
        return this.initiateCallEsProxy (endpoint, payload);
}
```

```yaml
bulkhead:
  backends:
    bulkHeadPostToES:
      maxWaitDuration: 20ms
      maxConcurrentCalls: 20

thread-pool-bulkhead:
  backends:
    bulkHeadPostToES:
      maxThreadPoolSize: 1
      coreThreadPoolSize: 1
      queueCapacity: 1
```

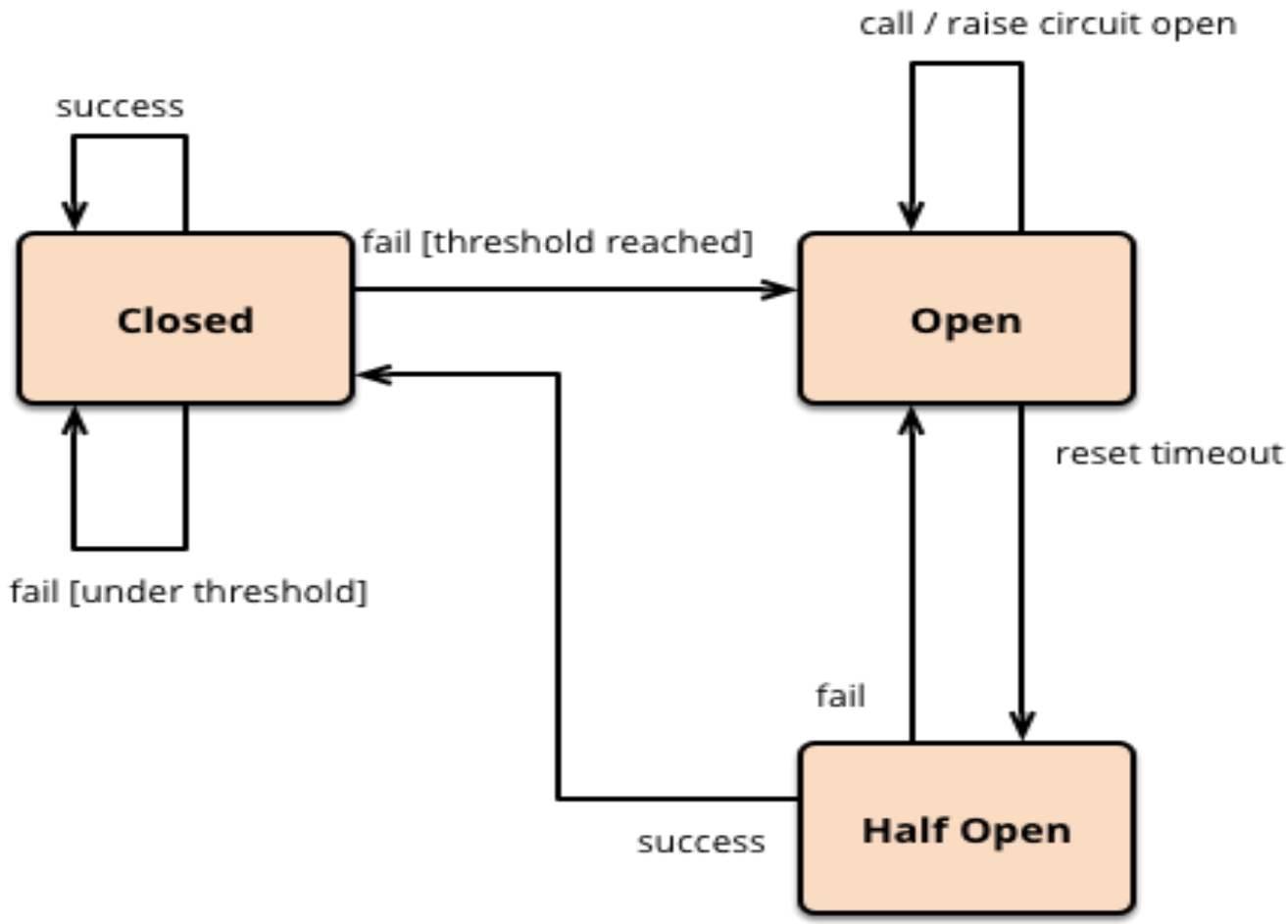
CircuitBreaker

When a service invokes another service, there is always a possibility that it may be down or having high latency.

This may lead to exhaustion of the threads as they might be waiting for other requests to complete.

This pattern functions in a similar fashion to an electrical Circuit Breaker:

- When a number of consecutive failures cross the defined threshold, the Circuit Breaker trips.
- For the duration of the timeout period, all requests invoking the remote service will fail immediately.
- 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.


call / raise circuit open
success
fail [threshold reached]
Closed
Open
reset timeout
fail [under threshold]
fail
success
Half Open

RateLimiter

Rate Limiting pattern ensures that a service accepts only a defined maximum number of requests during a window.

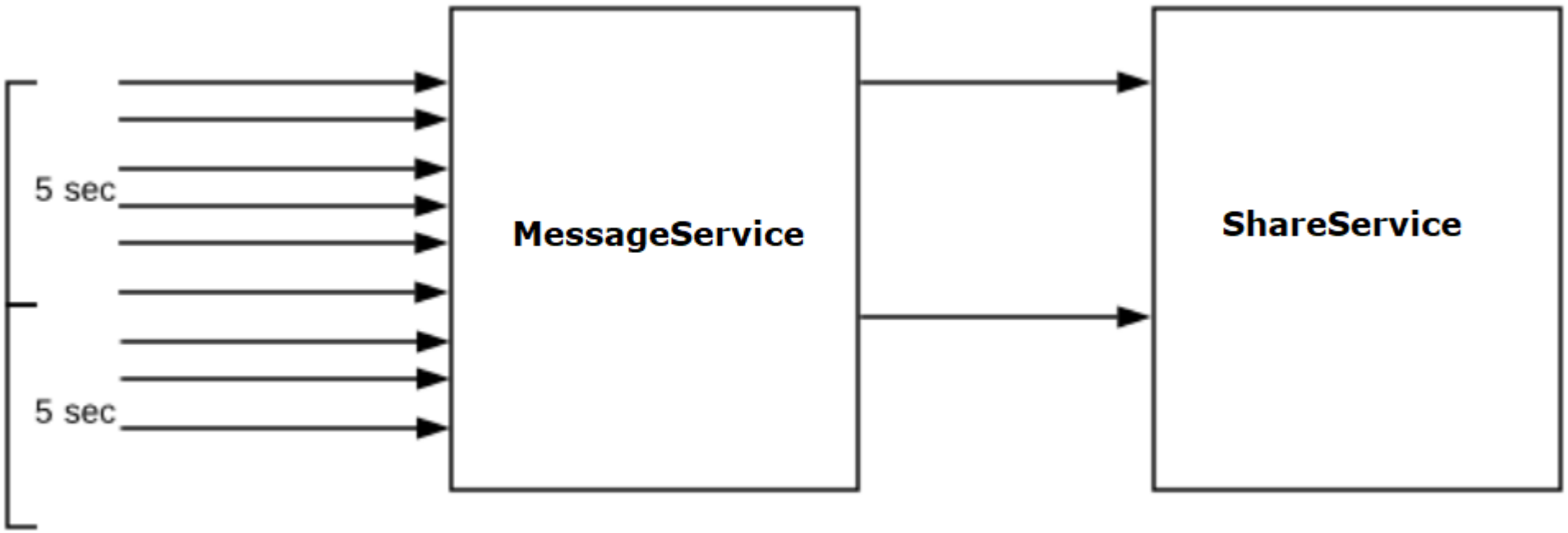
This ensures that underline resources are used as per their limits and don't exhaust.

Retry

Retry pattern enables an application to handle transient failures while calling to external services.

It ensures retrying operations on external resources a set number of times. If it doesn't succeed after all the retry attempts, it should fail and response should be handled gracefully by the application.

RateLimiter



Bulkhead

Bulkhead ensures the failure in one part of the system doesn't cause the whole system down. It controls the number of concurrent calls a component can take. This way, the number of resources waiting for the response from that component is limited.

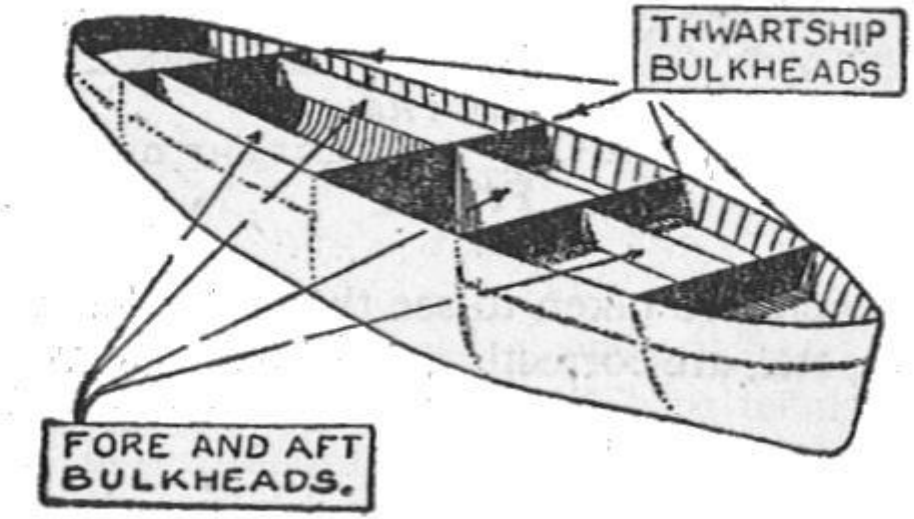
There are two types of bulkhead implementation:

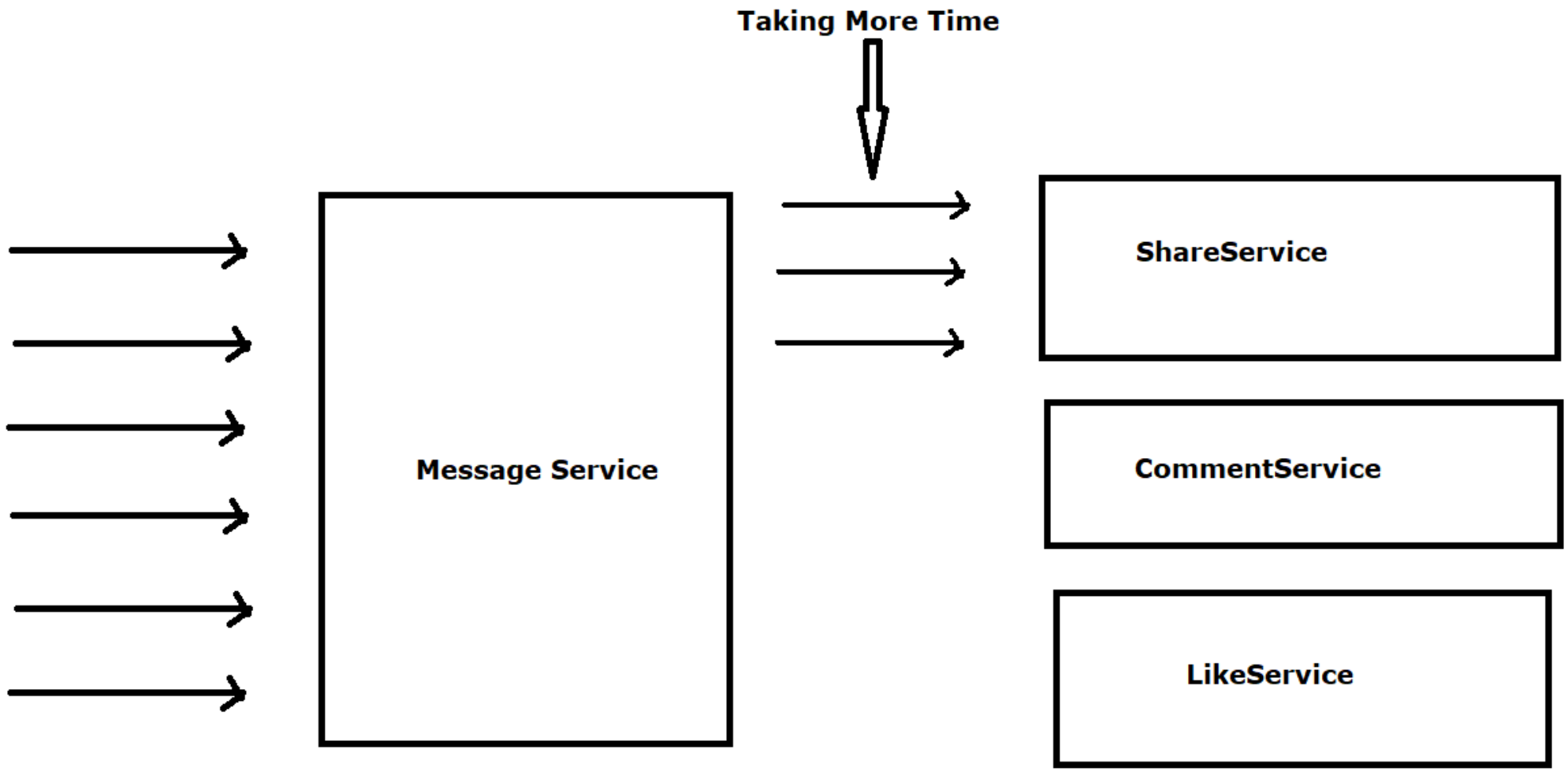
The semaphore isolation approach limits the number of concurrent requests to the service. It rejects requests immediately once the limit is hit. The thread pool isolation approach uses a thread pool to separate the service from the caller and contain it to a subset of system resources.

The thread pool approach also provides a waiting queue, rejecting requests only when both the pool and queue are full. Thread pool management adds some overhead, which slightly reduces performance compared to using a semaphore, but allows hanging threads to time out.

A ship is split into small multiple compartments using Bulkheads. Bulkheads are used to seal parts of the ship to prevent entire ship from sinking in case of flood.

Similarly failures should be expected when we design software. The application should be split into multiple components and resources should be isolated in such a way that failure of one component is not affecting the other.




Taking More Time
Message Service
ShareService
CommentService
LikeService

CircuitBreaker

Config property	Default Value	Description
failureRateThreshold	50	Configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallRateThreshold	100	Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than `slowCallDurationThreshold` When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallDurationThreshold	60000 [ms]	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
permittedNumberOfCalls InHalfOpenState	10	Configures the number of permitted calls when the CircuitBreaker is half open.

slidingWindowType	COUNT_BASED	Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. Sliding window can either be count-based or time-based. If the sliding window is COUNT_BASED, the last `slidingWindowSize` calls are recorded and aggregated. If the sliding window is TIME_BASED, the calls of the last `slidingWindowSize` seconds recorded and aggregated.
slidingWindowSize	100	Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.

minimumNumberOfCalls	10	Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate. For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.
waitDurationInOpenState	60000 [ms]	The time that the CircuitBreaker should wait before transitioning from open to half-open.

Create and configure a Bulkhead

You can provide a custom global BulkheadConfig. In order to create a custom global BulkheadConfig, you can use the BulkheadConfig builder. You can use the builder to configure the following properties.

Config property	Default value	Description
maxConcurrentCalls	25	Max amount of parallel executions allowed by the bulkhead
maxWaitDuration	0	Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

RateLimiter

Config property	Default value	Description
timeoutDuration	5 [s]	The default wait time a thread waits for a permission
limitRefreshPeriod	500 [ns]	The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
limitForPeriod	50	The number of permissions available during one limit refresh period

Retry

Config property	Default value	Description
maxAttempts	3	The maximum number of retry attempts
waitDuration	500 [ms]	A fixed wait duration between retry attempts
intervalFunction	numOfAttempts -> waitDuration	A function to modify the waiting interval after a failure. By default the wait duration remains constant.
retryOnResultPredicate	result -> false	Configures a Predicate which evaluates if a result should be retried. The Predicate must return true, if the result should be retried, otherwise it must return false.
retryOnExceptionPredicate	throwable -> true	Configures a Predicate which evaluates if an exception should be retried. The Predicate must return true, if the exception should be retried, otherwise it must return false.
retryExceptions	empty	Configures a list of error classes that are recorded as a failure and thus are retried.
ignoreExceptions	empty	Configures a list of error classes that are ignored and thus are not retried.

```
RetryConfig config = RetryConfig.custom()
  .maxAttempts(2)
  .waitDuration(Duration.ofMillis(1000))
  .retryOnResult(response -> response.getStatus() == 500)
  .retryOnException(e -> e instanceof WebServiceException)
  .retryExceptions(IOException.class, TimeoutException.class)
  .ignoreExceptions(BusinessException.class,
                    OtherBusinessException.class)
  .build();
```

resilience4j has the ability to add multiple fault tolerance features into one call. It is more configurable and amount of code needs to be written is less with right amount of abstractions.

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
Public String postCallProxyES(...) {
     ... . ..
}
```

The default Resilience4j Aspects order is the following:
Retry (CircuitBreaker (RateLimiter (TimeLimiter (Bulkhead (Function)))))

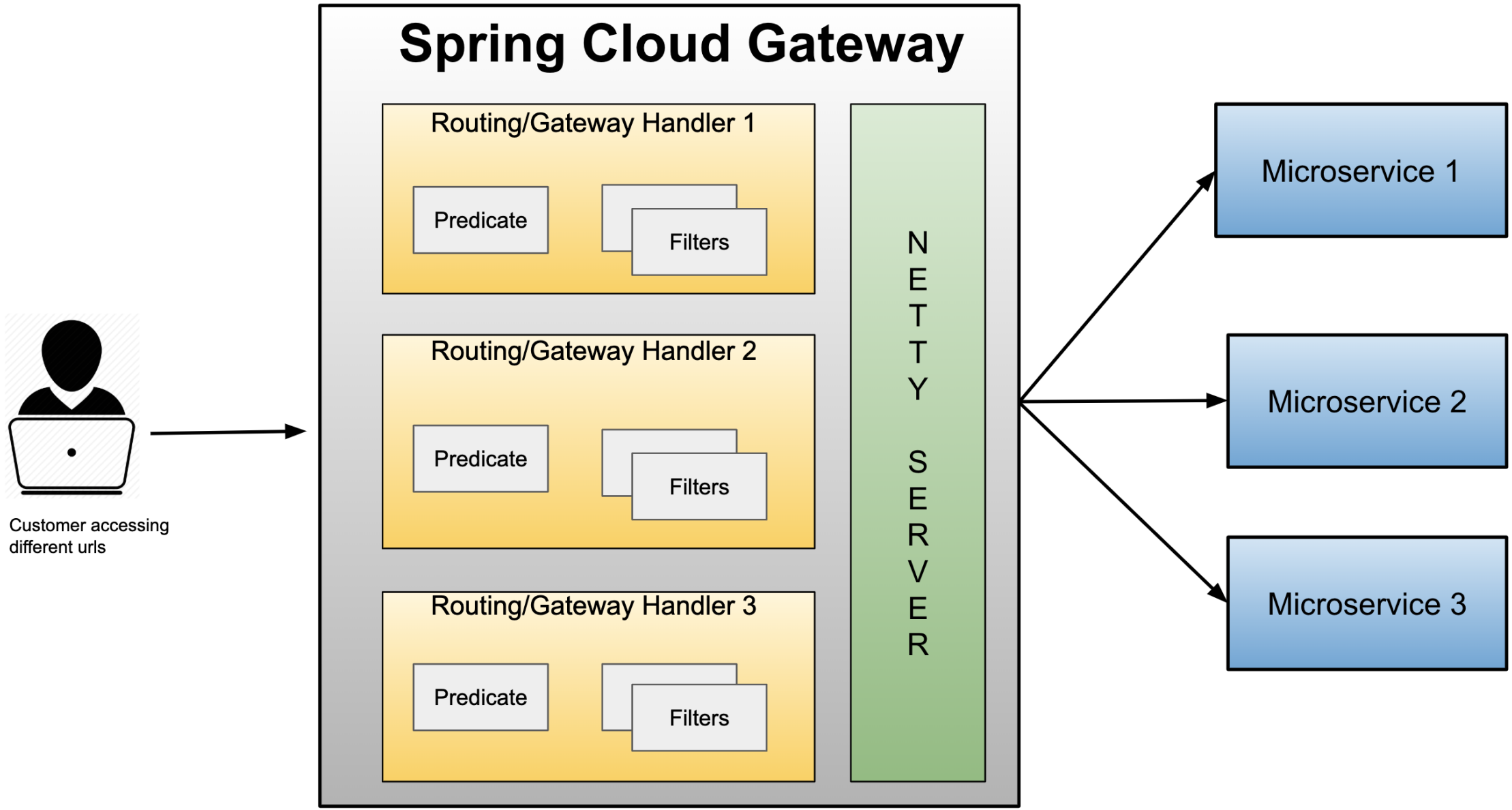
Bulkhead	Rate Limiter
Limit number of concurrent calls at a time.	Limit number total calls in given period of time
Ex: Allow 5 concurrent calls at a time	Ex: Allow 5 calls every 2 second.
In above example, first 5 calls will start processing in parallel while any further calls will keep waiting. As soon as one of those 5 in-process calls is finished, next waiting calls will be immediately eligible to be executed.	In above example, 2 second window starts & first 5 calls will start processing (may be parallel or not parallel). Those 5 calls might finish before 2 seconds, but next waiting calls will NOT be immediately eligible to be executed. After 2 seconds window is over, next 2 seconds window will start & then only next waiting calls will be eligible to be executed.

Spring Cloud Gateway

Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem.

It provides a simple and effective way to route incoming requests to the appropriate destination using Gateway Handler Mapping.

And Spring Cloud Gateway uses Netty server to provide non-blocking asynchronous request processing.


Spring Cloud Gateway
Routing/Gateway Handler 1
Predicate
Filters
Routing/Gateway Handler 2
Predicate
Filters
Routing/Gateway Handler 3
Predicate
Filters
NETTY SERVER
Microservice 1
Microservice 2
Microservice 3
Customer accessing different urls

Spring Cloud Gateway consists of 3 main building blocks:

Route: Think of this as the destination that we want a particular request to route to. It comprises of destination URI, a condition that has to satisfy — Or in terms of technical terms, Predicates, and one or more filters.

Predicate: This is literally a condition to match. i.e. kind of “if” condition. If requests has something

— e.g. path=blah or request header contains foo-bar etc. In technical terms, it is Java 8 Function Predicate

Filter: These are instances of Spring Framework WebFilter.

This is where you can apply your magic of modifying request or response.

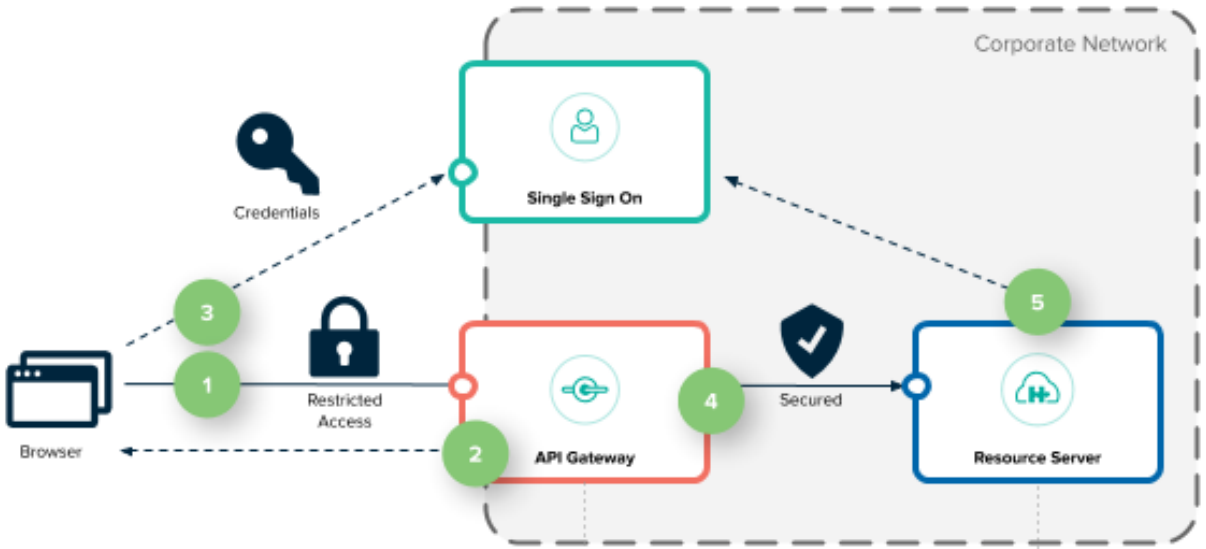
There are quite a lot of out of box WebFilter that framework provides

There are 2 different types of filters.

Pre Filters — if you want to add or change request object before we pass it down to destination service, you can use these filters.

Post Filters — if you want to add or change response object before we pass it back to client, you can use these filters.

Securing Services with Spring Cloud Gateway



1 User requests **`/resource`**

2 Gateway redirects browser to SSO

3 User authenticates & authorises the app

4 Request & JWT sent to the Resource Server

5 Resource Server checks user's JWT token

→ Users original intended path

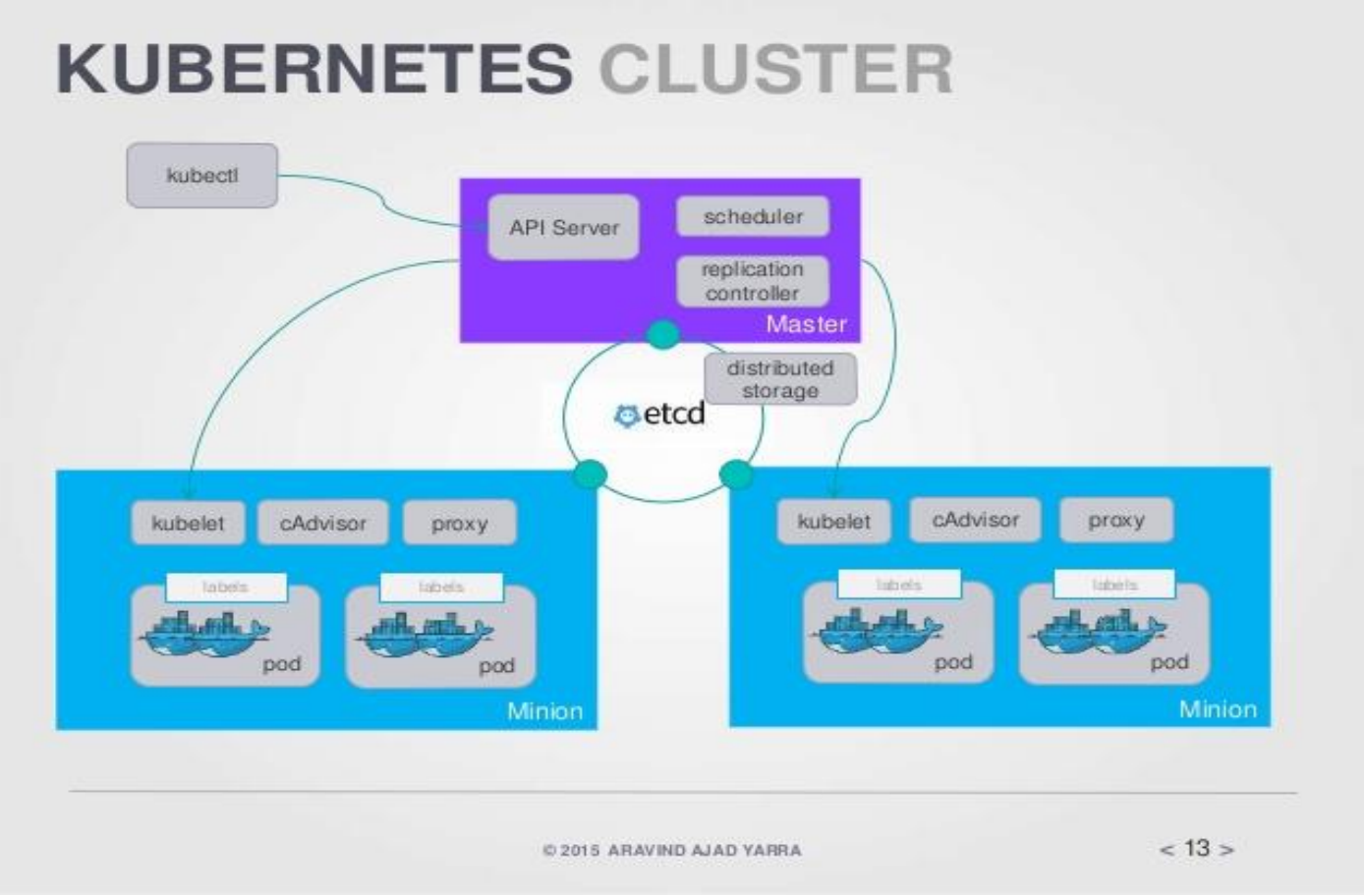
⇢ Additional OAuth / JOSE flows

org.springframework.boot:spring-boot-starter-webflux
org.springframework.cloud:spring-cloud-starter-gateway
org.springframework.cloud:spring-cloud-starter-security
org.springframework.boot:spring-boot-starter-oauth2-client

org.springframework.boot:spring-boot-starter-thymeleaf
org.thymeleaf.extras:thymeleaf-extras-springsecurity5

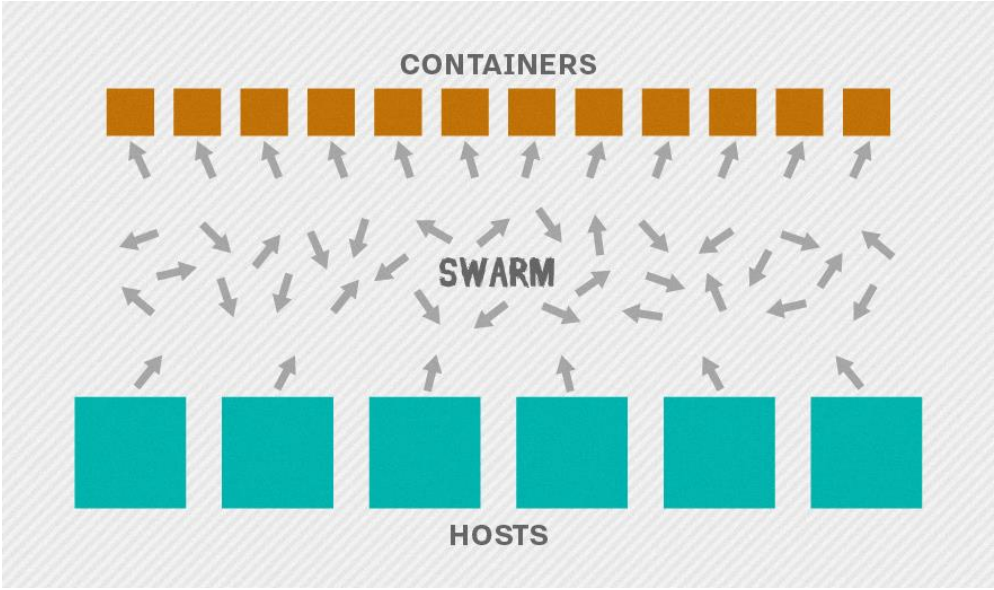
org.springframework.boot:spring-boot-starter-webflux
org.springframework.boot:spring-boot-starter-security
org.springframework.security:spring-security-oauth2-resource-server
org.springframework.security:spring-security-oauth2-jose

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.



Docker Swarm is a clustering and scheduling tool for Docker containers. With Swarm, IT administrators and developers can establish and manage a cluster of Docker nodes as a single virtual system

Swarm mode also exists natively for Docker Engine, the layer between the OS and container images. Swarm mode **integrates** the orchestration capabilities of Docker Swarm into **Docker Engine 1.12** and newer releases.



Pods

In Kubernetes pod is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pattern: Saga

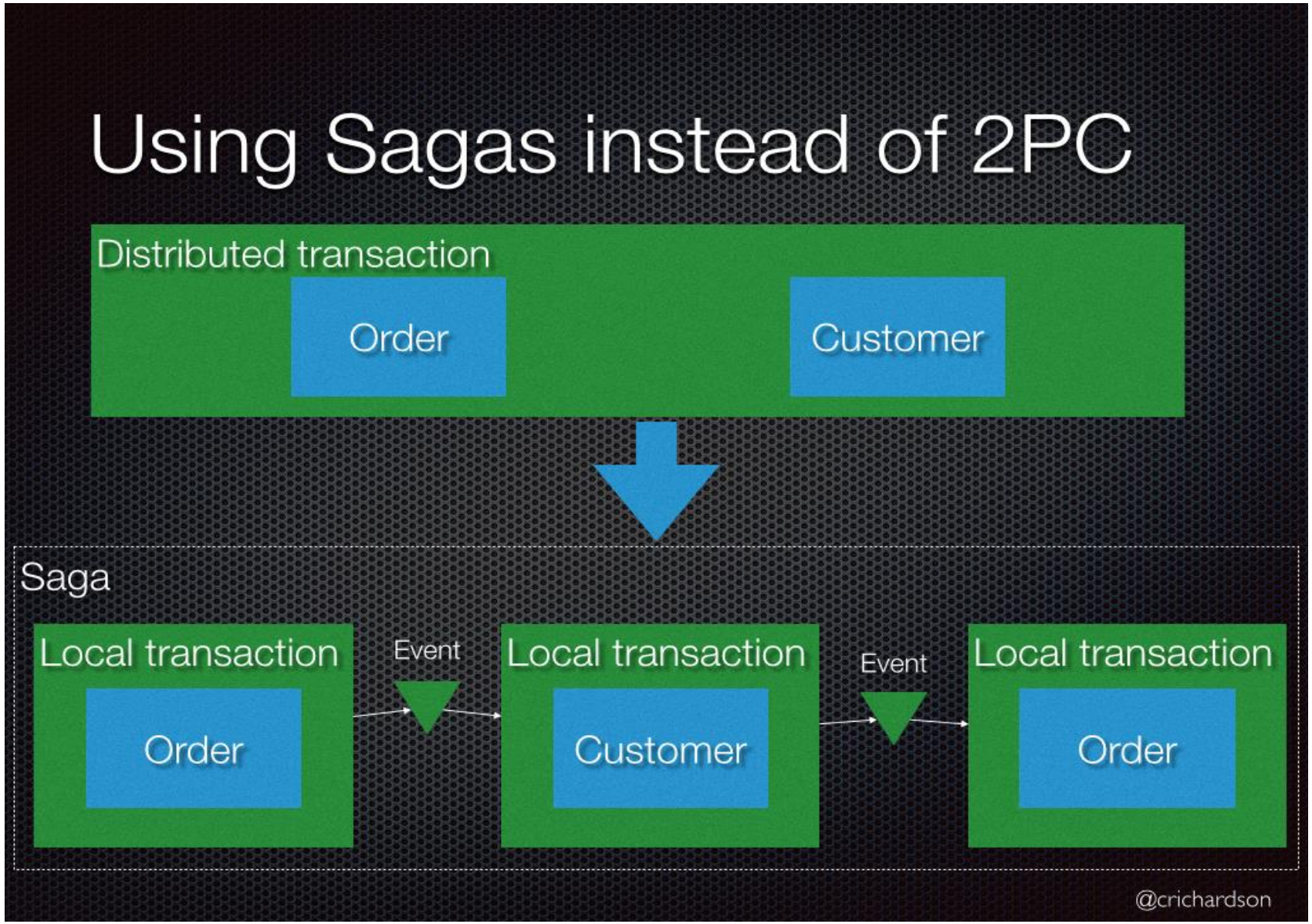
Context

You have applied the Database-server-per-service pattern.

Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.

For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit.

Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

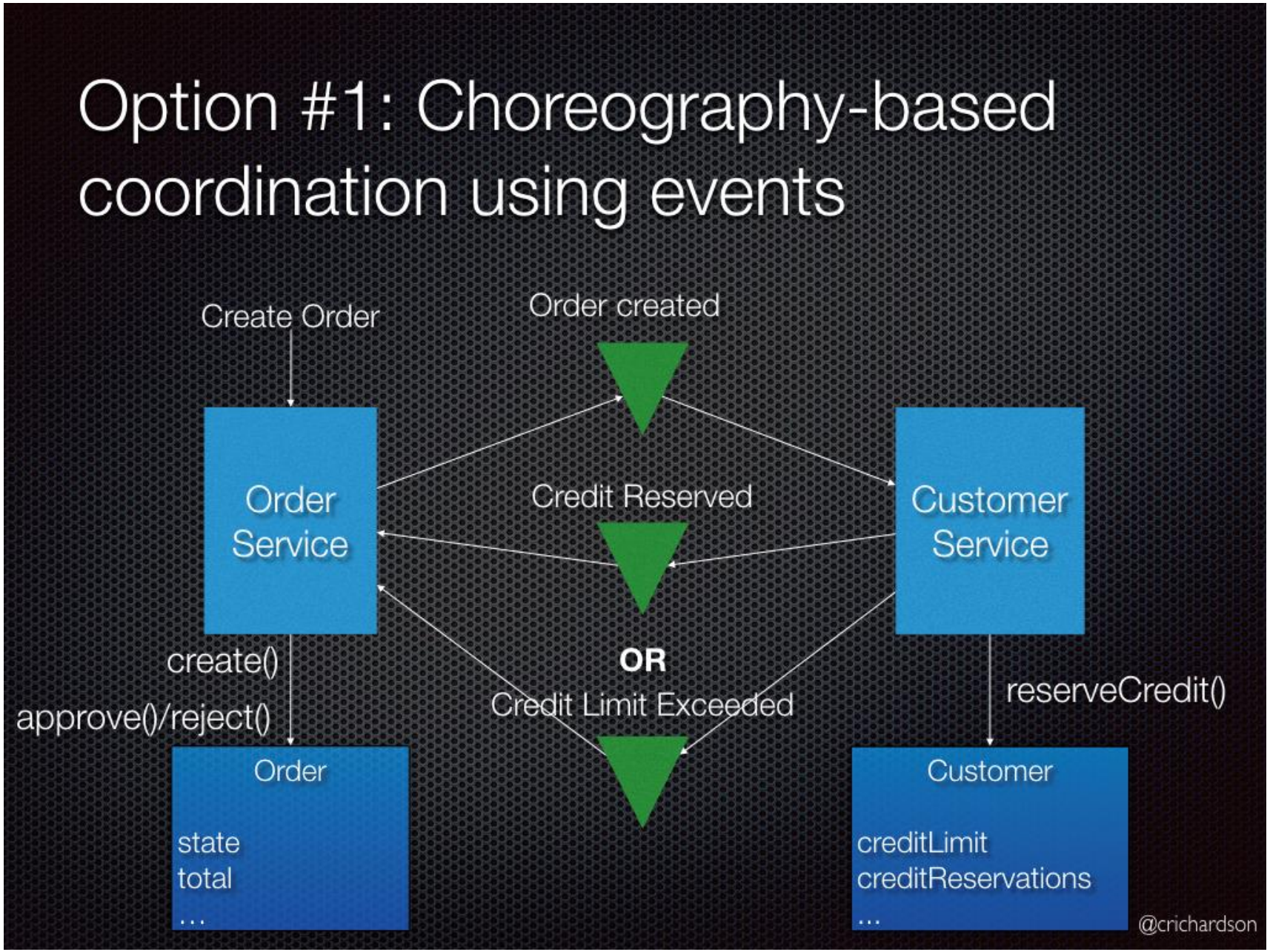

Using Sagas instead of 2PC
Distributed transaction
Order
Customer
Saga
Local transaction
Order
Event
Local transaction
Customer
Event
Local transaction
Order
@crichardson

There are two ways of coordination sagas:

Choreography - each local transaction publishes domain events that trigger local transactions in other services (Event Sourcing)

Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Option #1: Choreography-based coordination using events



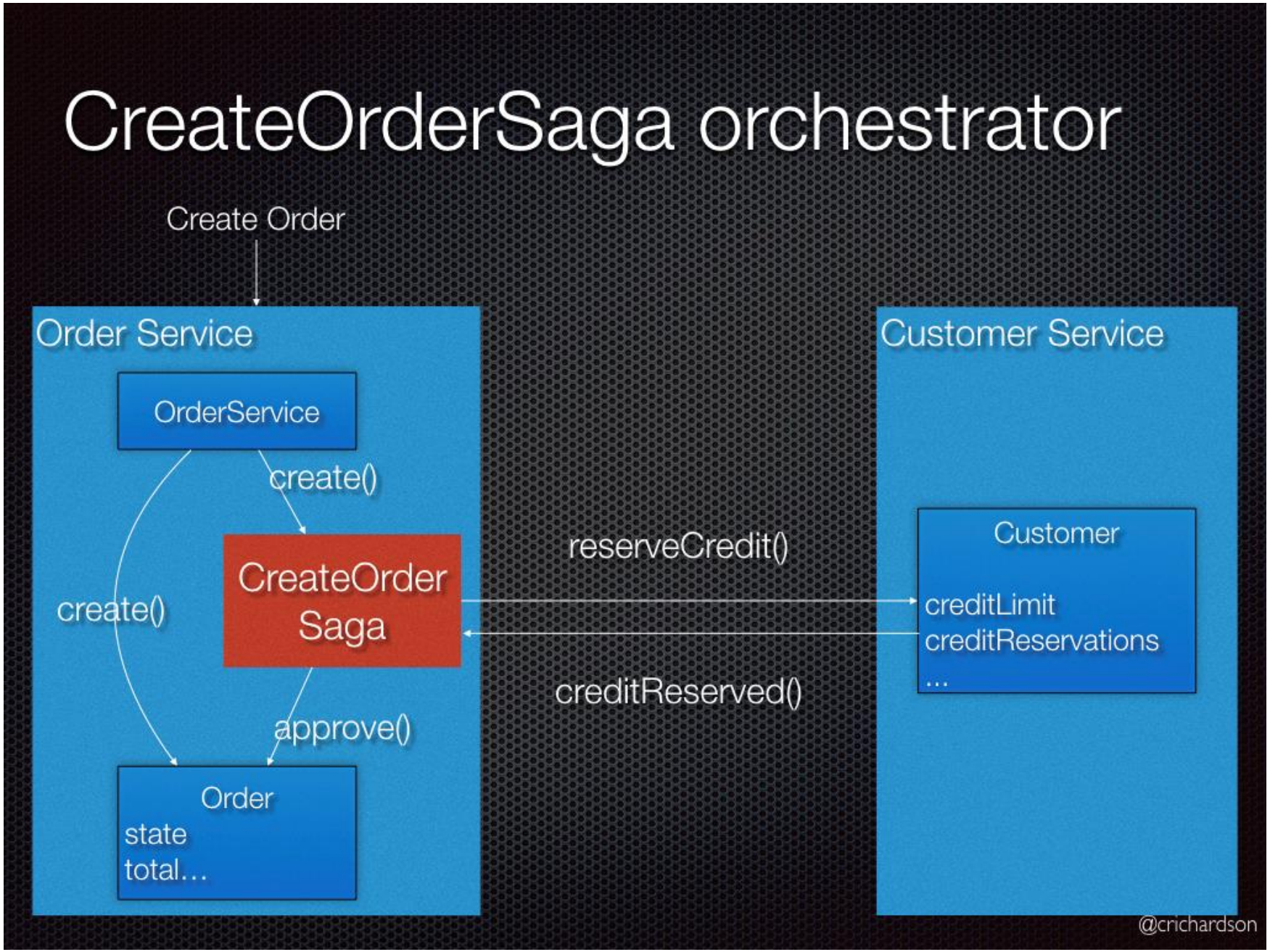
An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:

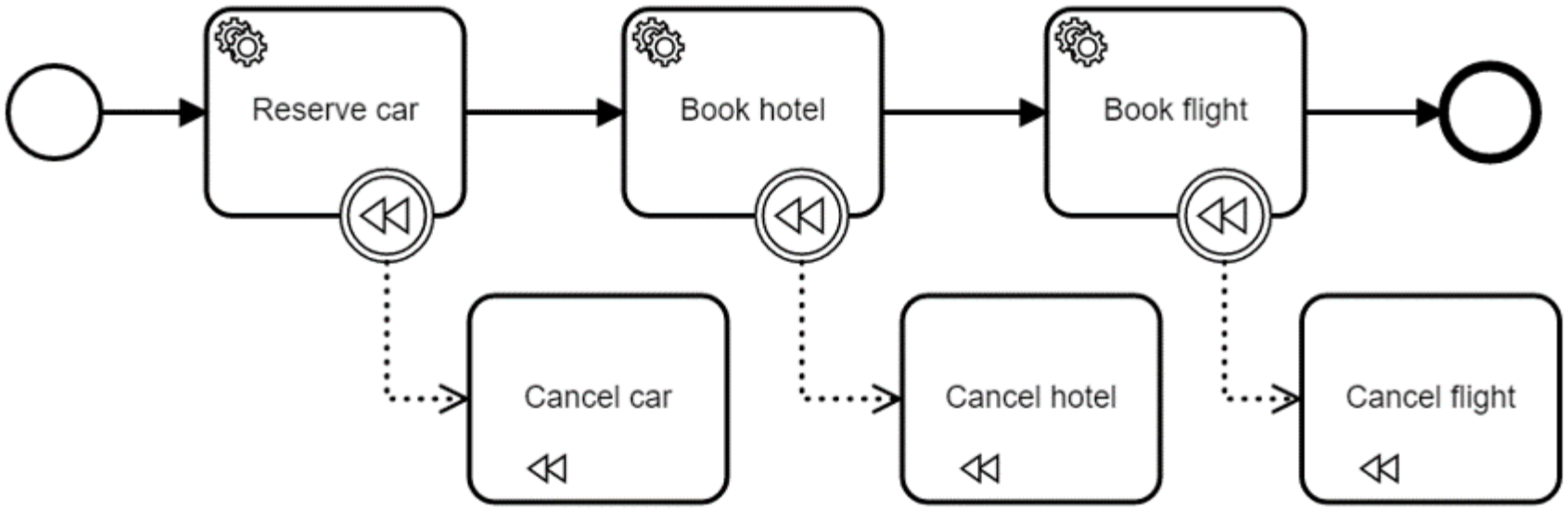
The Order Service creates an Order in a pending state and publishes an OrderCreated event

The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.

The Order Service receives the event and changes the state of the order to either approved or cancelled

CreateOrderSaga orchestrator





BPMN can act as Saga Coordinator!

three frameworks that presently support saga processing, namely Narayana LRA, Axon framework and Eventuate.io.

Narayana LRA

Narayana Long Running Actions is a specification developed by the Narayana team in the collaboration with the Eclipse MicroProfile initiative. The main focus is to introduce an API for coordinating long running activities with the assurance of the globally consistent outcome and without any locking mechanisms.

[https://github.com/eclipse/microprofile-sandbox/tree/master/proposals/0009-LRA]

Axon framework

Axon framework is Java based framework for building scalable and highly performant applications. Axon is based on the Command Query Responsibility Segregation (CQRS) pattern. The main motion is the event processing which includes the separated Command bus for updates and the Event bus for queries.

[http://www.axonframework.org/]

Eventuate.io

Eventuate is a platform that provides an event-driven programming model that focus on solving distributed data management in microservices architectures. Similarly to the Axon, it is based upon CQRS principles. The framework stores events in the MySQL database and it distributes them through the Apache Kafka platform.

[http://eventuate.io]