Domain Driven Design in Action

Mohamed Sweelam

Software Engineer

md.sweelam@gmail.com

July 6, 2024

Objectives

1. Provide good Arabic content for the topic

Objectives

1. Provide good Arabic content for the topic
2. Overview of DDD and Deep Dive in Microservices

Objectives

1. Provide good Arabic content for the topic
2. Overview of DDD and Deep Dive in Microservices
3. Move step forwards towards recent cloud tools

Objectives

1. Provide good Arabic content for the topic
2. Overview of DDD and Deep Dive in Microservices
3. Move step forwards towards recent cloud tools
4. Leave your fear, and let's do it

Table of Contents

Definition

Monolithic

A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.

Microservices

Microservices is a software development technique that arranges an application as a collection of loosely coupled services.

https://en.wikipedia.org/wiki/Monolithic_application
https://en.wikipedia.org/wiki/Microservices

Domain-Driven Design(DDD) is a collection of principles and patterns that help developers craft elegant object systems.

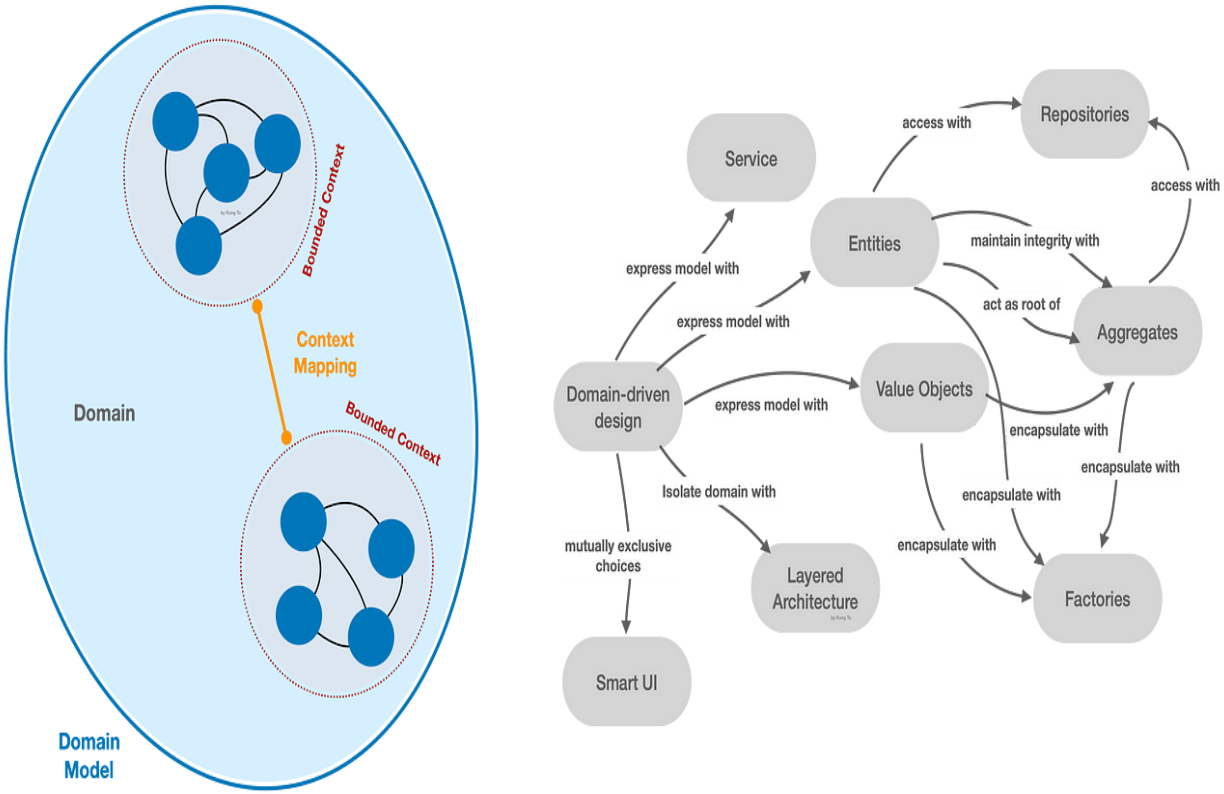


Figure: DDD Map

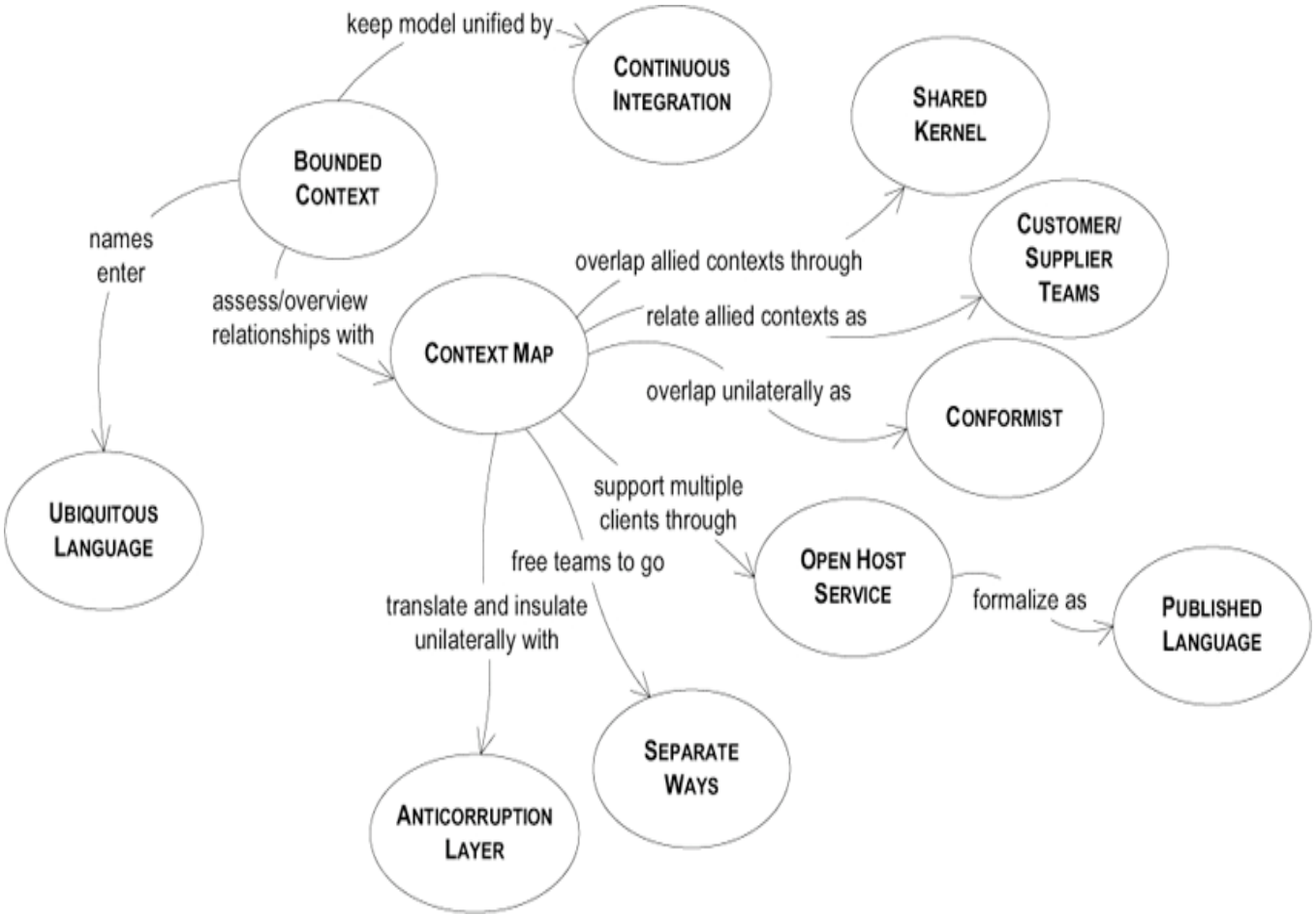


Figure: Strategic Design Map

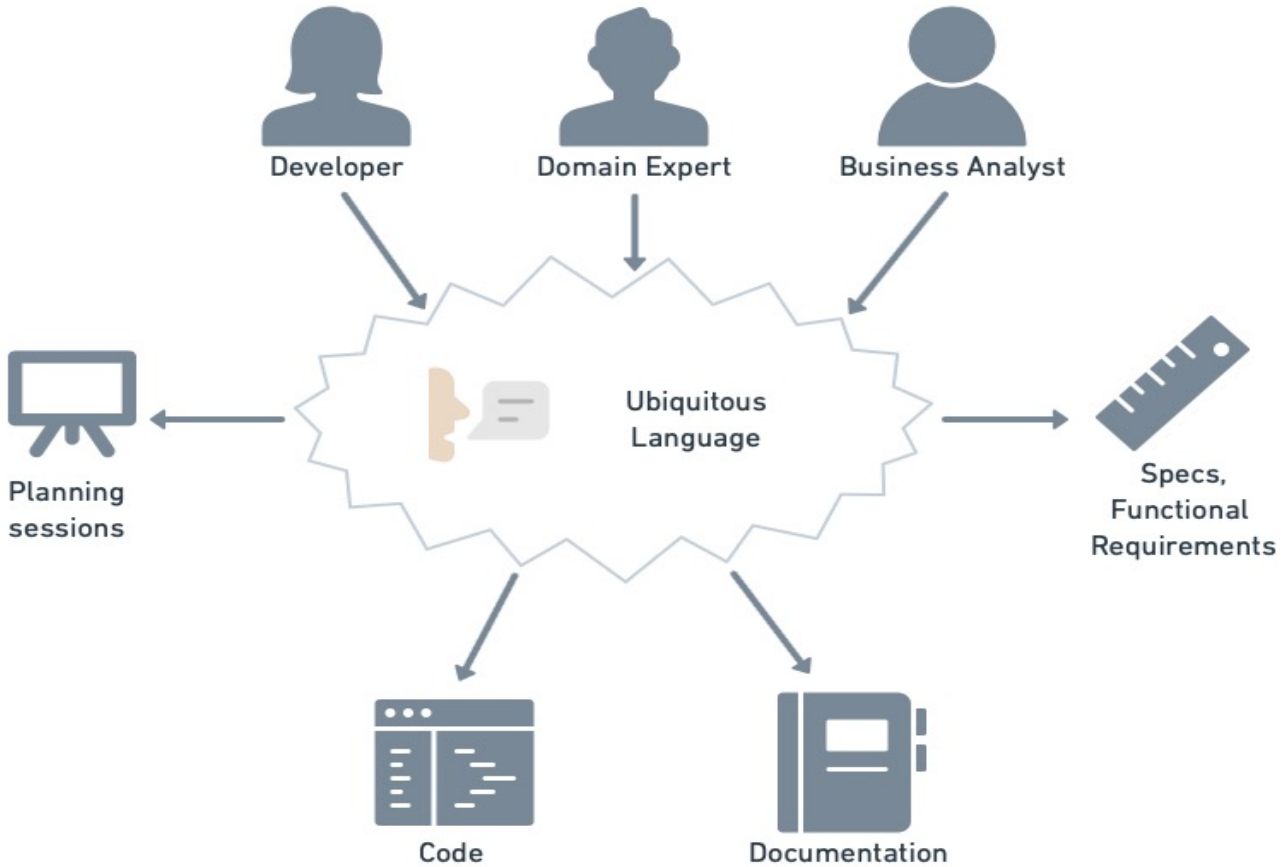


Figure: All Speak The Same Language "Ubiquitous Language"

—

—

—

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.
4. Resiliency
   - The application functionality is distributed across multiple services.
   - If a microservice fails, the functionality offered by the others continues to be available.

Why Microservices?

1. Unit design
   - The application consists of loosely coupled services.
   - Each service supports a single business task.
2. Flexibility
   - Each microservice can be developed using a programming language and framework that best suits.
3. Maintainability
   - Simple, focused, and independent. So the application is easier to maintain.
4. Resiliency
   - The application functionality is distributed across multiple services.
   - If a microservice fails, the functionality offered by the others continues to be available.
5. Scalability
   - Each microservice can be scaled independently of the other services.

Microservices vs Monolithic

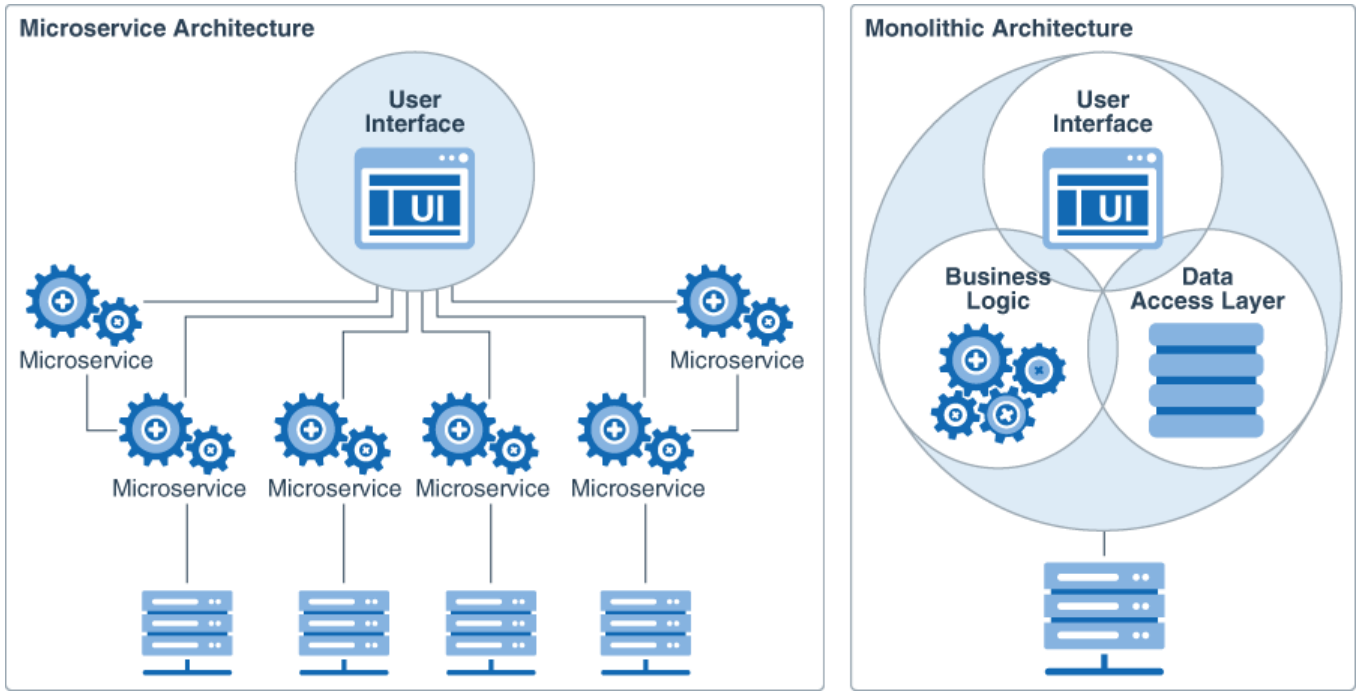


Figure: mivroservices vs monolithic

https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html

Closer Look

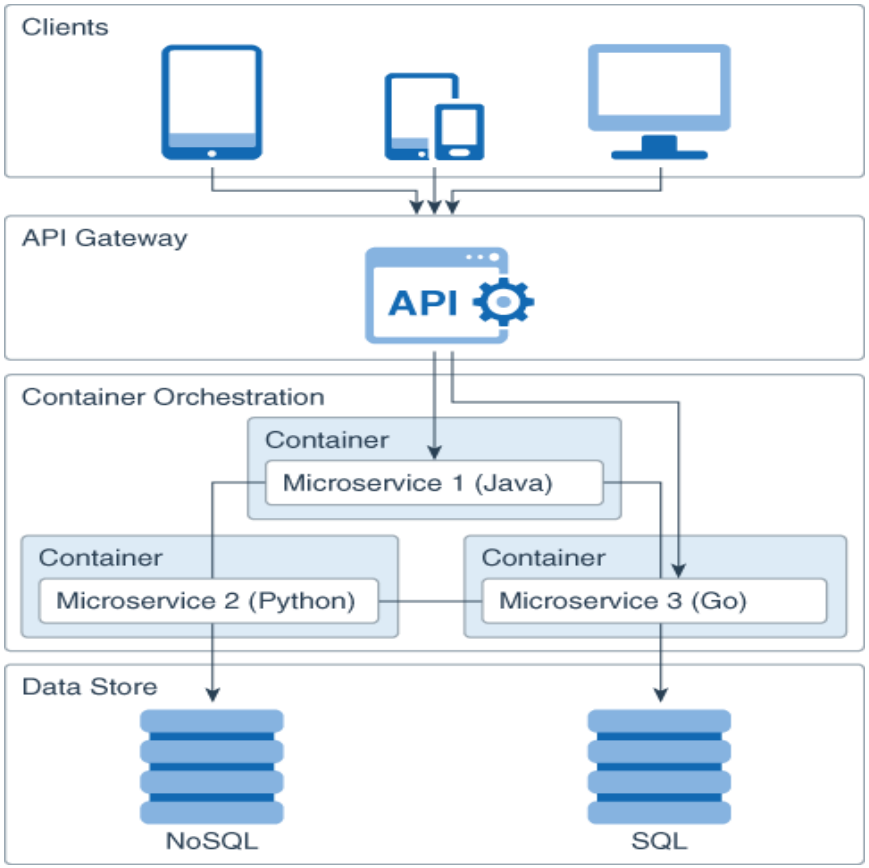


Figure: Microservices In Depth

https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html

Microservices Architecture

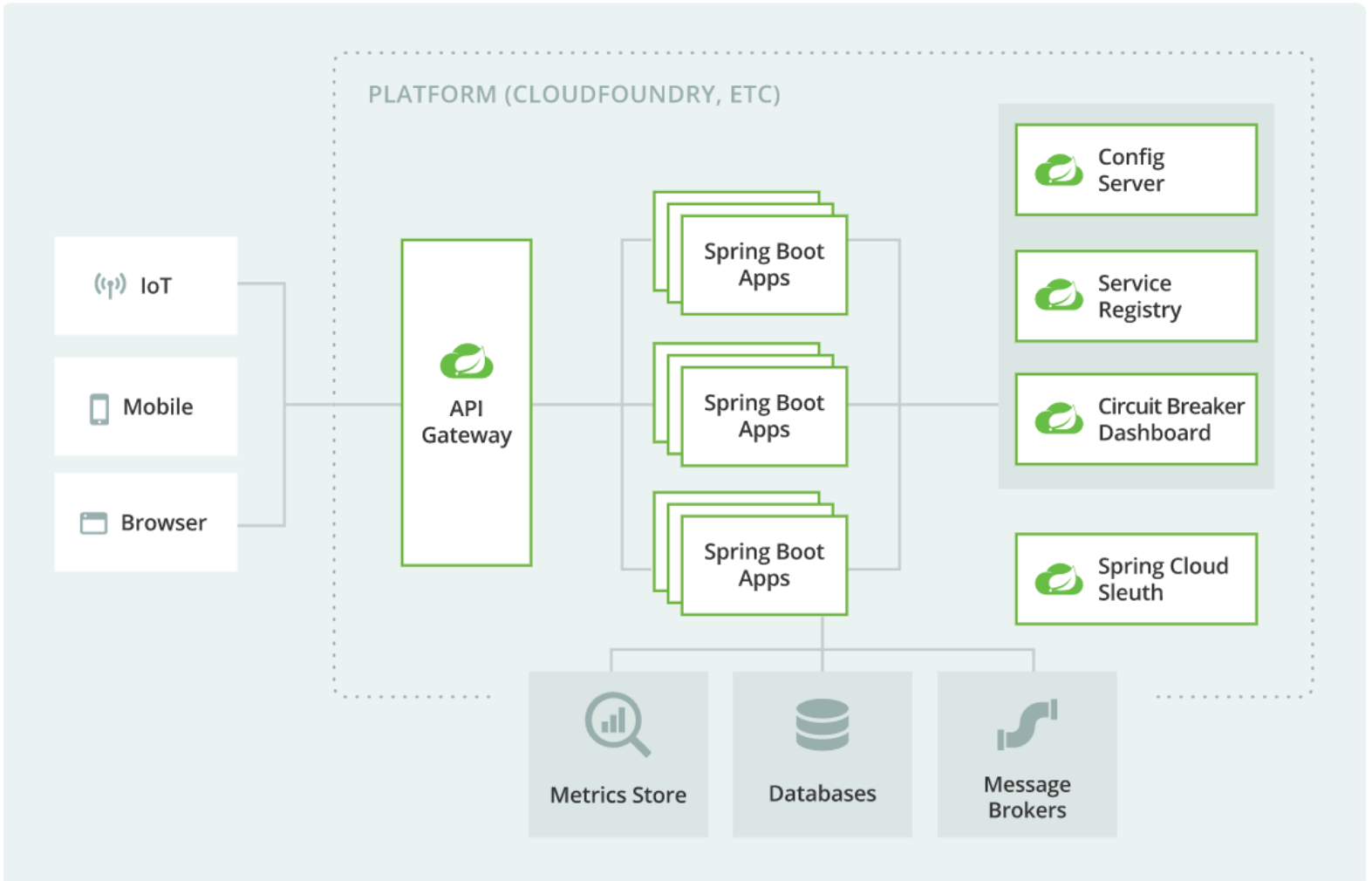
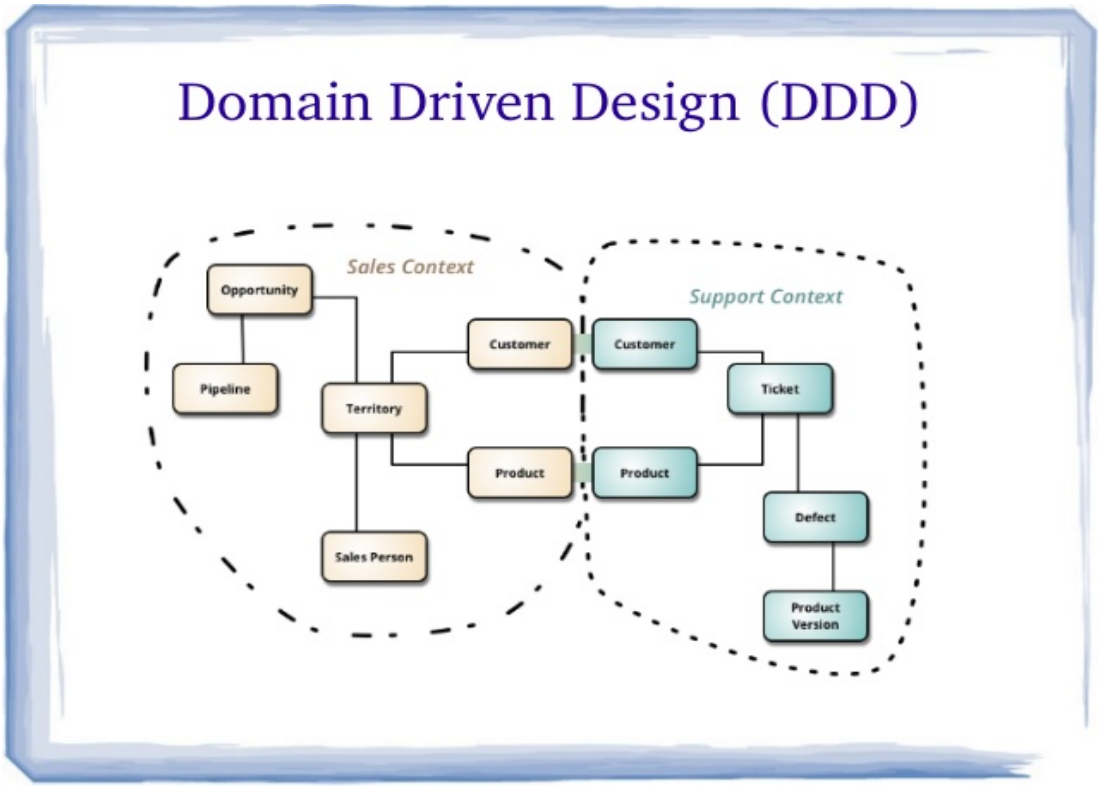


Figure: Microservices with Spring Cloud

Microservice characteristics

Single Responsibility

- Business Boundary
- Function Boundary



https://martinfowler.com/bliki/BoundedContext.html

Communication Design

HTTP communication

Also known as **Synchronous communication**, the calls between services is a viable option for **service-to-service** via REST API.

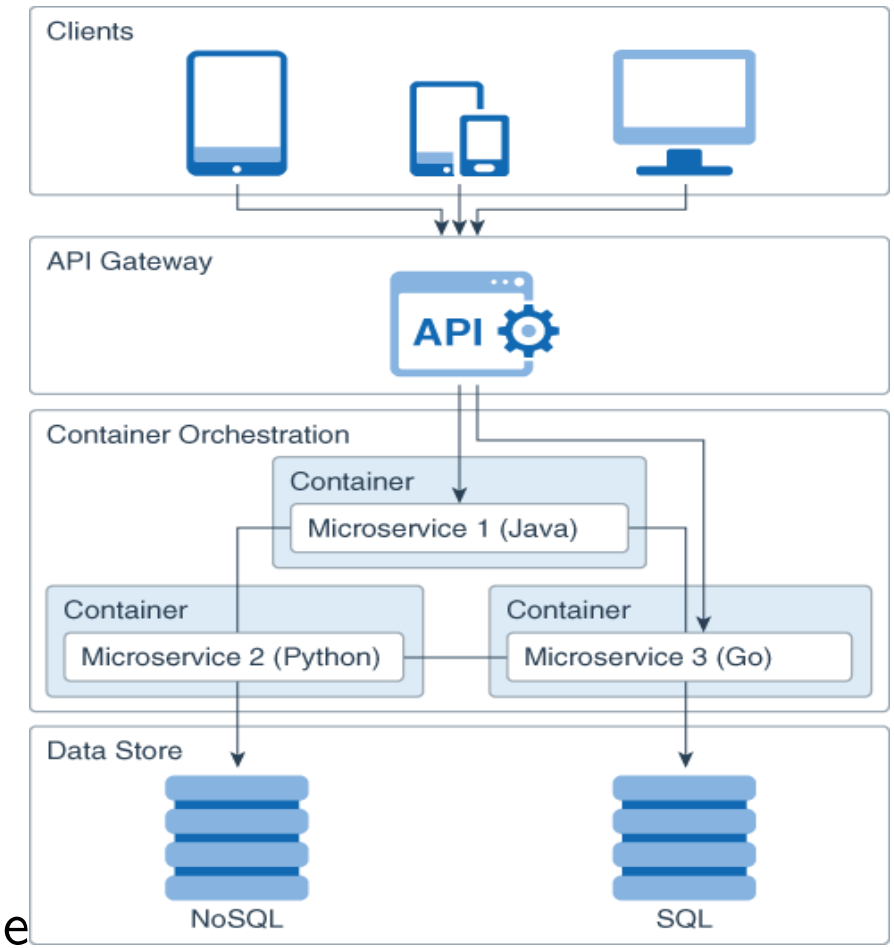
Message communication

Also known as **Asynchronous communication**, the services push messages to a message broker that other services subscribe to.

Event-driven communication

Another type of **Asynchronous communication**, the services does not need to know the common message structure. Communication between services takes place via events that individual services produce.

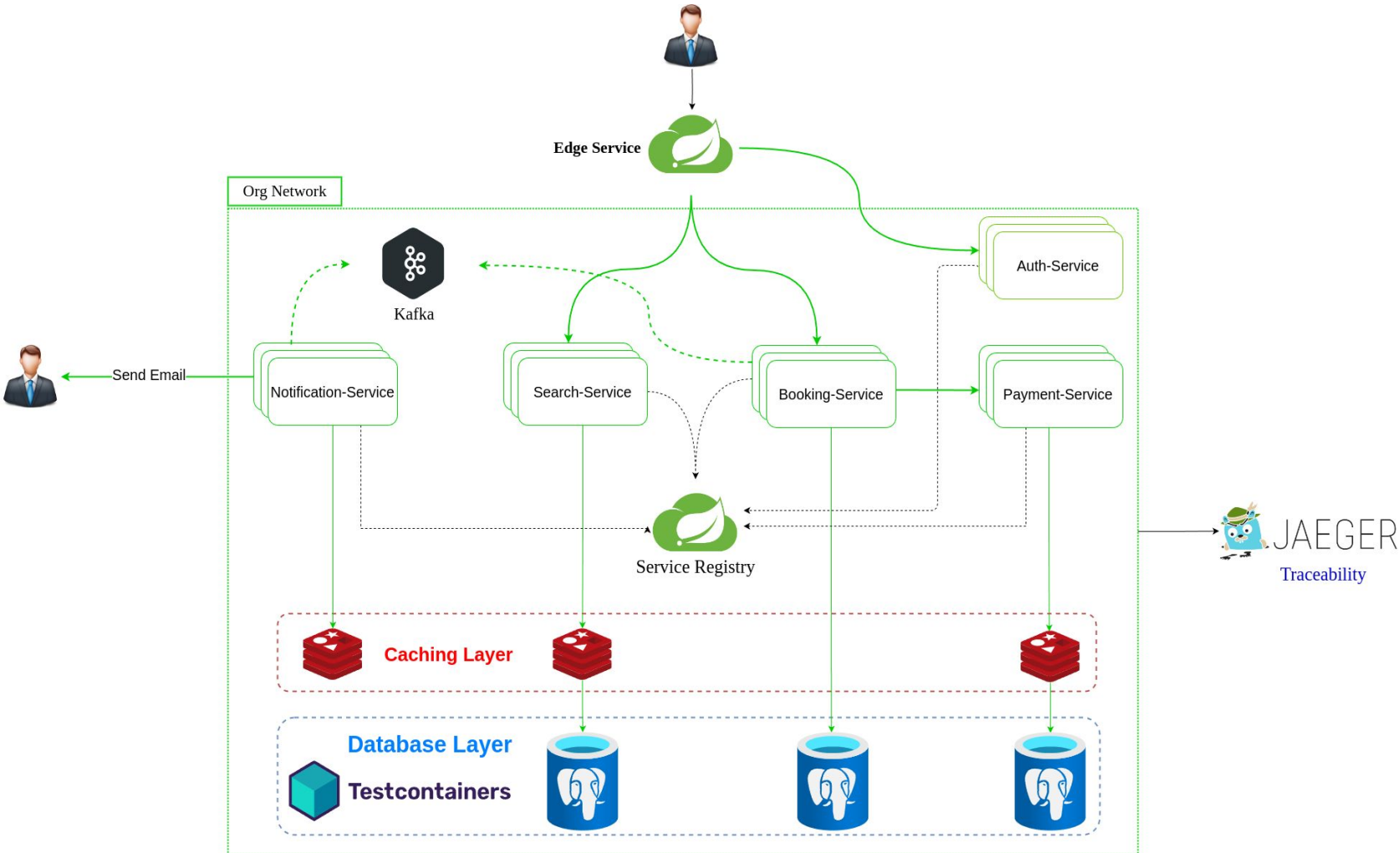
HTTP communication



scale

Figure: Service to Service Calls

Welcome to Flight System



Event-driven communication

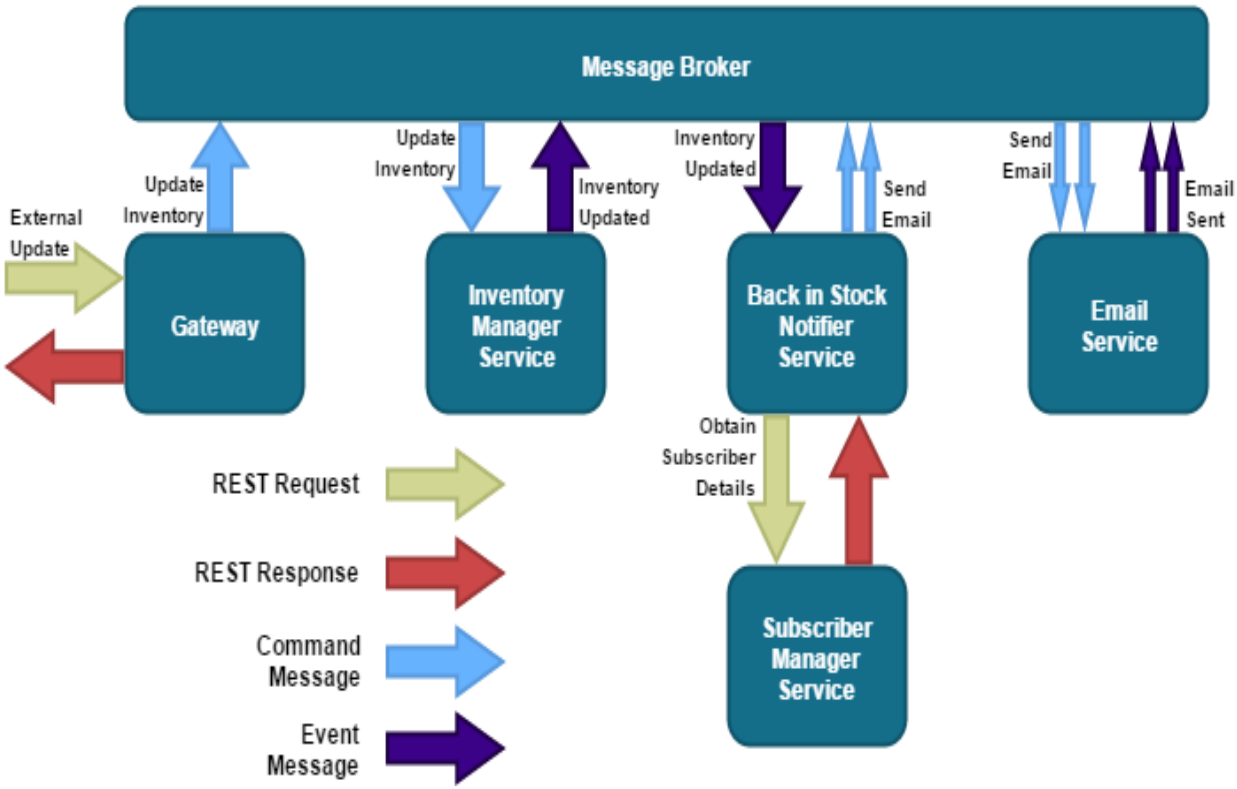


Figure: Asynchronous calls

https://capgemini.github.io/architecture/is-rest-best-microservices

Why not SOAP?

It is possible to build a microservices-based architecture using SOAP which uses HTTP. But:

- it only uses POST messages to transfer data to a server.
- SOAP lacks concepts such as HATEOAS that enable relationships between microservices to be handled flexibly.
- The interfaces have to be completely defined by the server and known on the client.

Microservices; Flexible Software Architecture. "Eberhard Wolff"

API Gateway

API Gateway

API Gateway is a tool that makes it easy for developers to create(1), publish(2), maintain(3), monitor(4), and secure(5) APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services.

Figure: Amazon Gateway

Orchestration and API Gateway cont...

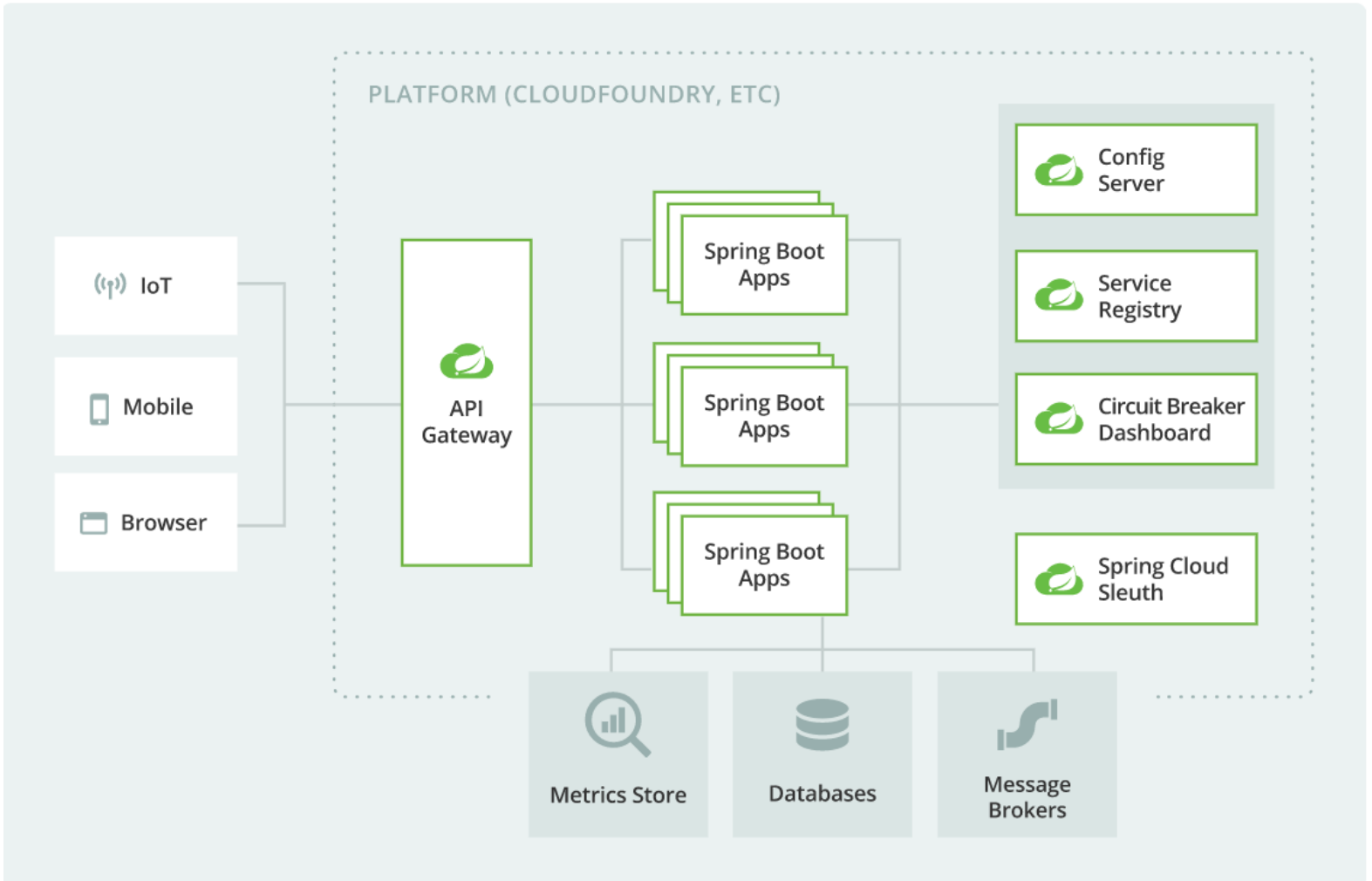


Figure: Microservices with Spring Cloud

Available Market Options

Figure: API Gateway Products

Service Discovery

Problem
In any distributed architecture, we need to find the physical address of where a machine is located.

Solution
Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.

Service Discovery

Problem
In any distributed architecture, we need to find the physical address of where a machine is located.

Solution
Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.
2. Resilient
   - client should "cache" service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.

Service Discovery

Problem
In any distributed architecture, we need to find the physical address of where a machine is located.

Solution
Using service discovery, a service can register itself when it is up and healthy. By using such technology you can achieve:

1. Load balanced
   - dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.
2. Resilient
   - client should "cache" service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.
3. Fault-tolerant
   - detect when a service instance isn't healthy and remove the instance from the list of available services.

Service Discovery with Gateway

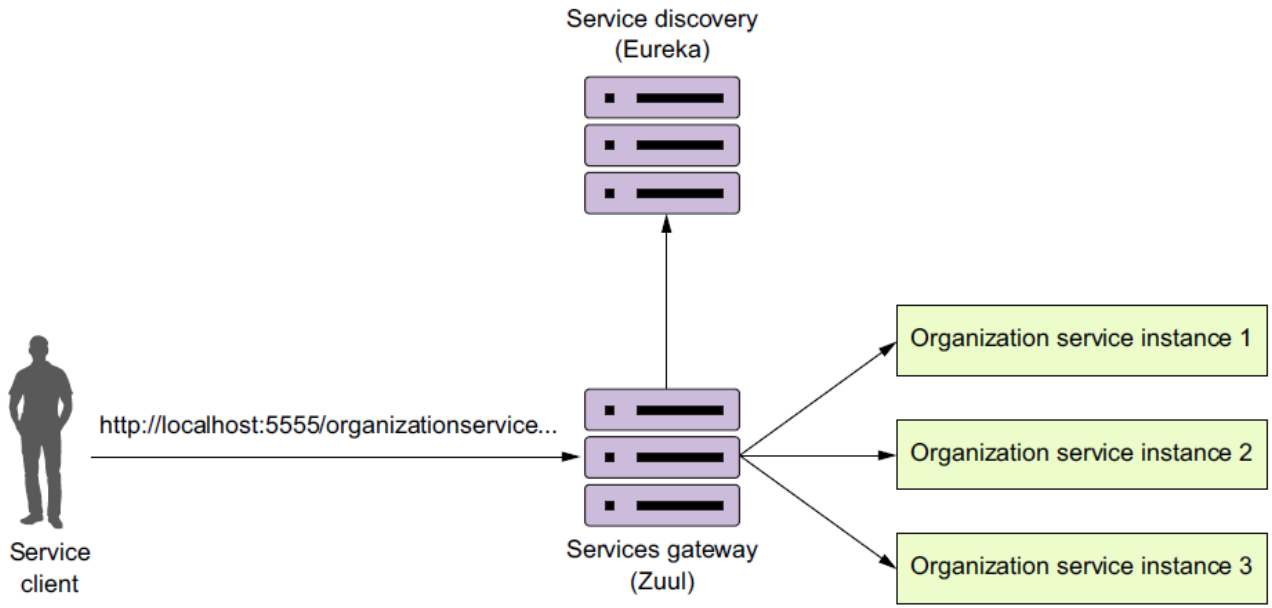


Figure: Service Registry and Gateway

Available Market Options

Figure: Service Registry Products

Externalized and Dynamic Configurations

Problem

Configurations will vary from environment to another, How to manage them?

Solution

Centralize your configuration

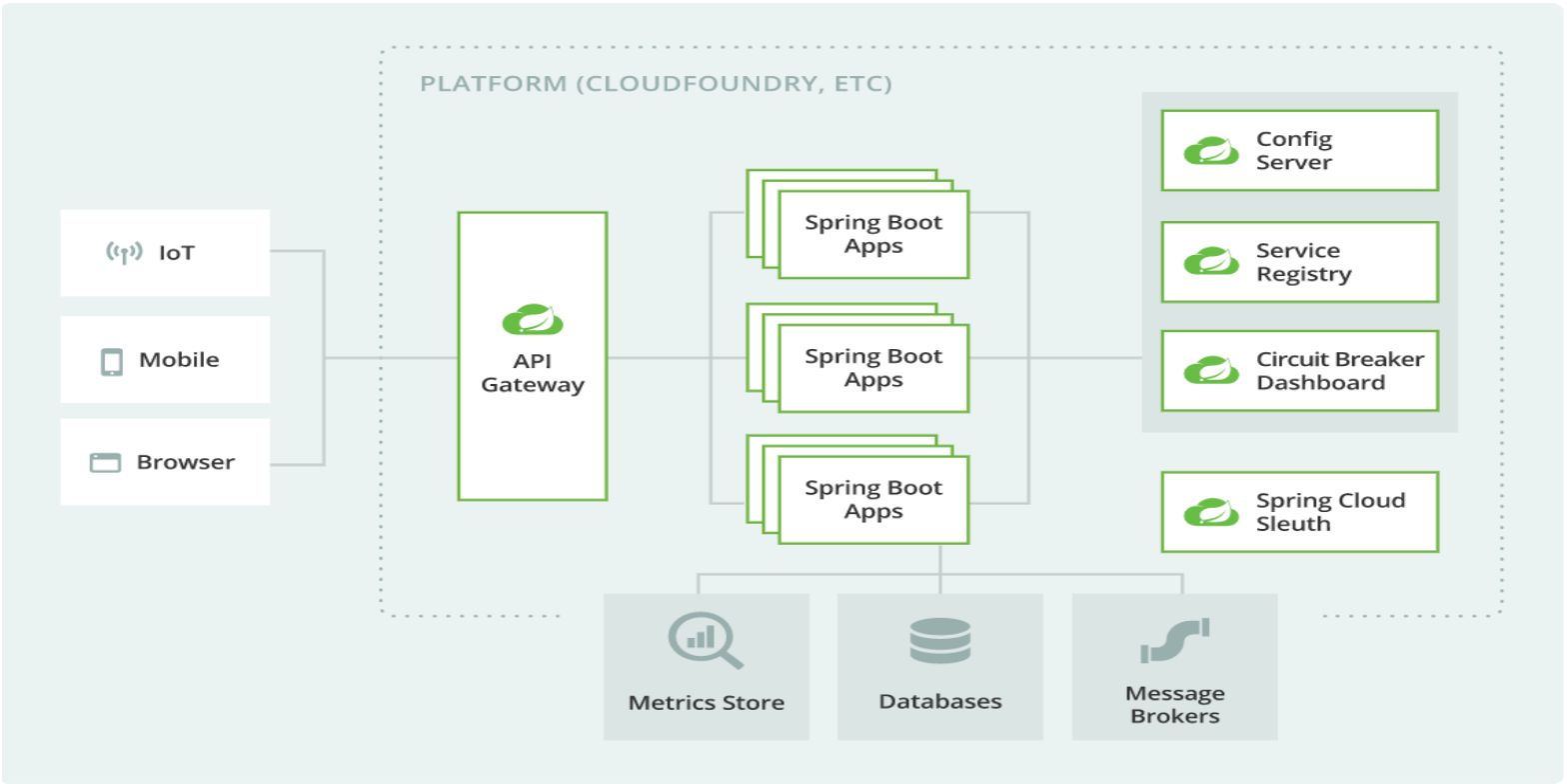


Figure: Microservices with Spring Cloud

Available Market Options

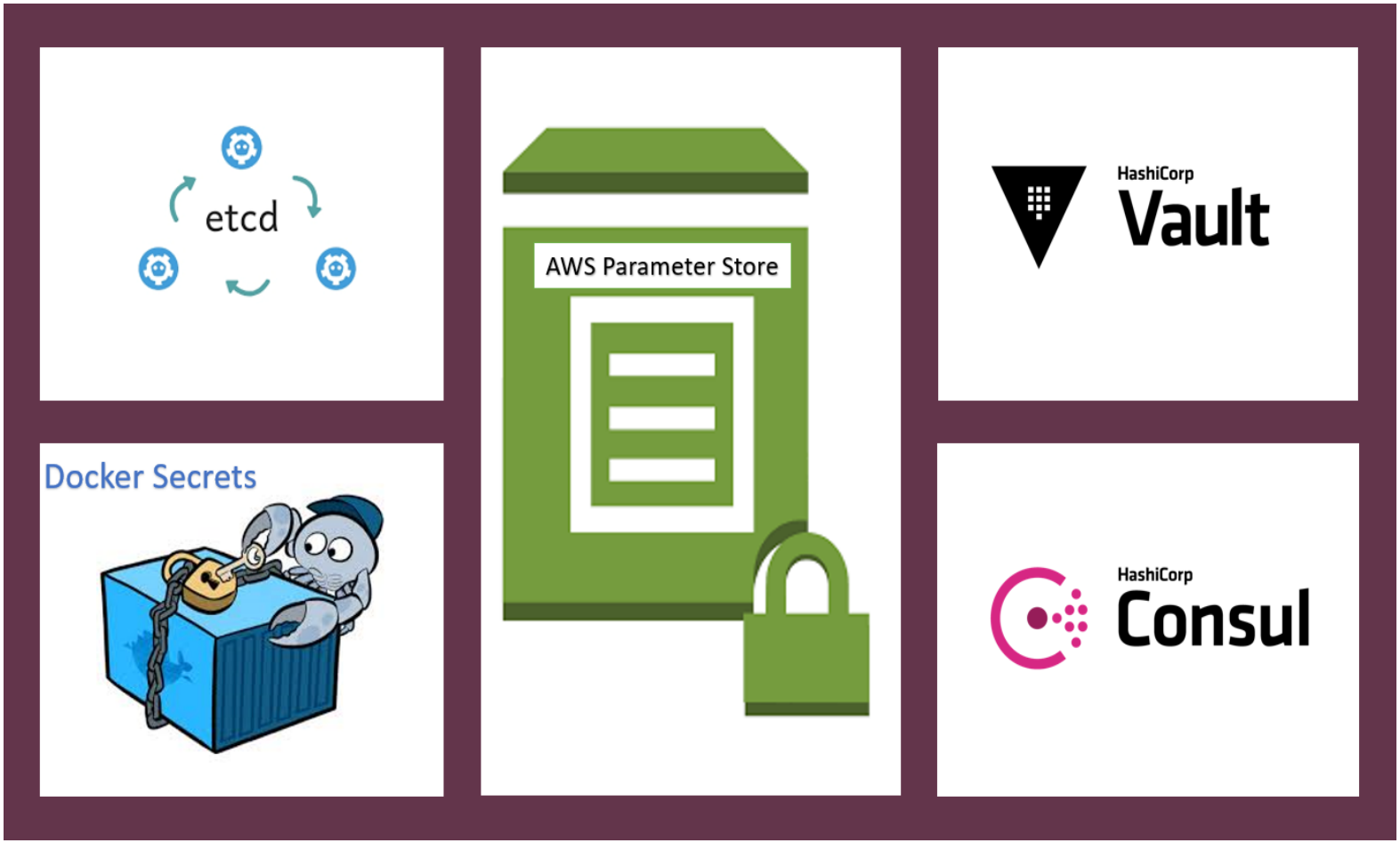


Figure: Popular Config Stores

Circuit Breaker Pattern

Problem

- One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached.

Solution

Fault Tolerance

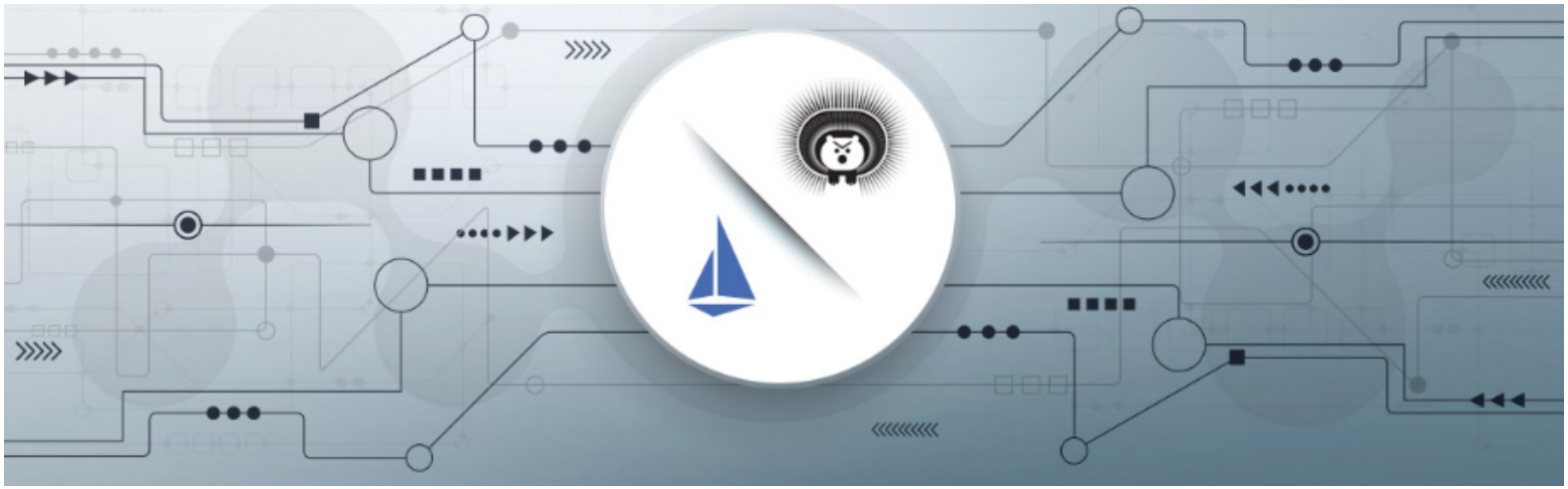


Figure: Circuit Breaker

Circuit Breaker Pattern

Problem

- One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached.
- What's worse if you have many callers on a unresponsive supplier, then you can run out of critical resources leading to cascading failures across multiple systems.

Solution

Fault Tolerance

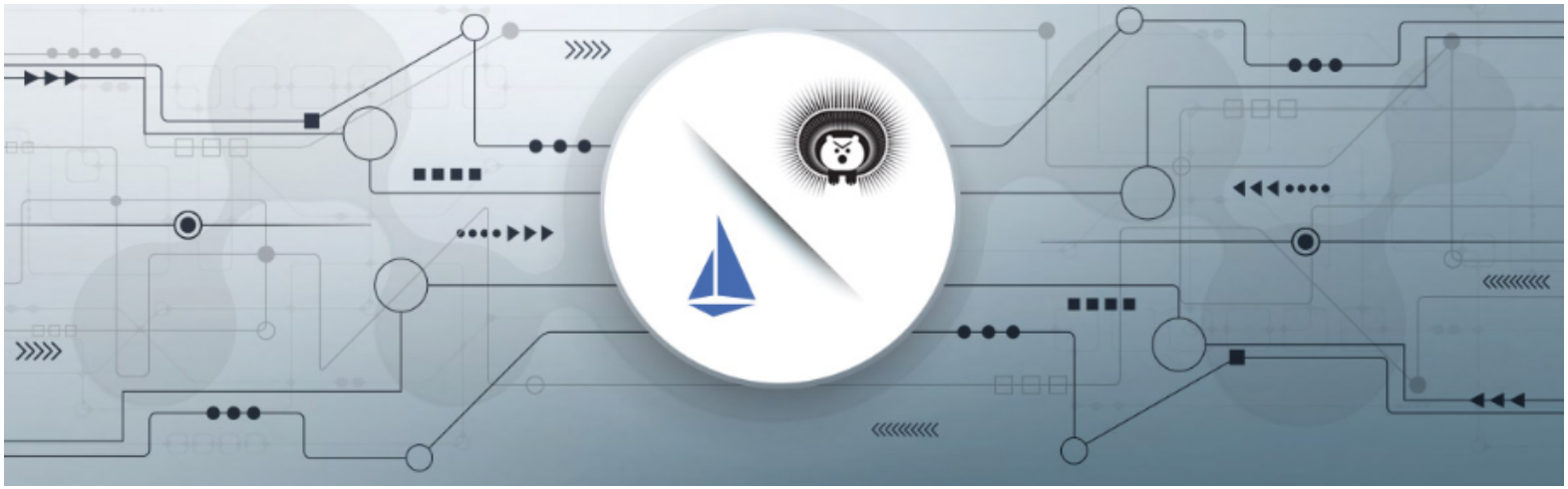


Figure: Circuit Breaker

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

- Placing services on nodes.

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.
- Service discovery

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.
- Service discovery
- Scaling the number of instances of a service

Service Orchestrators

An orchestrator handles tasks of deploying and managing a set of services. With orchestrator you can

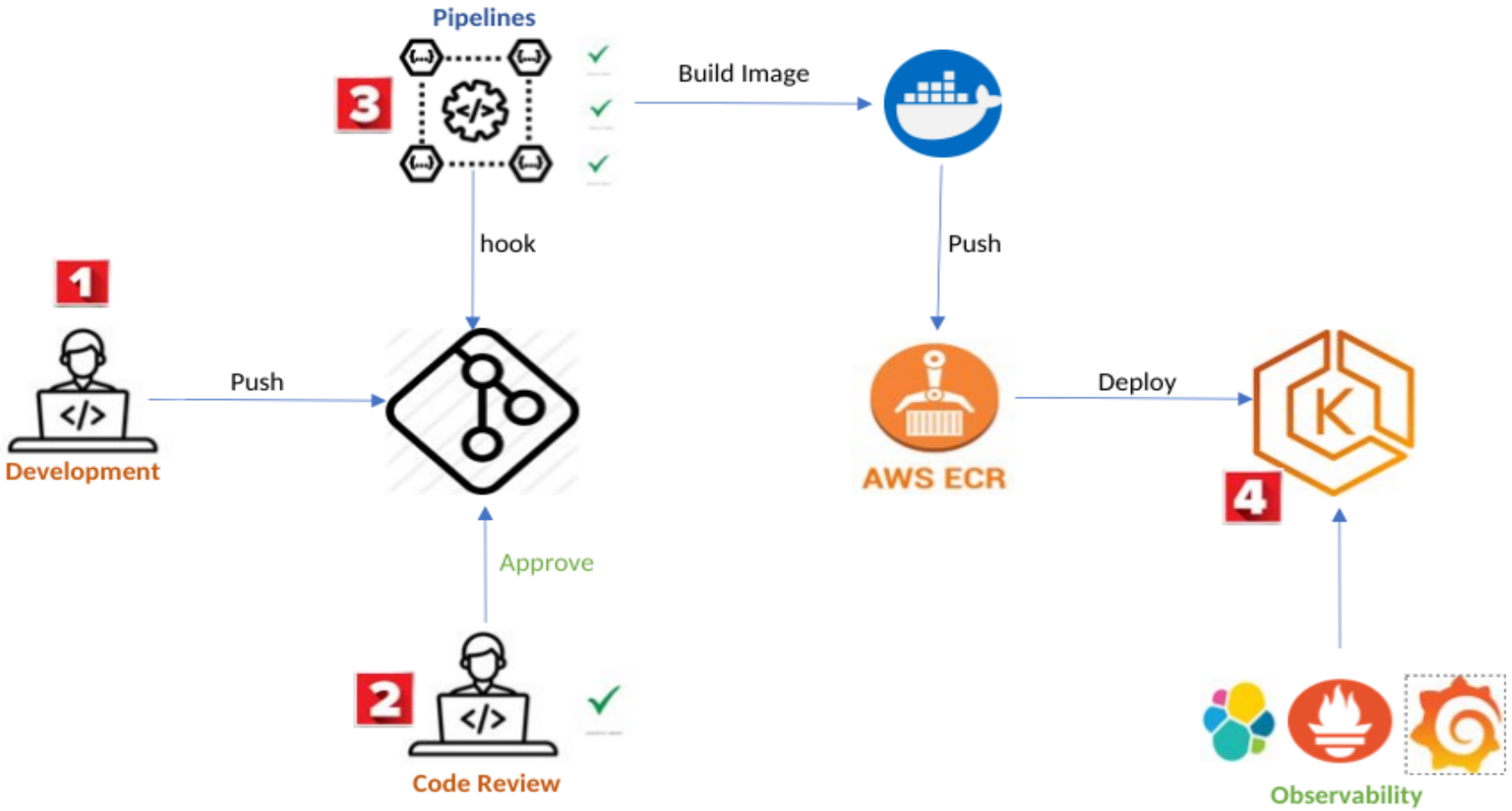
- Placing services on nodes.
- Monitoring the health of services and restarting unhealthy services.
- Load balancing network traffic across service instances.
- Service discovery
- Scaling the number of instances of a service

Popular Orchestrators

- Docker Swarm
- Kubernates
- AWS ECS
- Service Fabric
- Openshift

Real Example

Continuous Integration, and Deployment are your friends



Infrastructure as Code "**IaC**"

What is Infrastructure?

Infrastructure as Code "**IaC**"

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Infrastructure as Code **"IaC"**

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

Infrastructure as Code "**IaC**"

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job

Infrastructure as Code "**IaC**"

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job
- Everything should be automated and following pipelines

Infrastructure as Code "**IaC**"

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job
- Everything should be automated and following pipelines
- It is better to use **Declarative** way on building such pipelines

Infrastructure as Code **"IaC"**

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job
- Everything should be automated and following pipelines
- It is better to use **Declarative** way on building such pipelines
- Writing declarative files (JSON, YAML or XML) can be reviewed, enhanced and apply versioning on it like normal development

Infrastructure as Code "**IaC**"

What is Infrastructure?

The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job
- Everything should be automated and following pipelines
- It is better to use **Declarative** way on building such pipelines
- Writing declarative files (JSON, YAML or XML) can be reviewed, enhanced and apply versioning on it like normal development
- IaC should be always **Idempotent** this will enable Ops team to test their work on production-like environment before actual production

Infrastructure as Code "**IaC**"

What is Infrastructure?

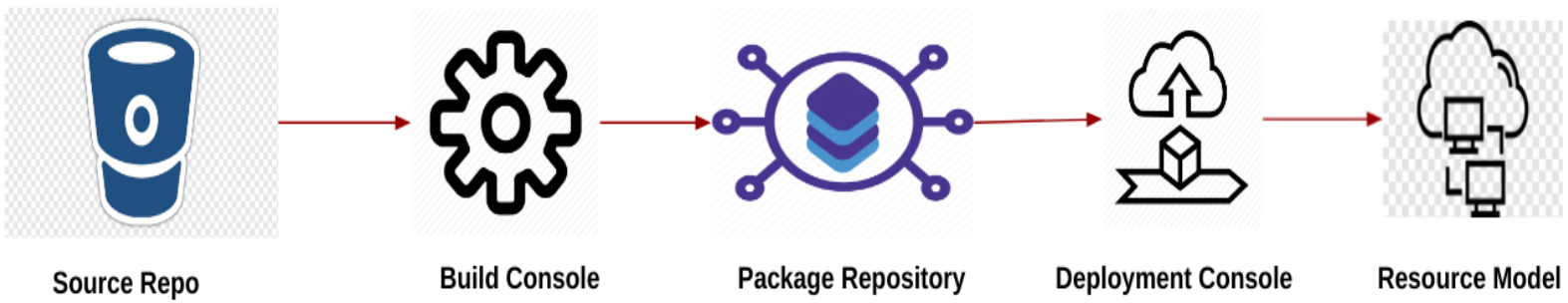
The components required to operate and manage IT environments like hardware, software, networking components, an operating system, etc.

Problem Statement

Building and running servers manually using scripts or UI consoles is headache, and usually lead to configuration drift, and unexpected errors that require extra time which might not be planned!

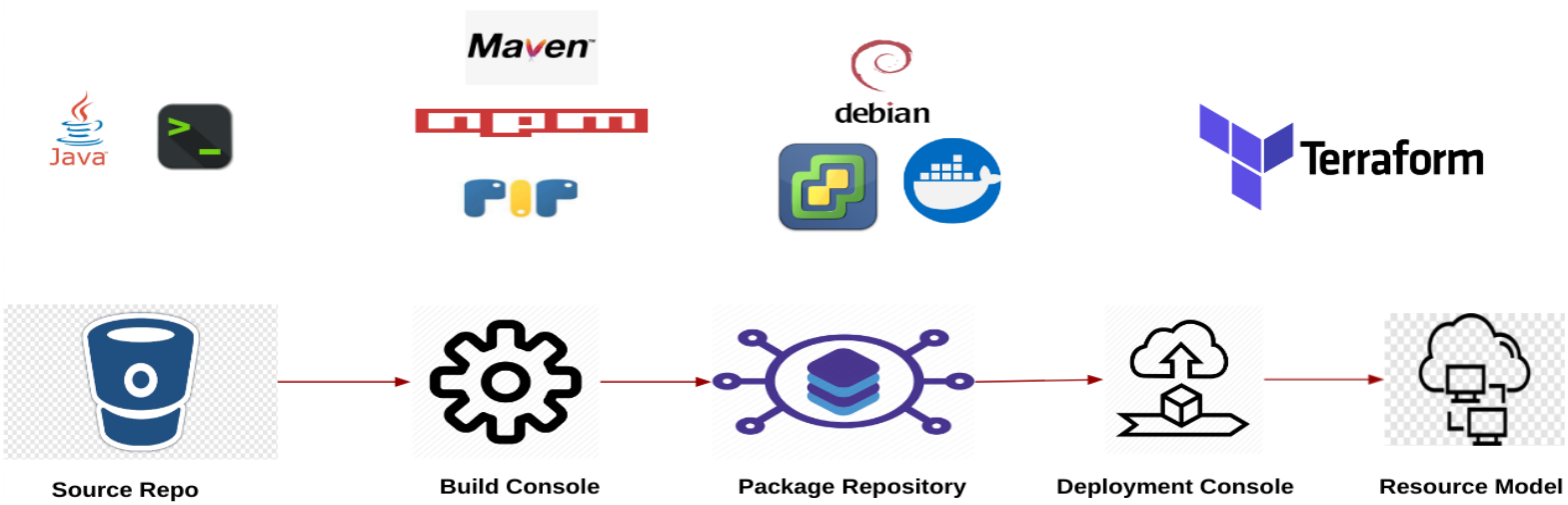
- The reason is Operator/SysAdmin usually uses **Imperative** way on building his job
- Everything should be automated and following pipelines
- It is better to use **Declarative** way on building such pipelines
- Writing declarative files (JSON, YAML or XML) can be reviewed, enhanced and apply versioning on it like normal development
- IaC should be always **Idempotent** this will enable Ops team to test their work on production-like environment before actual production
- Should I (SysAdmin/Operator) learn programming to apply IaC?

Infrastructure as Code "**IaC**"



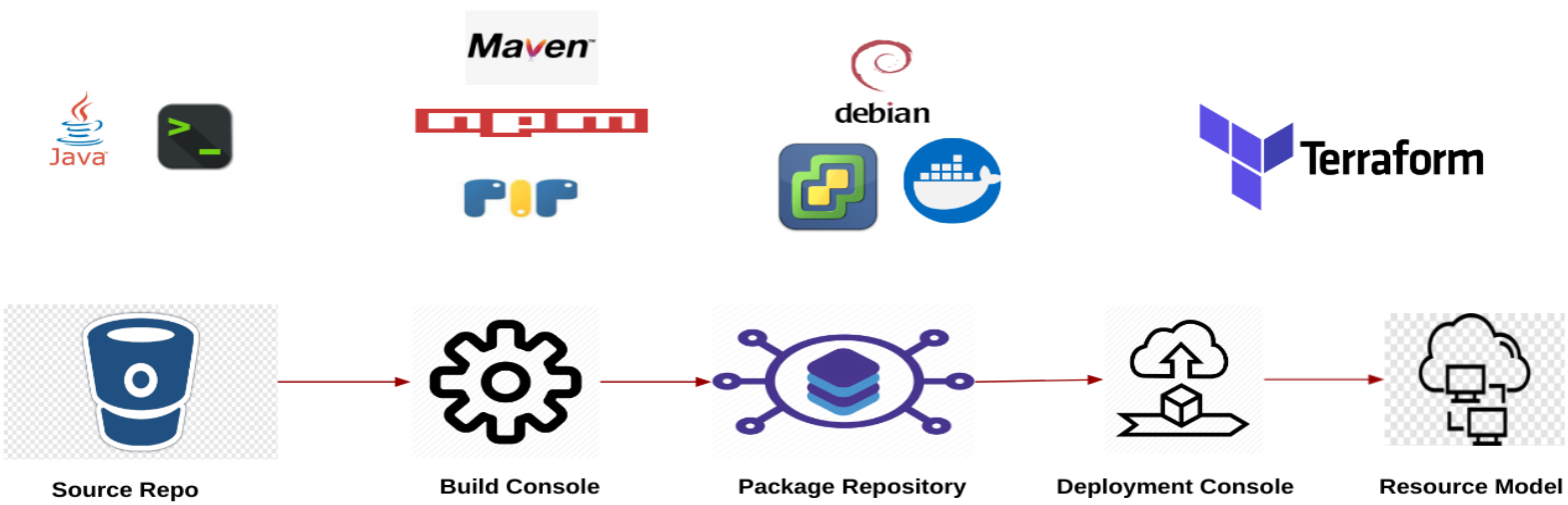
Infrastructure as Code "**IaC**"

- **But how to ensure quality of code?**



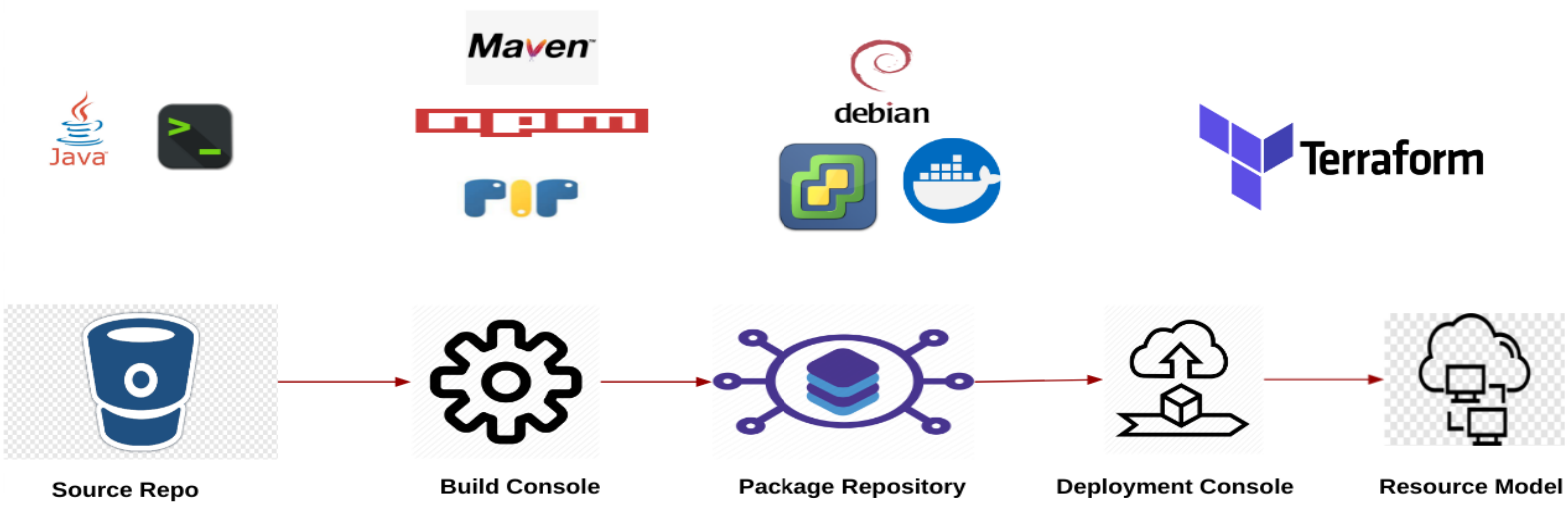
Infrastructure as Code "**IaC**"

- **But how to ensure quality of code?**
- Answer: **Testing**



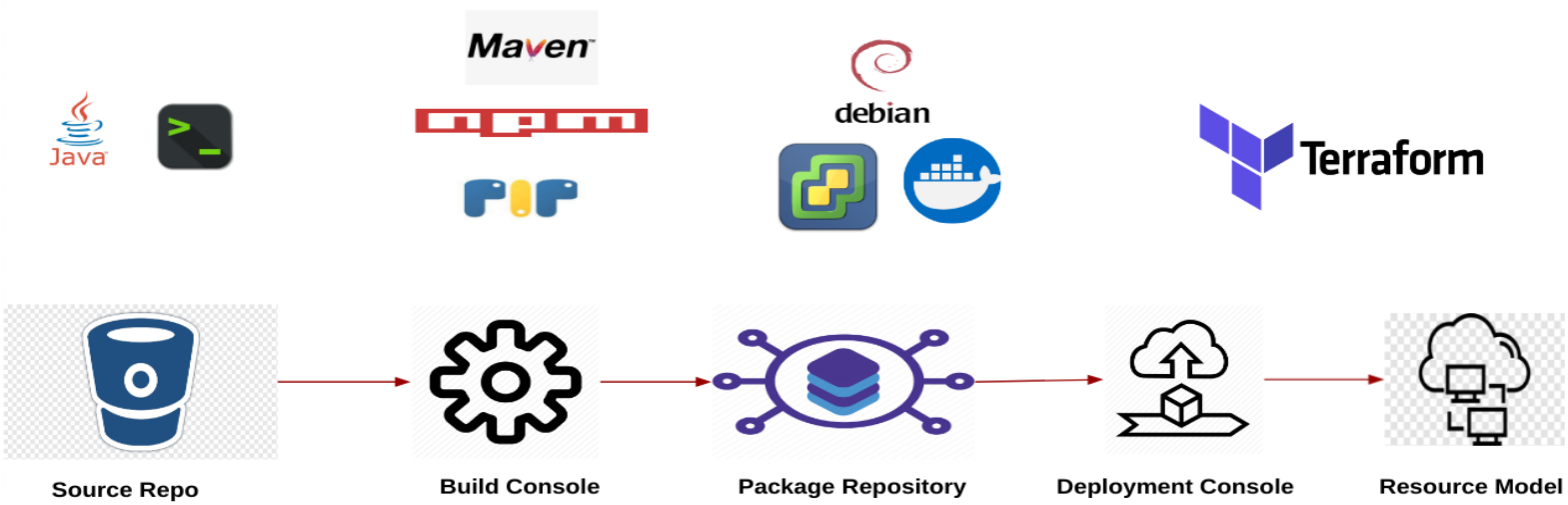
Infrastructure as Code "**IaC**"

- **But how to ensure quality of code?**
- Answer: **Testing**
- Techniques used in normal development should be applied, e.g Unit & Integration Test.
  - **Rubocop** Formatter and **Foodcritic** Linter
  - **ChefSpec** Test Simulator and Coverage
  - **ServerSpec** and **TestInfra** for Integration testing



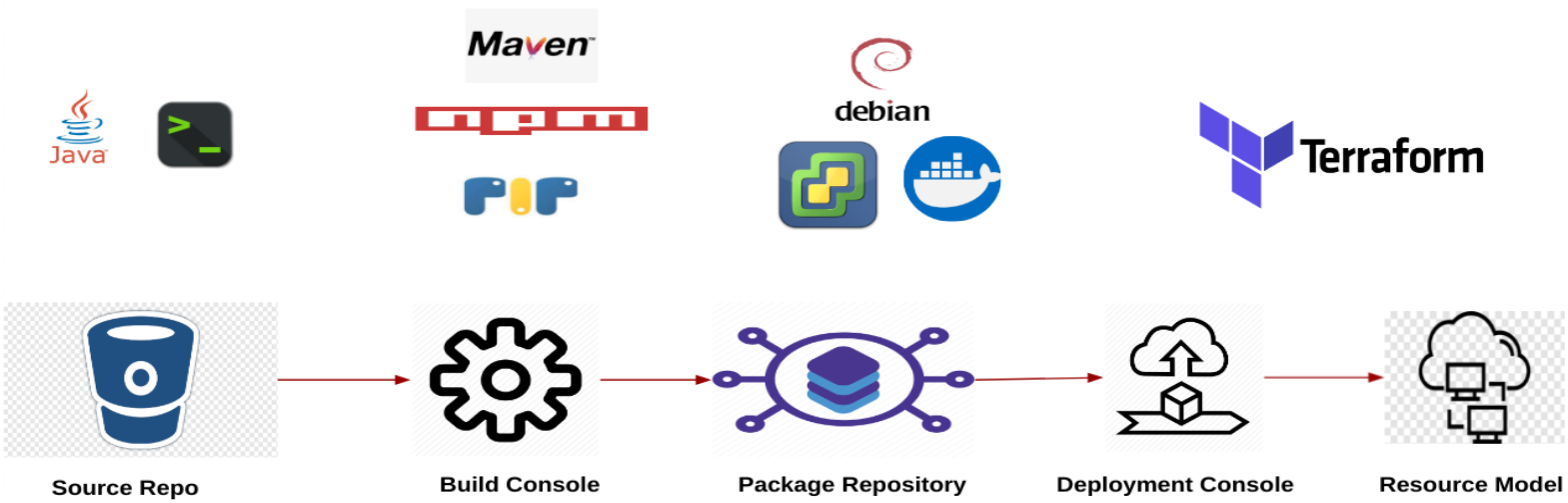
Infrastructure as Code "**IaC**"

- **But how to ensure quality of code?**
- Answer: **Testing**
- Techniques used in normal development should be applied, e.g Unit & Integration Test.
  - **Rubocop** Formatter and **Foodcritic** Linter
  - **ChefSpec** Test Simulator and Coverage
  - **ServerSpec** and **TestInfra** for Integration testing
- The output of Build phase is the deliverable (Binaries), once it is created, should never be changed; this is called **Immutable Deployment**



Infrastructure as Code "**IaC**"

- **But how to ensure quality of code?**
- Answer: **Testing**
- Techniques used in normal development should be applied, e.g Unit & Integration Test.
  - **Rubocop** Formatter and **Foodcritic** Linter
  - **ChefSpec** Test Simulator and Coverage
  - **ServerSpec** and **TestInfra** for Integration testing
- The output of Build phase is the deliverable (Binaries), once it is created, should never be changed; this is called **Immutable Deployment**
- There will be always kind of tension between Dev team and Ops team; especially if Ops team is applying SRE principles



Communication Design

HTTP communication

Also known as **Synchronous communication**, the calls between services is a viable option for **service-to-service** via REST API.

Message communication

Also known as **Asynchronous communication**, the services push messages to a message broker that other services subscribe to.

Event-driven communication

Another type of **Asynchronous communication**, the services does not need to know the common message structure. Communication between services takes place via events that individual services produce.

Communication Design - Async & Messaging

Online purchase example

Communication Design - Async & Messaging

Online purchase example

- Save it

Communication Design - Async & Messaging

Online purchase example

- Save it
- Check Duplicates

Communication Design - Async & Messaging

Online purchase example

- Save it
- Check Duplicates
- Charge Customer

Communication Design - Async & Messaging

Online purchase example

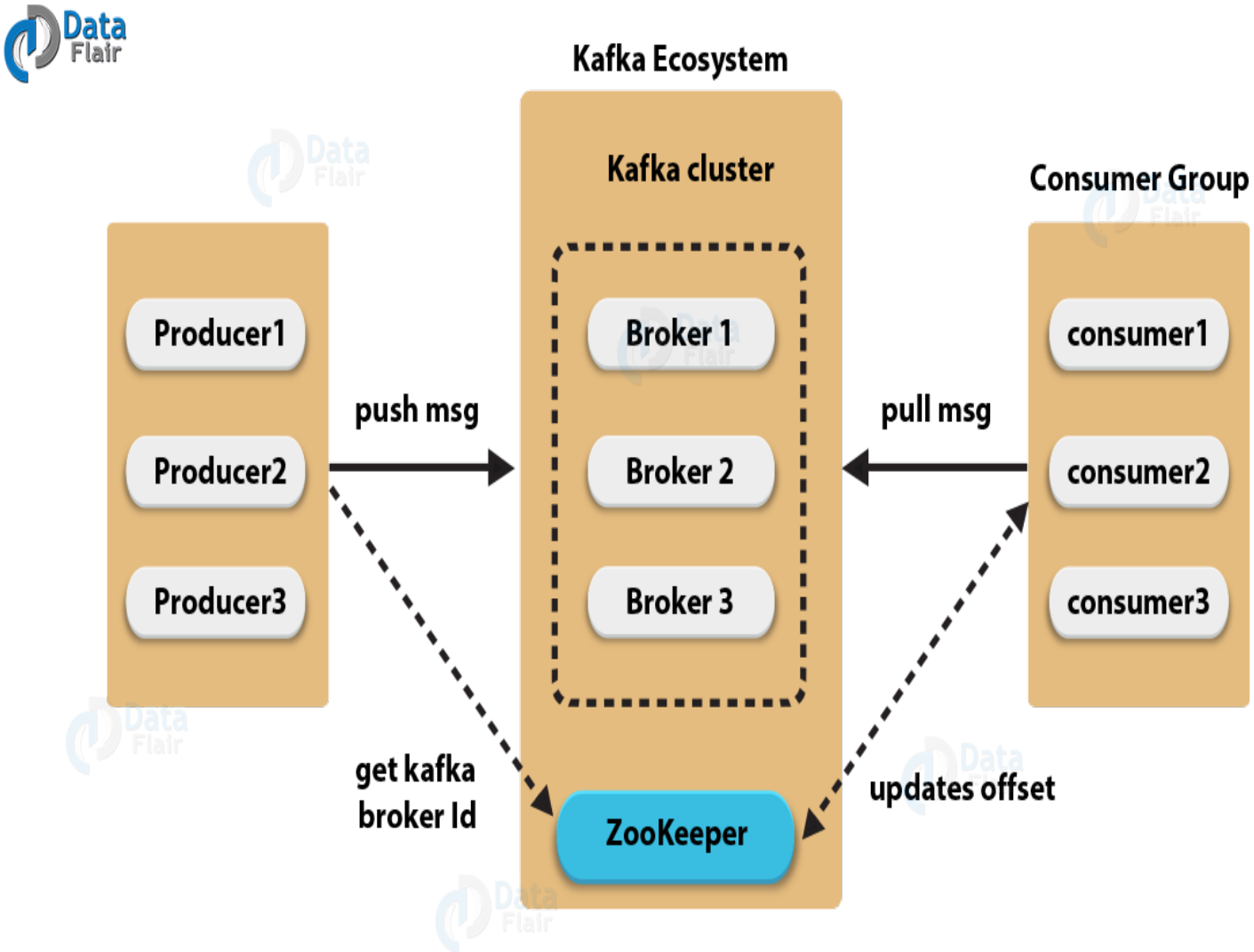
- Save it
- Check Duplicates
- Charge Customer
- Send Email to customer

Communication Design - Async & Messaging

Online purchase example

- Save it
- Check Duplicates
- Charge Customer
- Send Email to customer
- Audit Order details

Communication Design - Async & Messaging



Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity
  - Good choice for distributed systems

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity
  - Good choice for distributed systems
  - Cons: **Network Latency and Management Complexity**

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity
  - Good choice for distributed systems
  - Cons: **Network Latency and Management Complexity**
- High Availability

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

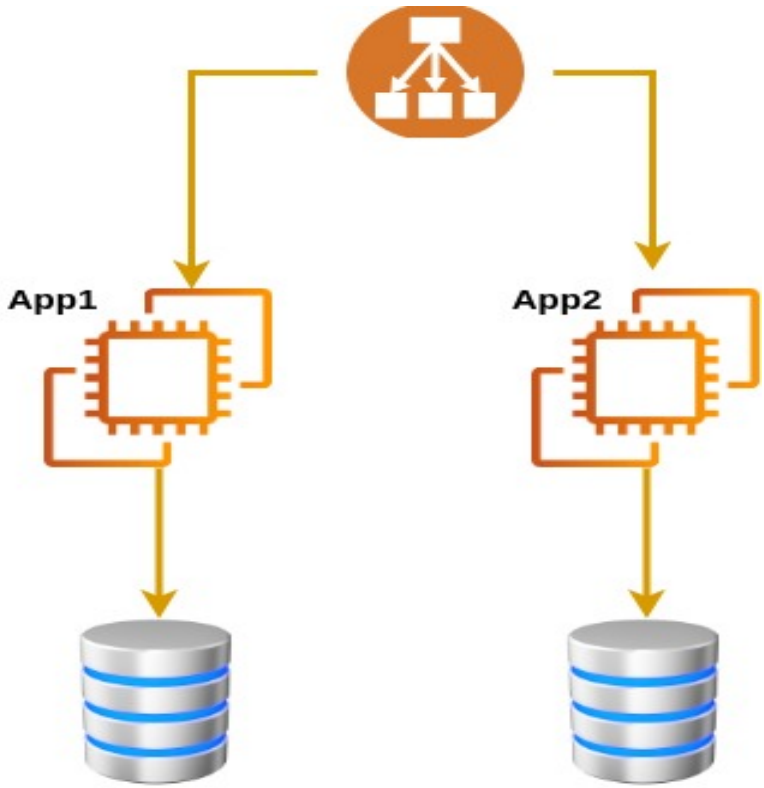
- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity
  - Good choice for distributed systems
  - Cons: **Network Latency and Management Complexity**
- High Availability
  - The availablity percentage of your service within period of time

Scalability and High Availability

Scalability is about your system capability to scale (In or Out) in order to handle workload and throughput

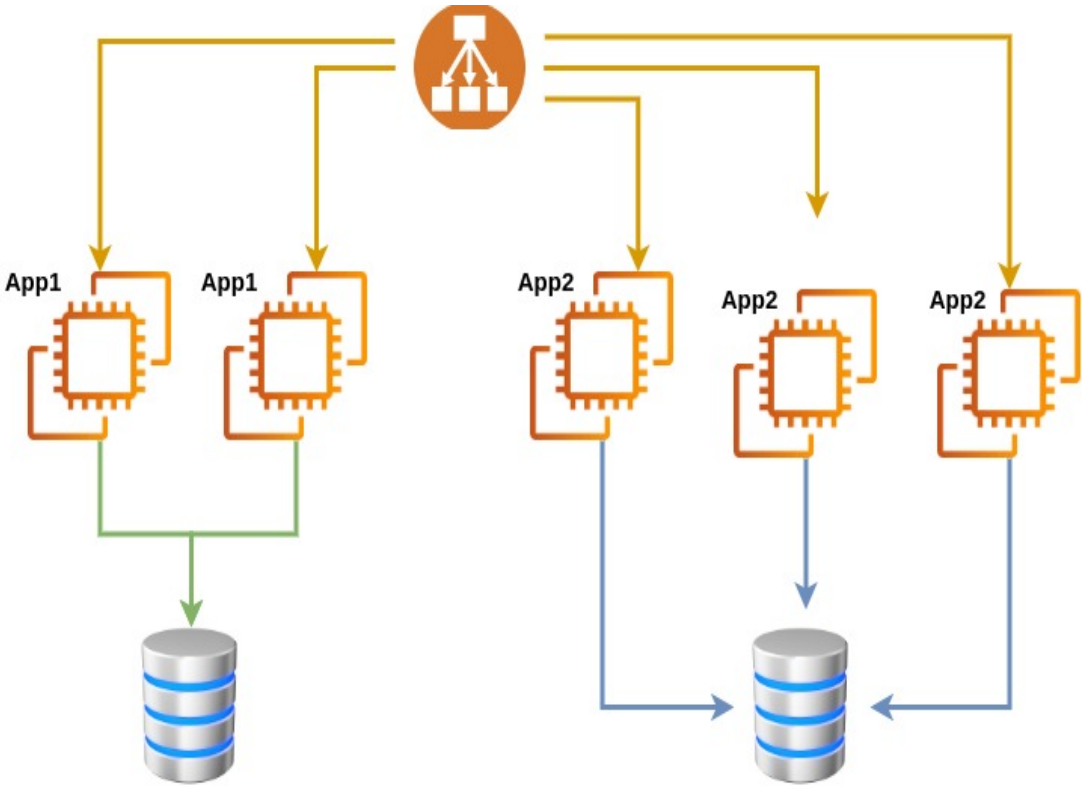
- Vertical Scaling
  - Database scaling for example by adding more CPU and Memory power
  - Cons: **Can't work in all cases, expensive**
- Horizontal Scaling
  - Increase numbers of workers, with same power capacity
  - Good choice for distributed systems
  - Cons: **Network Latency and Management Complexity**
- High Availability
  - The availablity percentage of your service within period of time
  - SLA is 99.99%

Scalability and High Availability



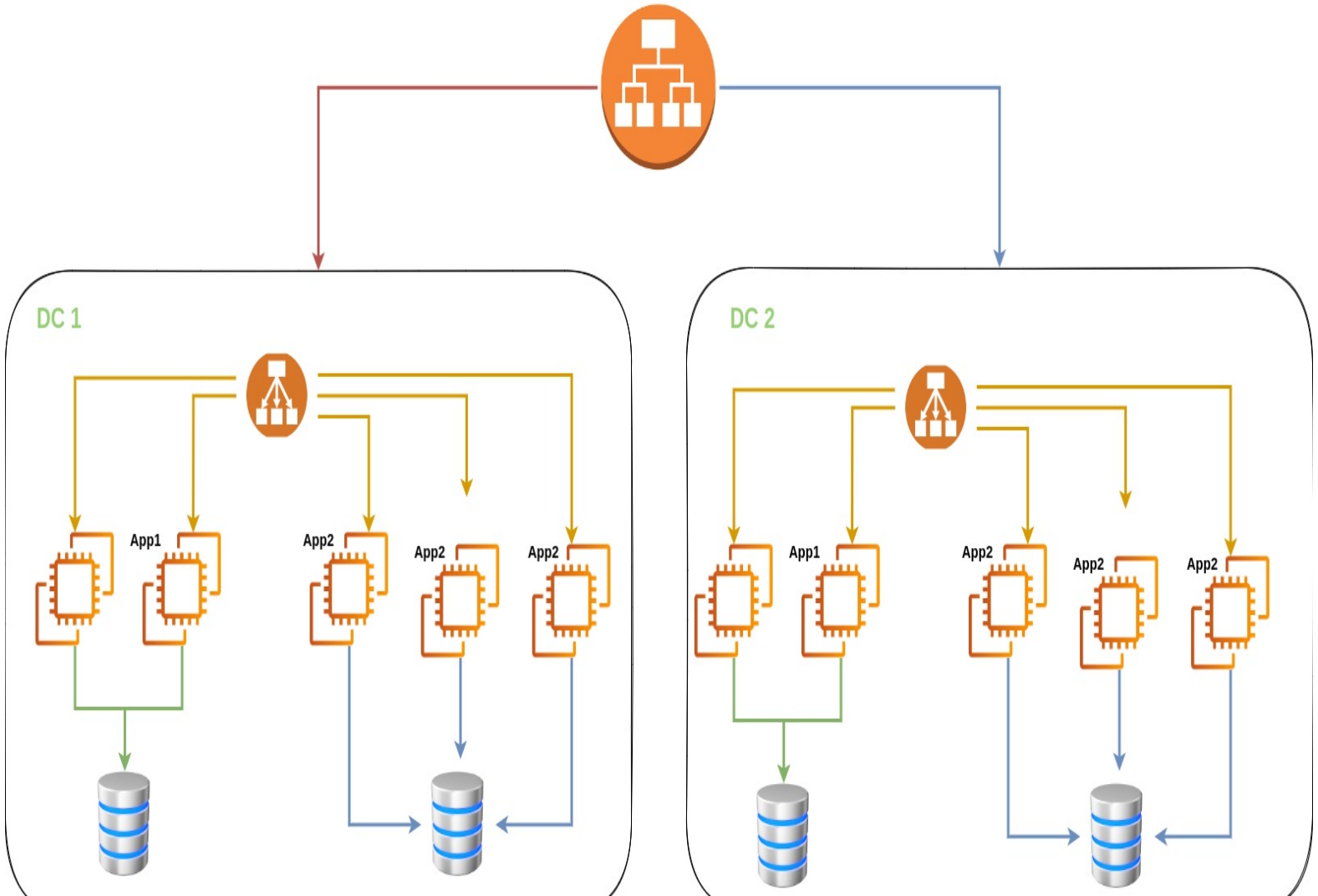
Scalability and High Availability

Horizontal Scaling



Scalability and High Availability

Fault Tolerant



References


Eric Evans, (2003)
Domain-Driven Design: Tackling Complexity in the Heart of Software

Thank You