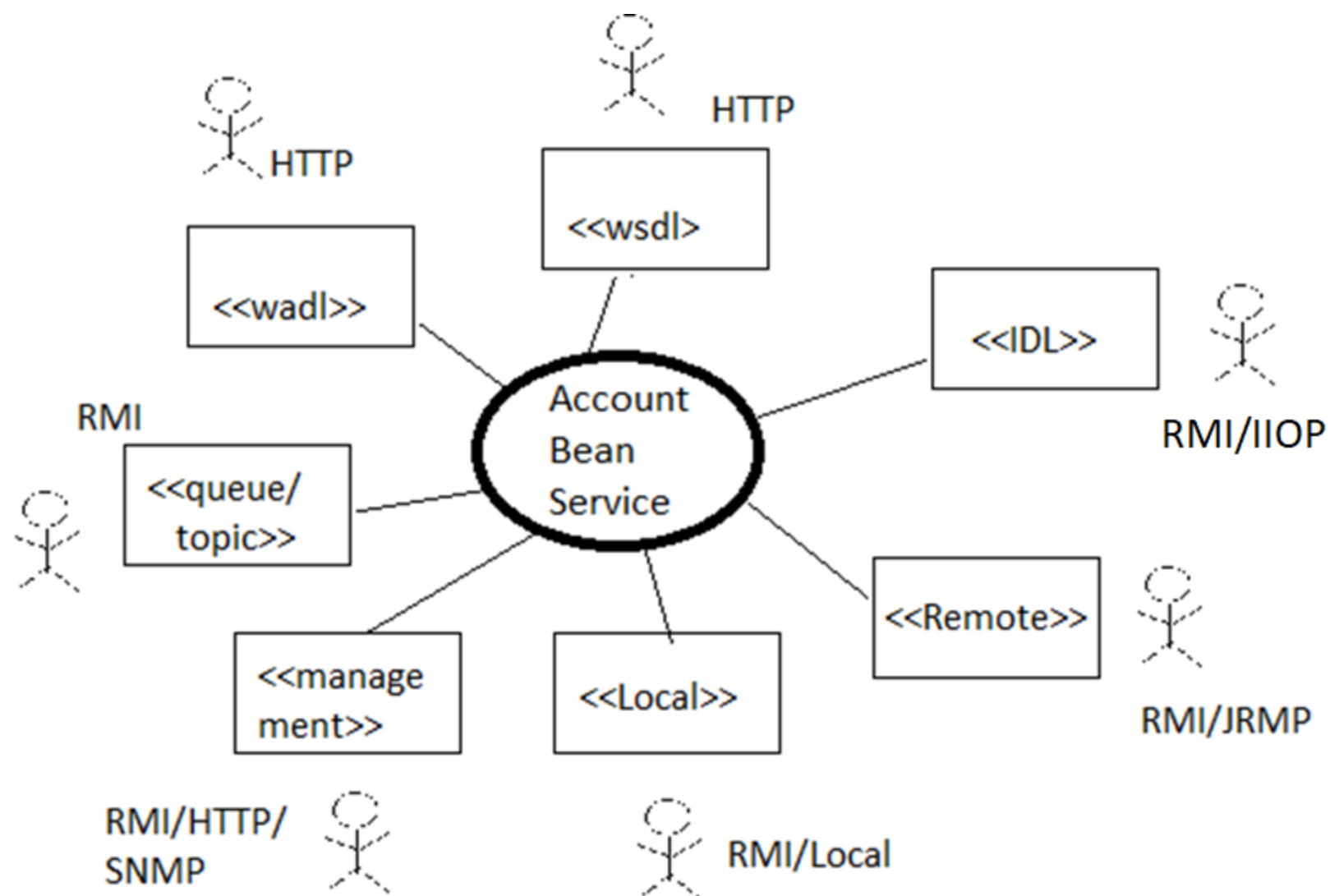
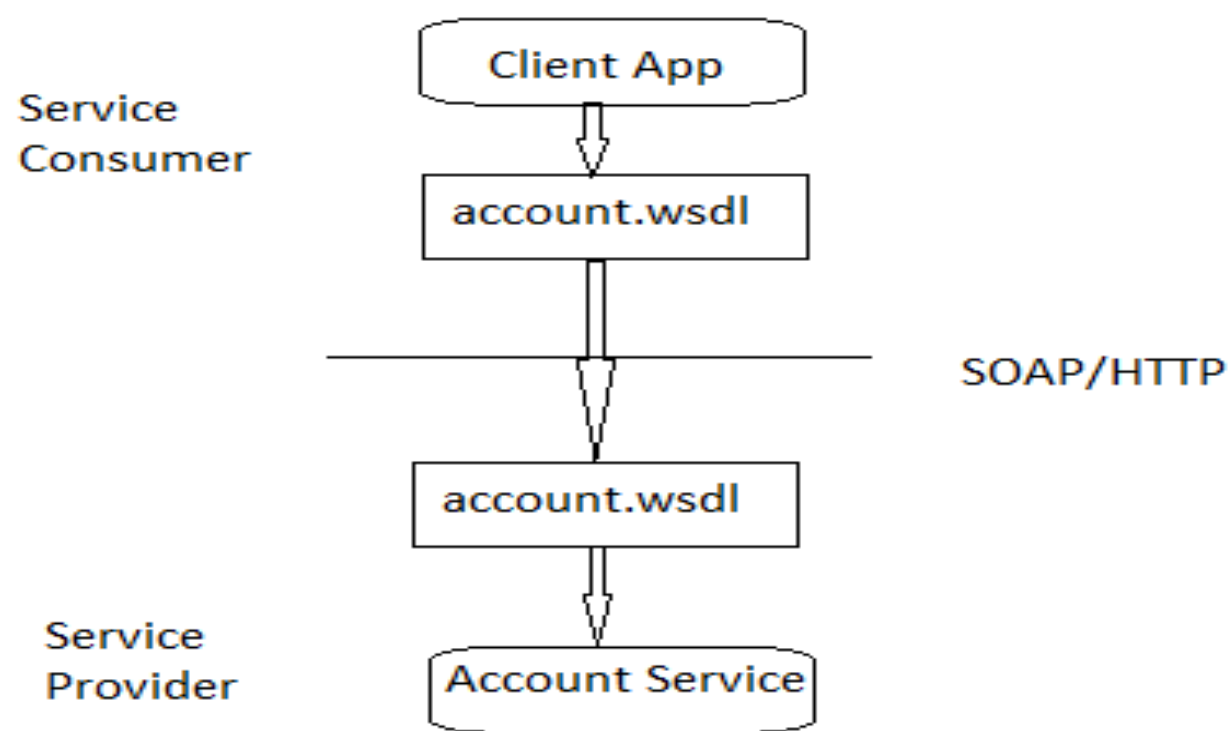


# Introduction To Microservice

D  
I  
S  
T  
R  
I  
B  
U  
T  
E  
D

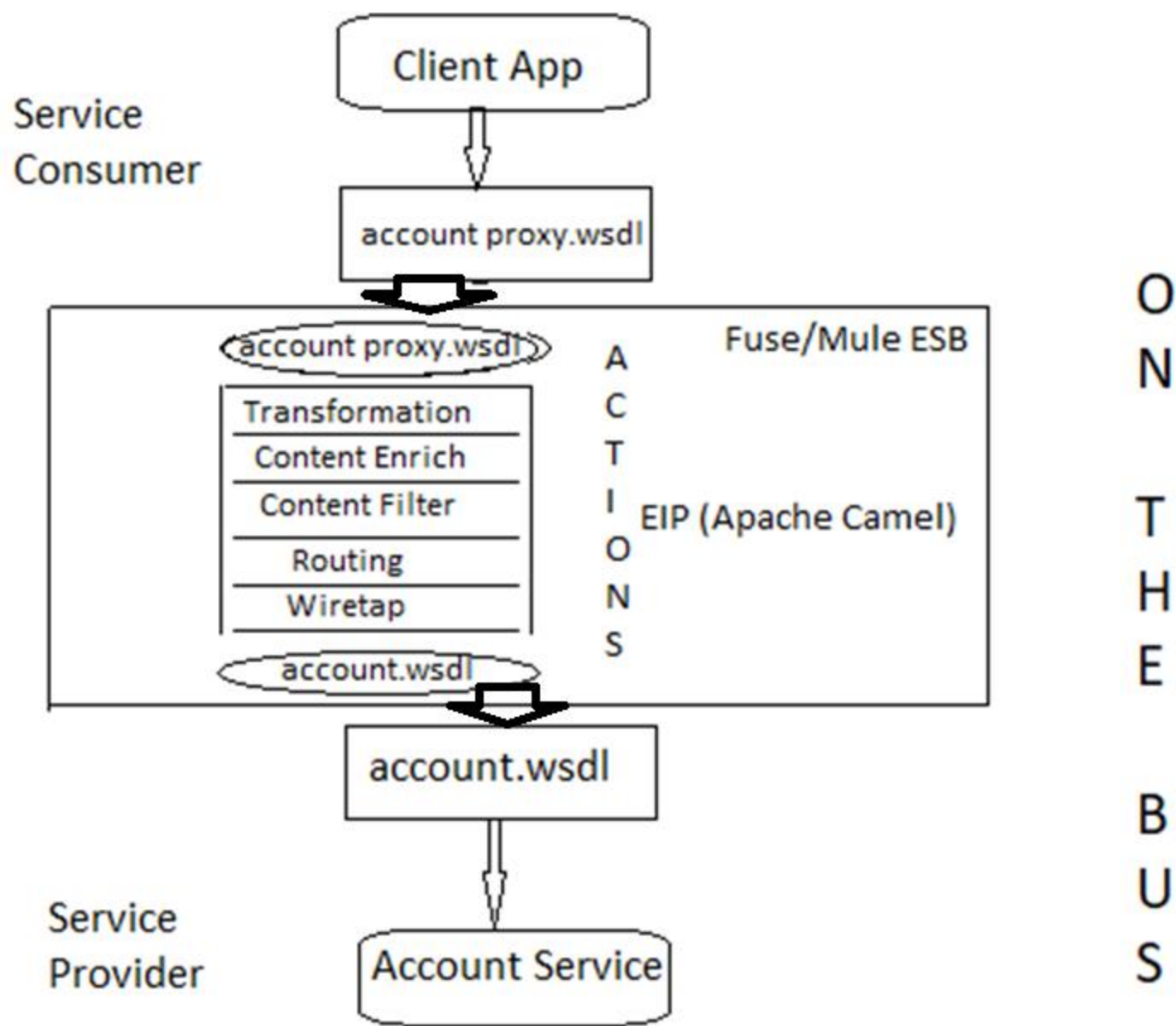


E  
N  
V  
I  
R  
O  
N  
M  
E  
N  
T



**Webserivce OFF the BUS**

W  
E  
B  
  
S  
E  
R  
V  
I  
C  
E



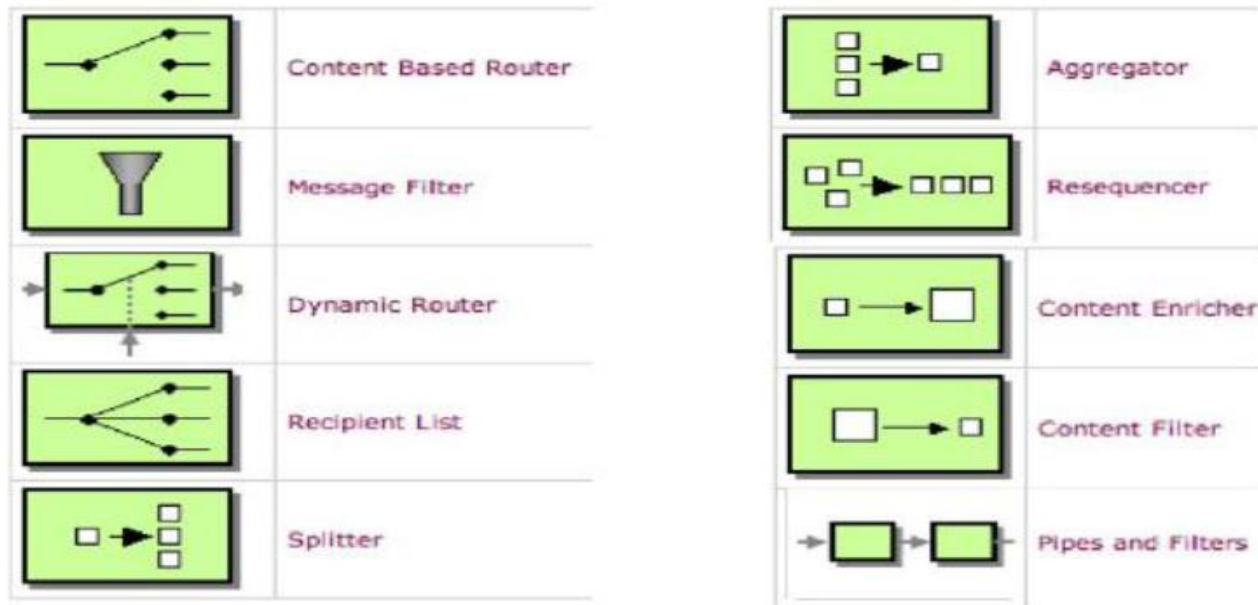
# SOA

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

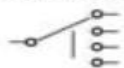
## Camel Endpoints

- \* 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

## Camel Components

- \* Provide a uniform Endpoint Interface
- \* Act as connectors to all other systems

### JMS Component

JMS API



JMS Provider  
ActiveMQ | IBM |  
Tibco | Sonic ...

### HTTP Component

Servlet API

Web Container  
Jetty | Tomcat | ...



HTTP Client

### File Component

File System

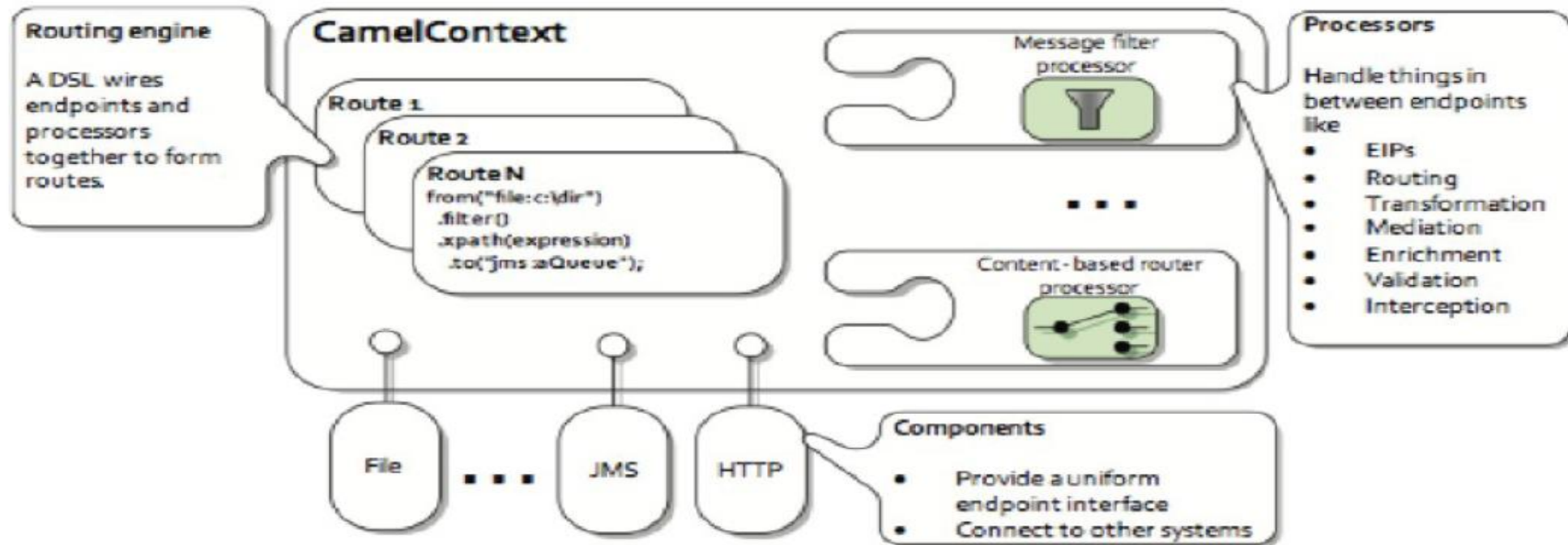


Local File System



# What is Apache Camel?

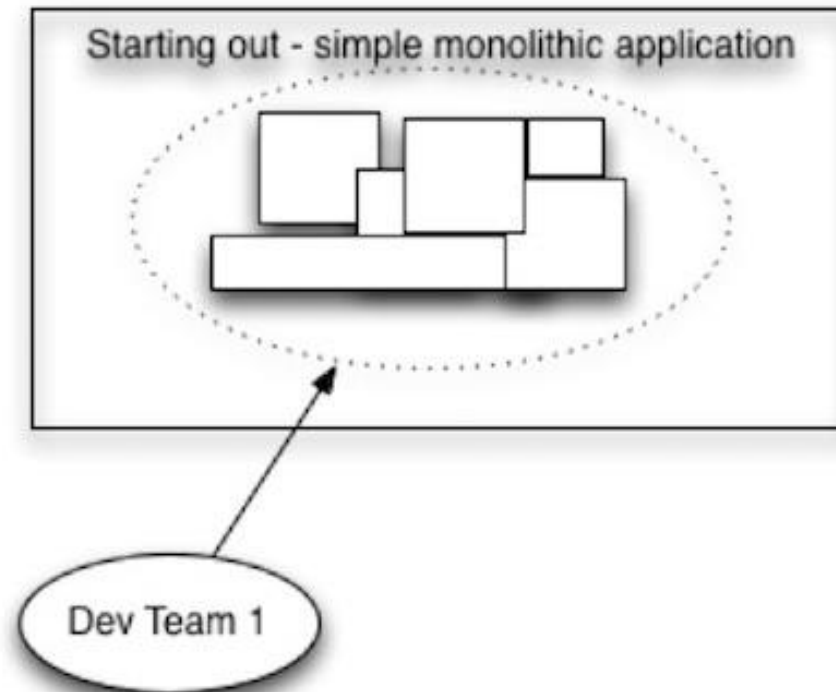
- Camel's Architecture



## Monolith Application:

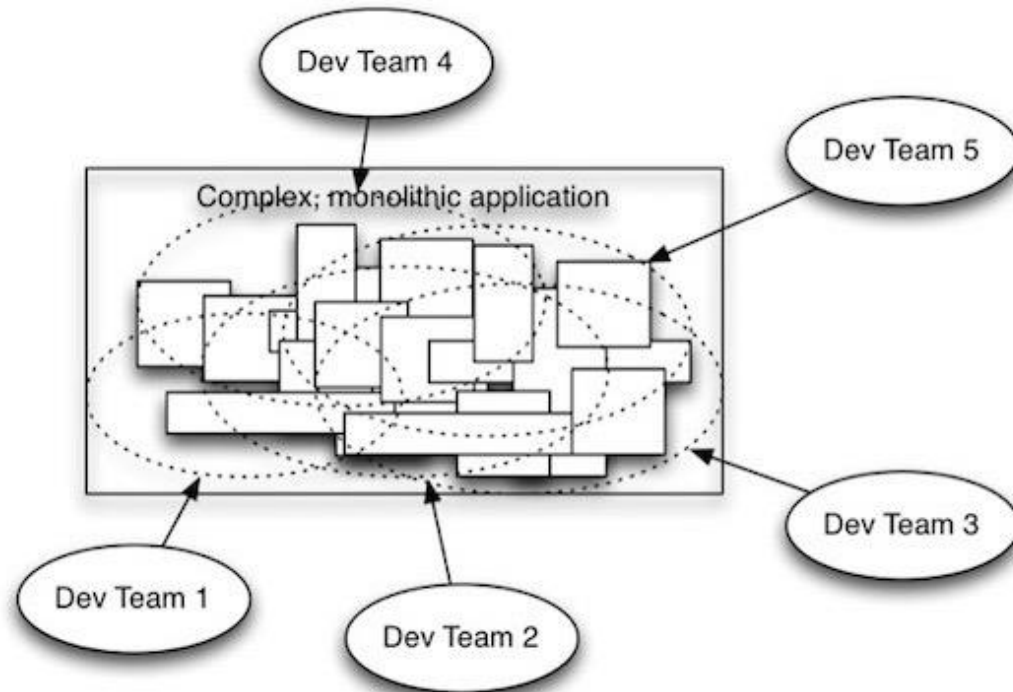
A single monolith would contain all the code for all the business activities an application performed.

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.

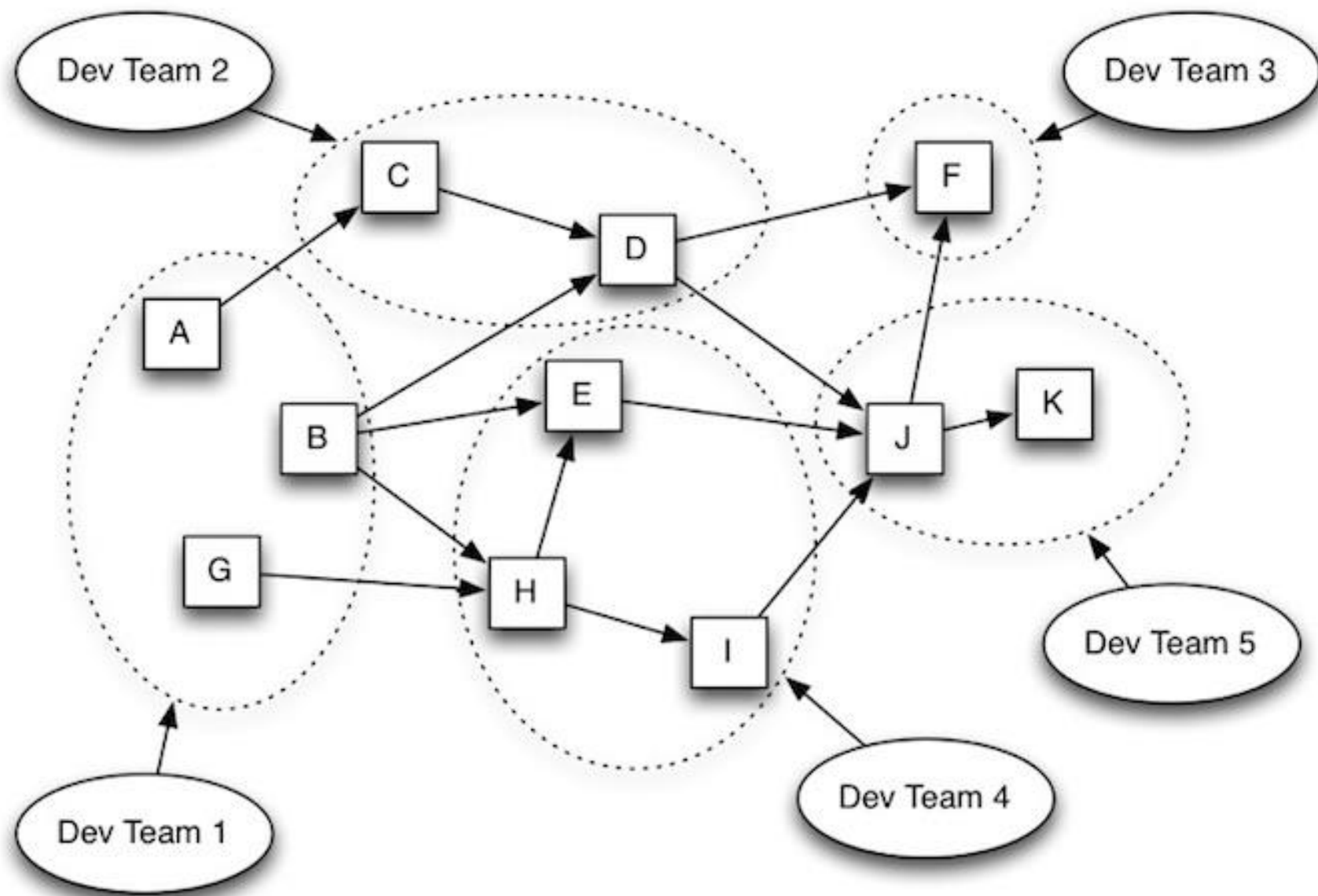
The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

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



## **Microservices**

So what exactly is a microservice architecture?

According to Martin Fowler:

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.

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

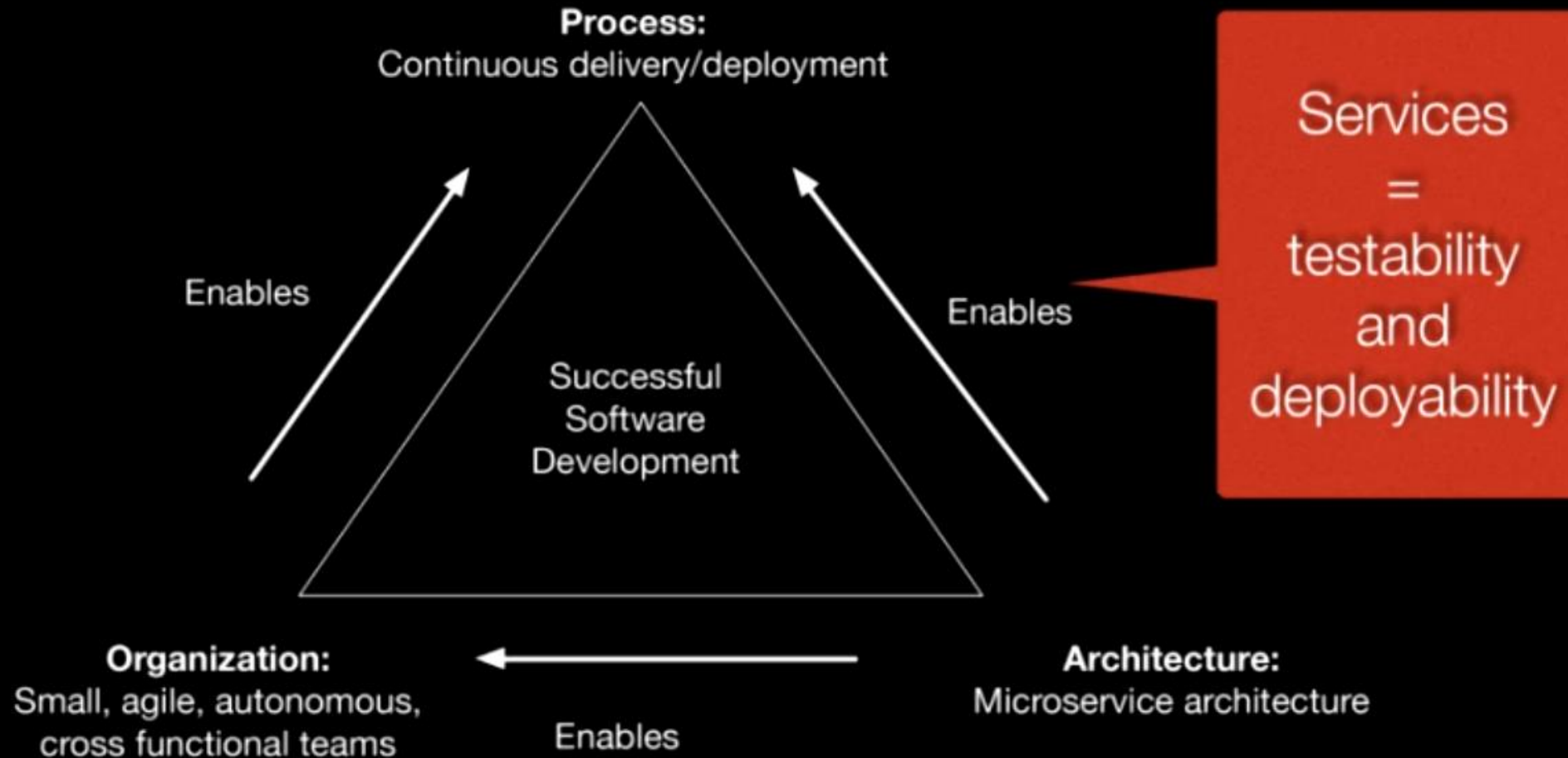
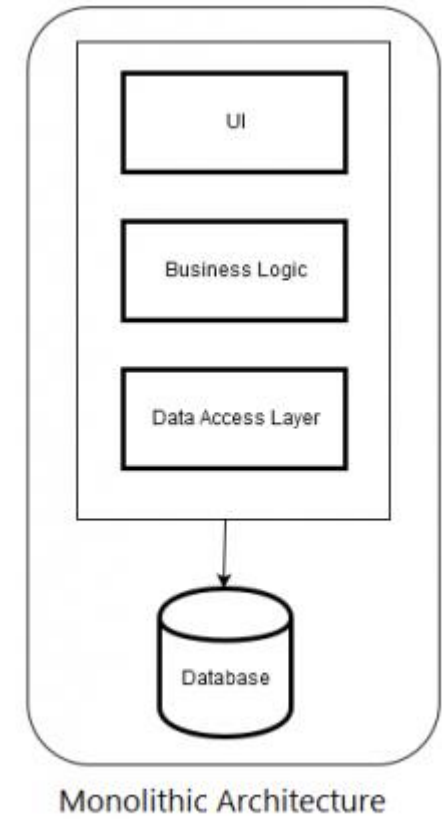
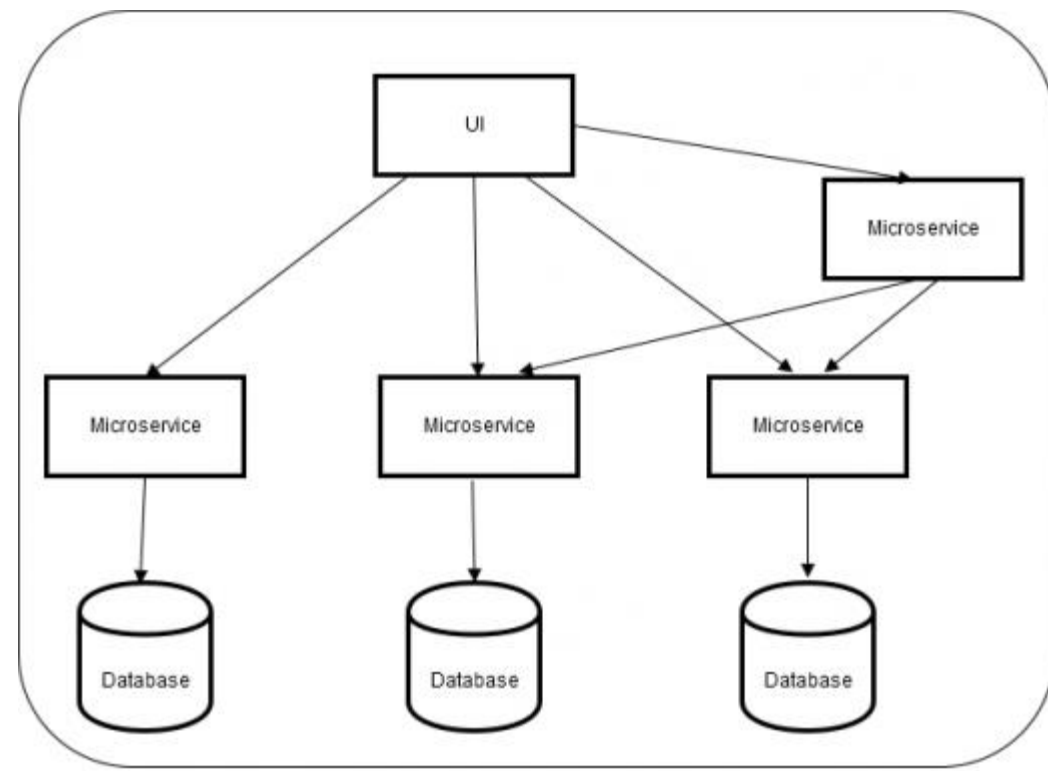
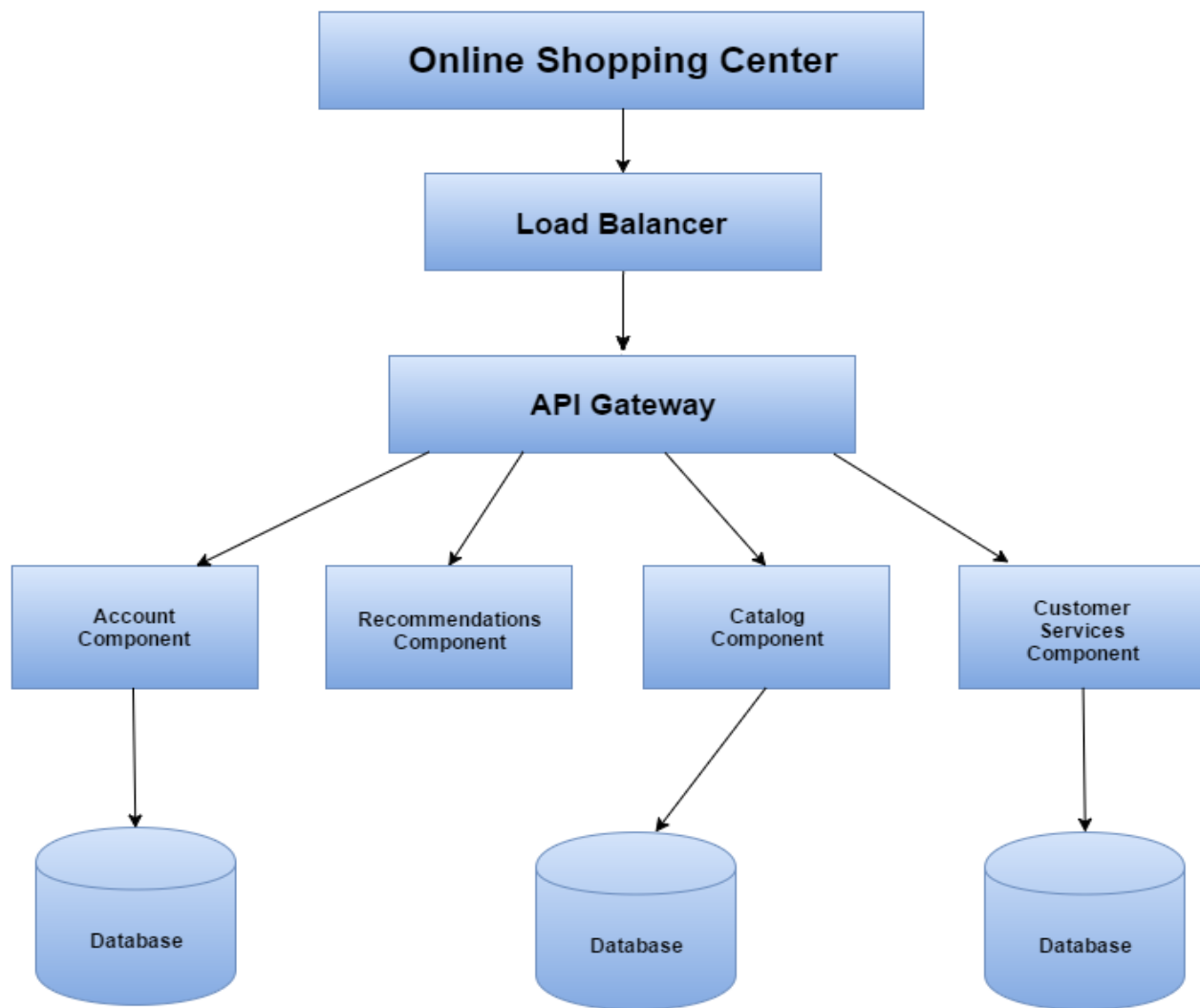




Figure 1: Conventional Approach





Microservices Architecture

Figure 2: Micro Services Approach

## SOA Vs Microservices

### SOA Architecture



### Microservices Architecture



## 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

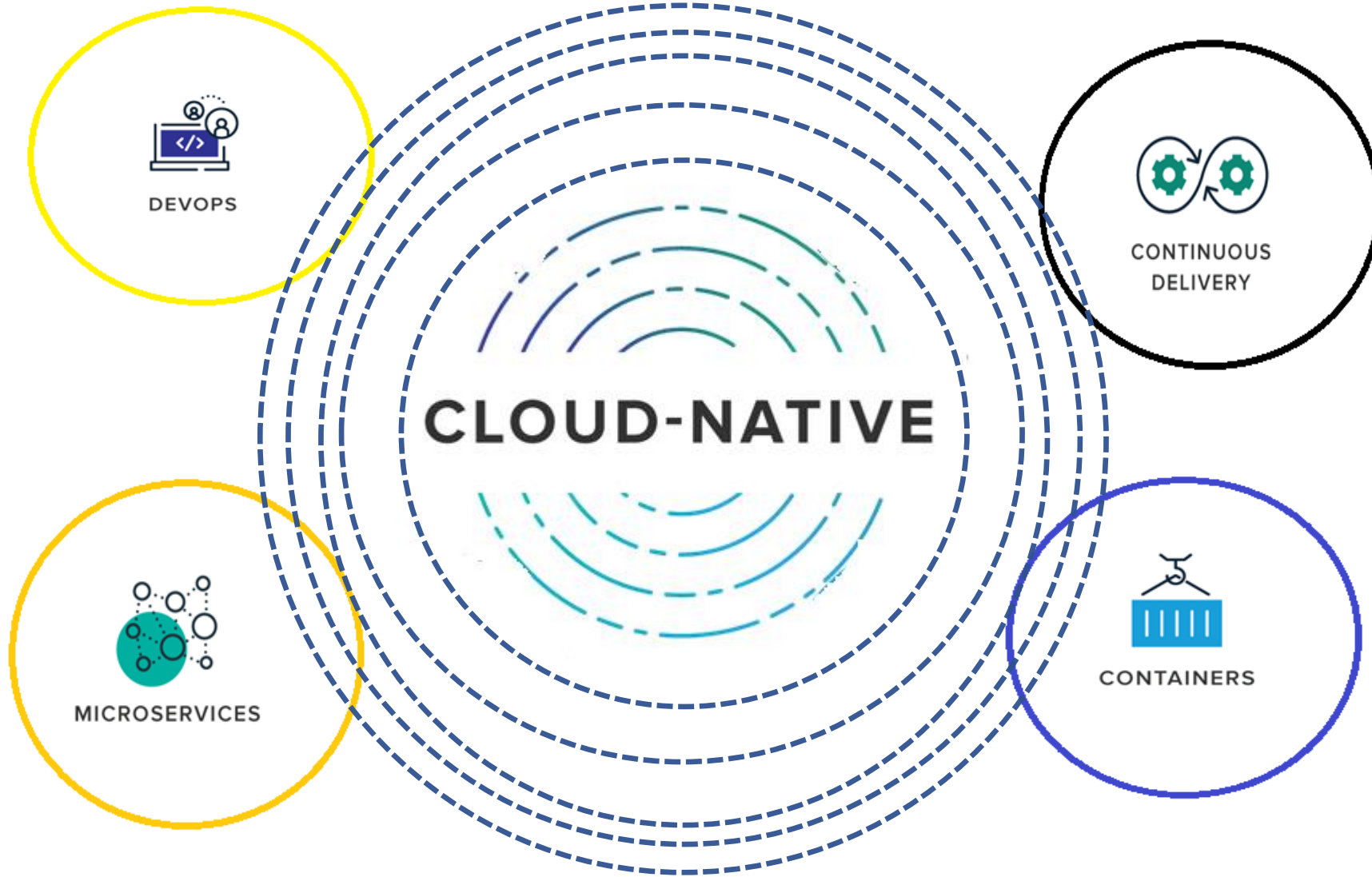
## **Microservices Challenges**

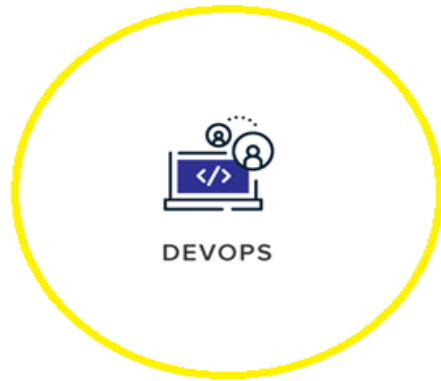
- ❑ Difficult to achieve strong consistency across services
- ❑ ACID transactions do not span multiple processes.
- ❑ Distributed System so hard to debug and trace the issues
- ❑ Greater need for end to end testing  
(JUnit, Selenium ,Arquillian, Hoverfly, AssertJ )
- ❑ Required cultural changes in across 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**.



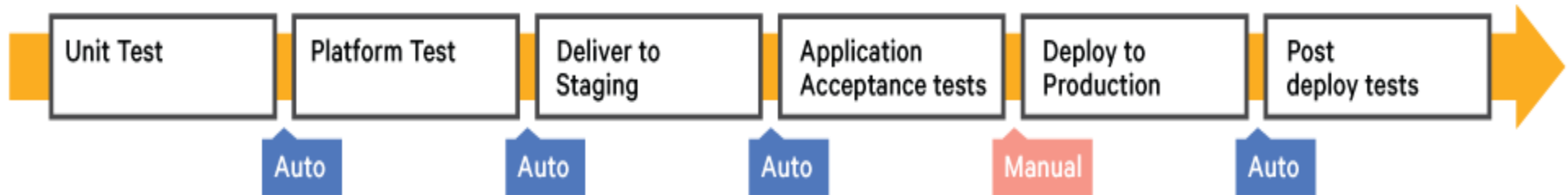


A DevOps culture helps developers and operations work together to deliver shared value to the customer.

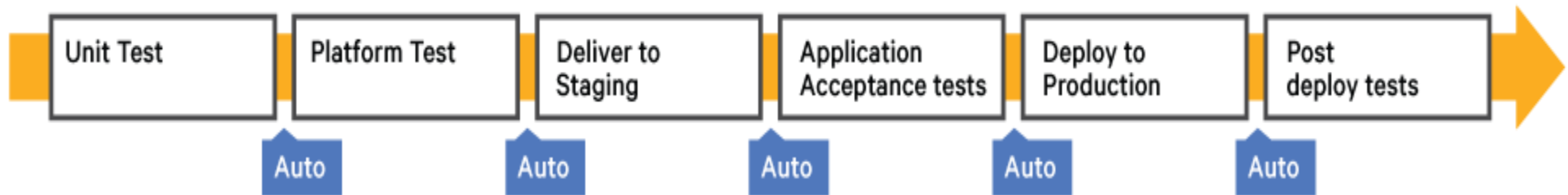


**Continuous Delivery**, enabled by Agile product development practices, is about shipping small batches of software to production constantly, through automation, at less risk, and get feedback faster from end users.

## Continuous Delivery



## Continuous Deployment





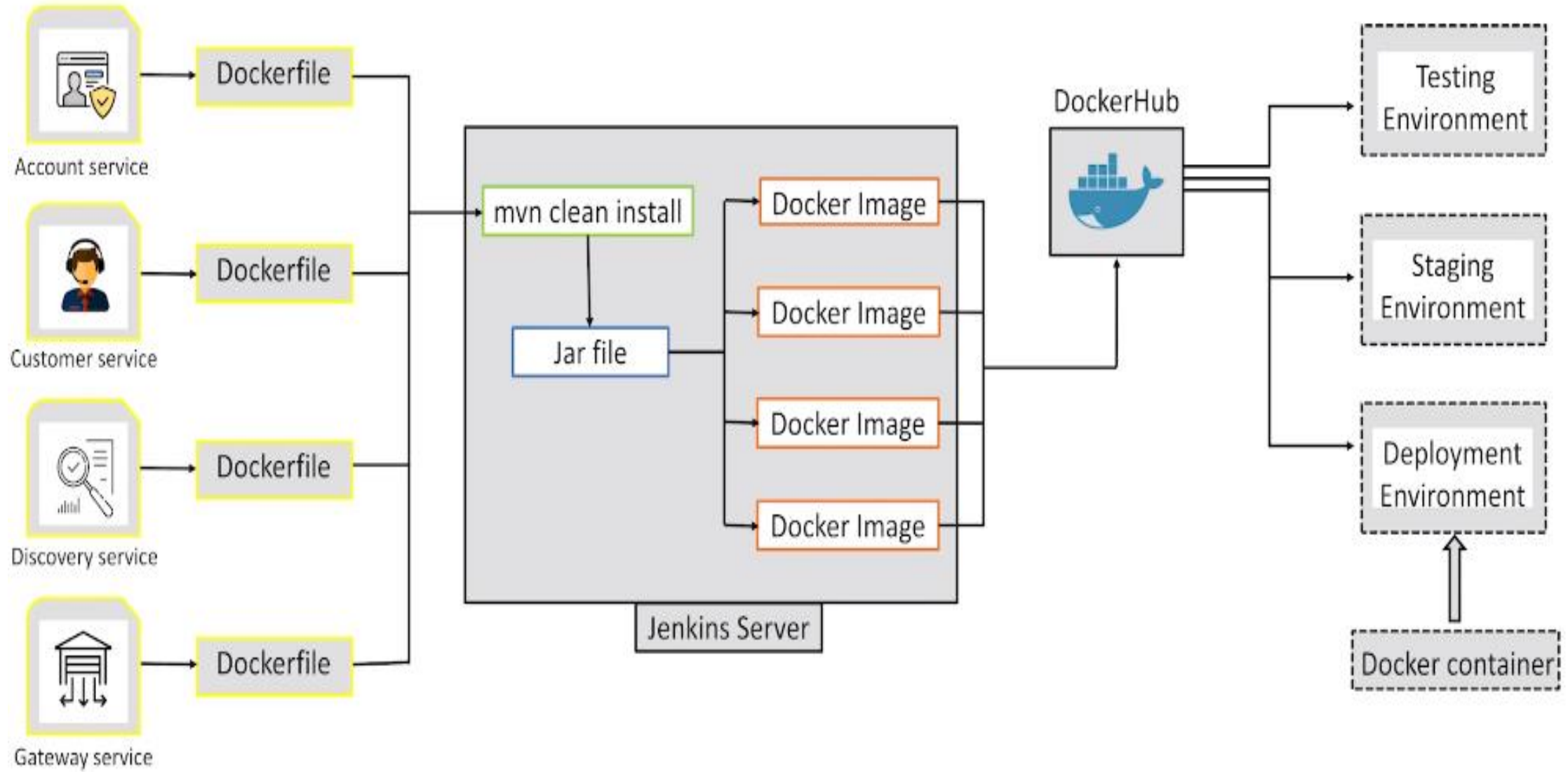


A microservices architecture is a method of developing software applications as a suite of:

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



Containerization -- also called container-based virtualization and application containerization -- is an OS-level virtualization method for **deploying and running distributed applications** without launching an entire VM for each application.



# Cloud Native Platform

Developer



## 1. Application Framework



Contract – 12 Factor App

Dev+Ops

## 2. Container Runtime



Contract – BOSH Release

IT Ops

## 3. Infrastructure Automation



Contract – Cloud Provider Interface

IT Ops

## 4. Infrastructure



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 Architecture



### Microservices Architecture



# 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 for new versions of code, running multiple versions of same service at same time

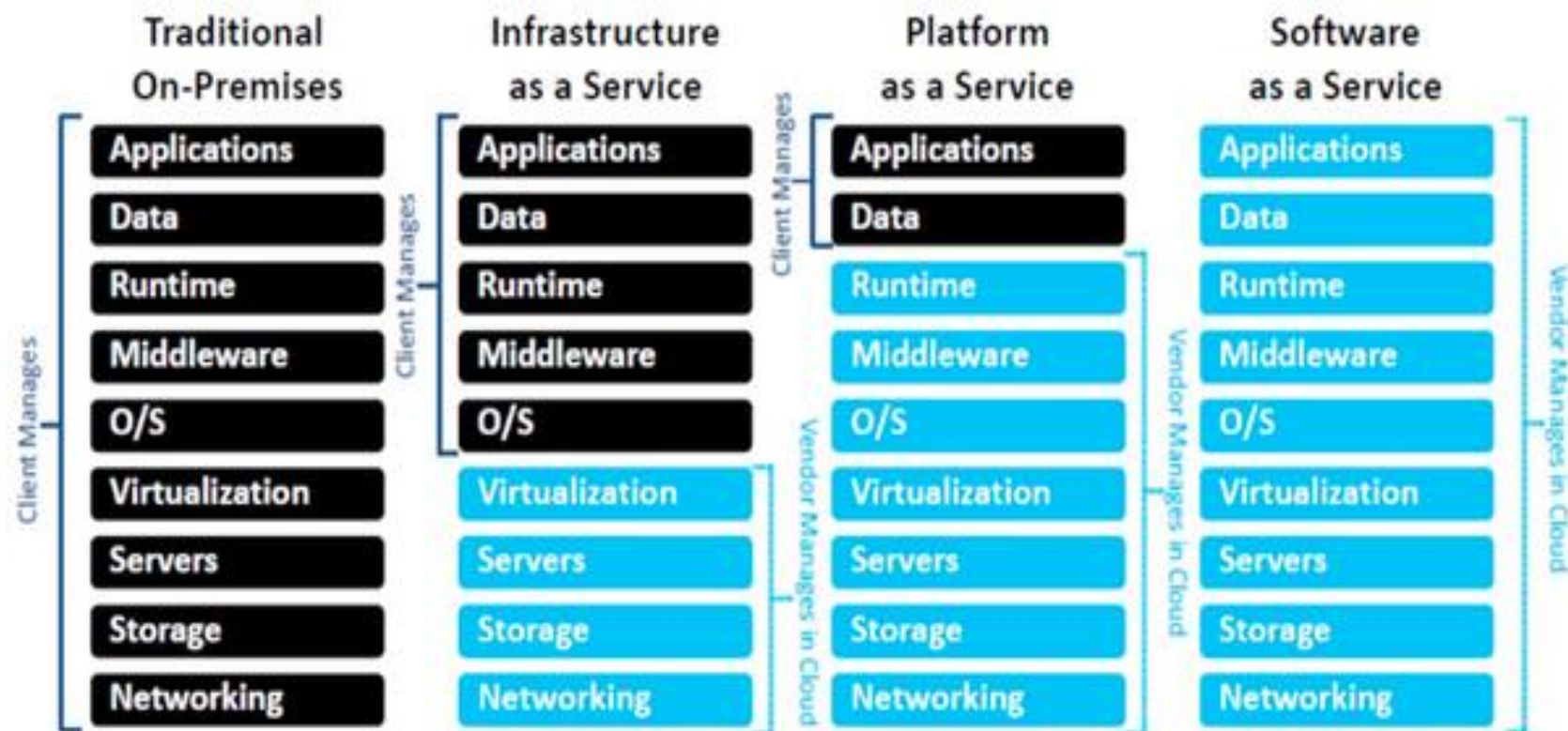
## **Virtualization ,Cloud Computing & Docker**



- ❑ **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.



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

**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.

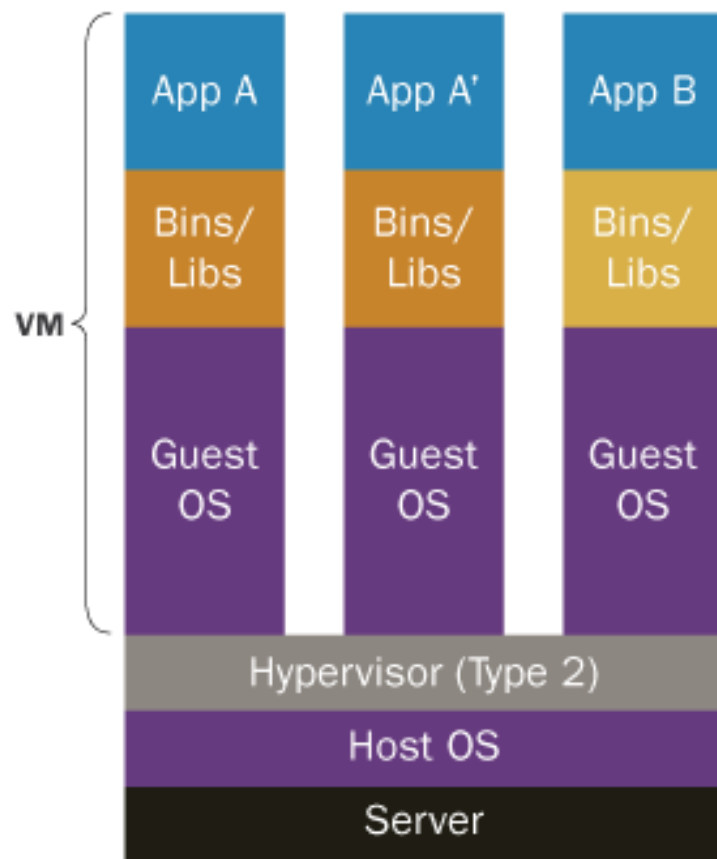
But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer.

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

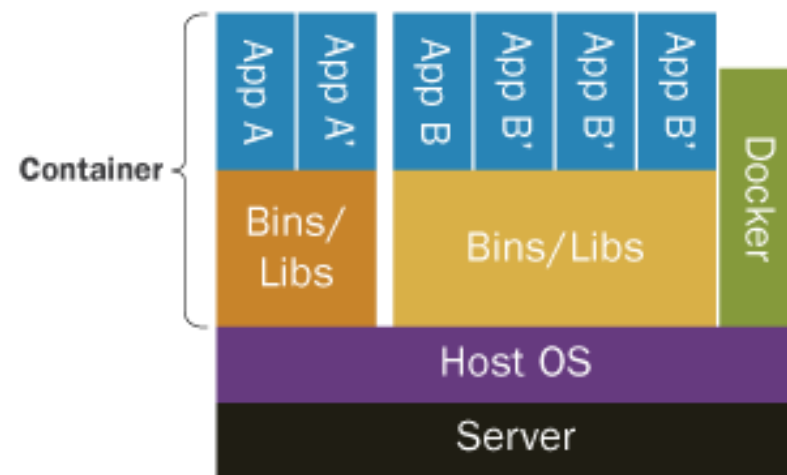
Docker is not a virtualization technology, it's an application delivery technology

PCF uses Garden container. Docker is another option

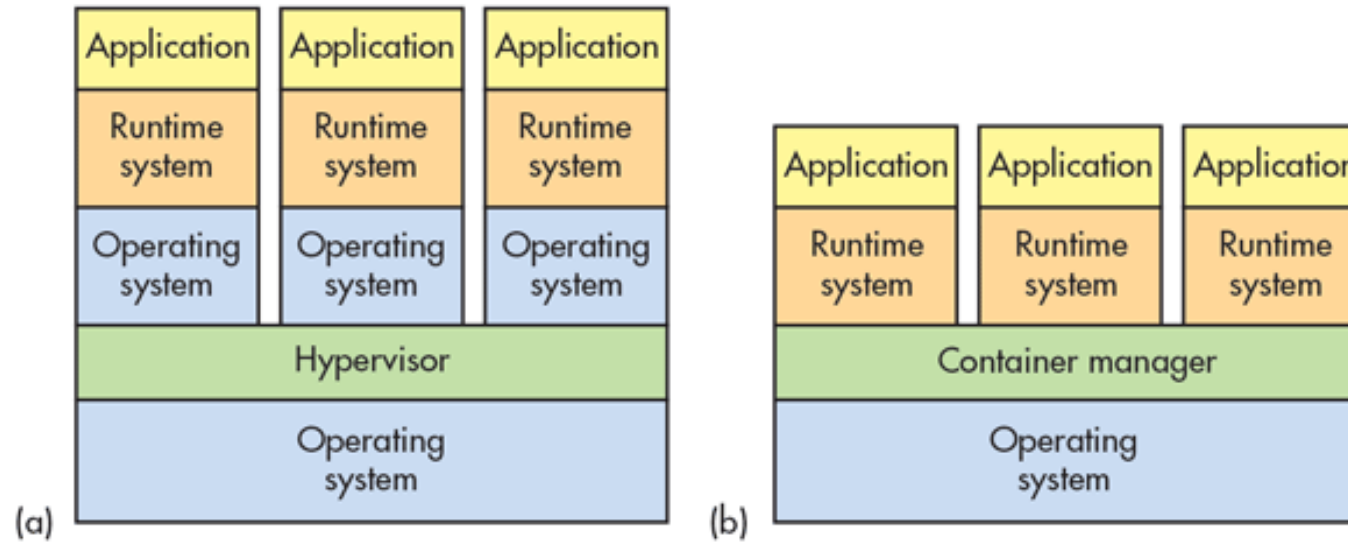
## Containers vs. VMs



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



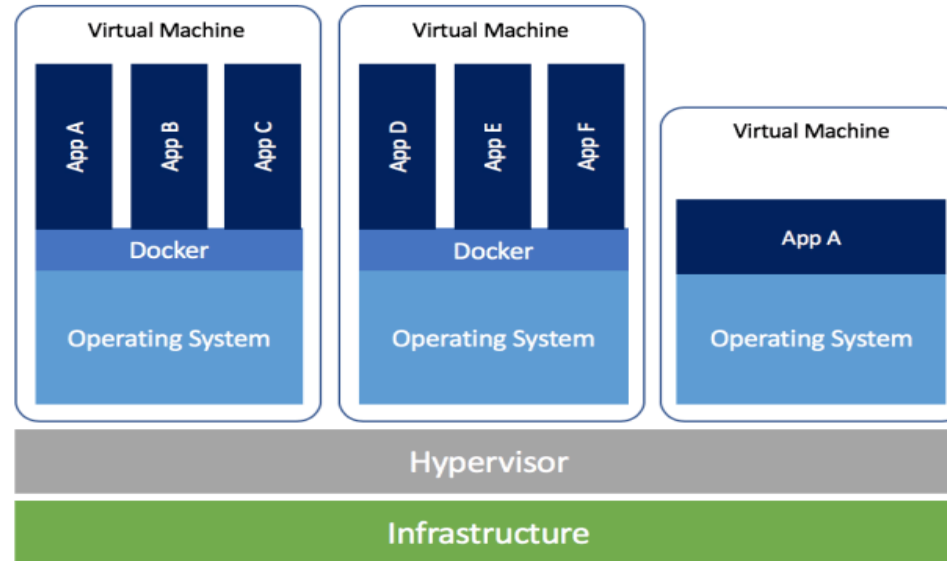




Virtual machines (VM) are managed by a hypervisor and utilize VM hardware

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



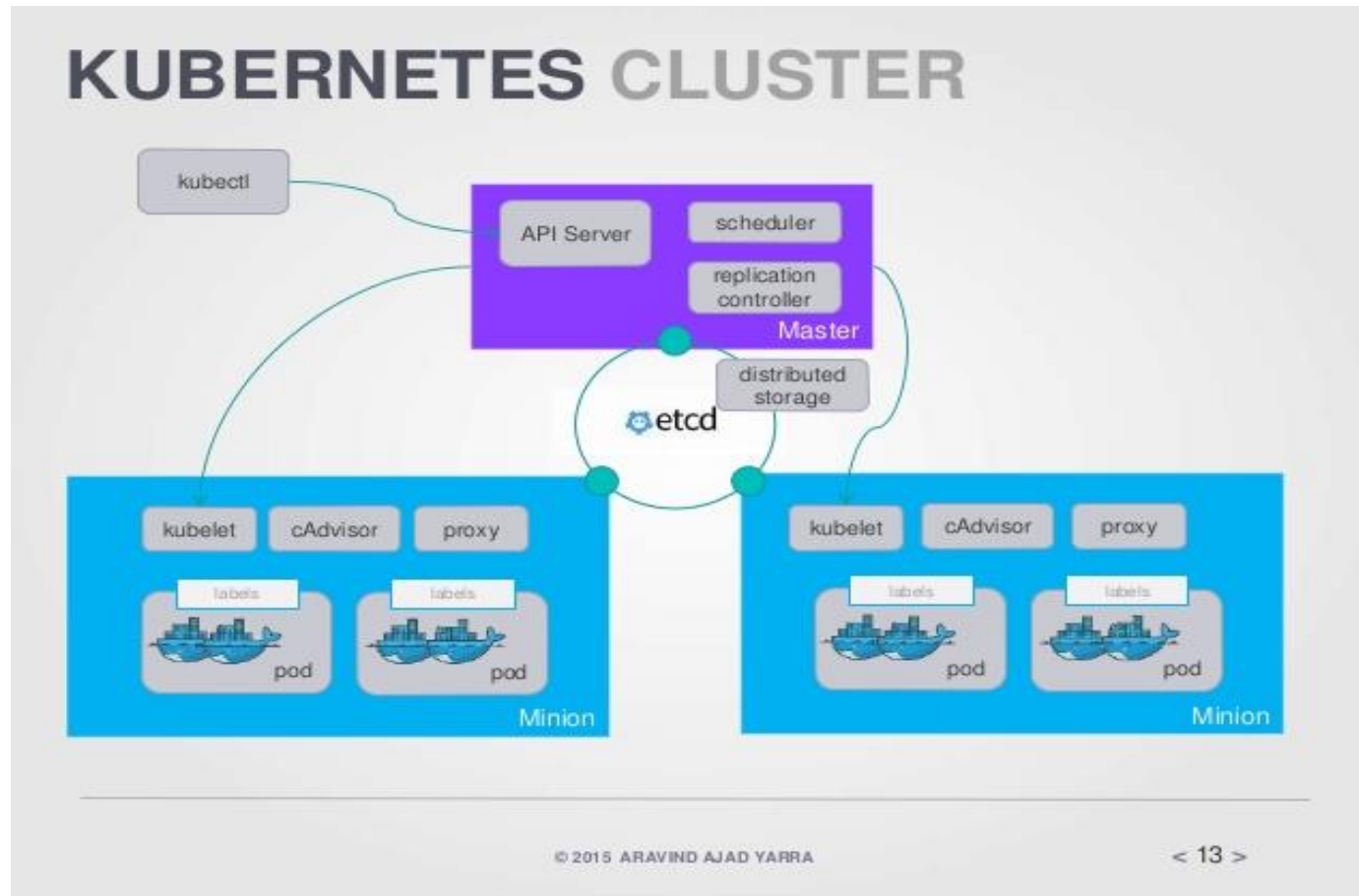
Containerized applications share common operating system and software libraries which greatly improves CPU utilization within a VM. This means an organization can reduce the overall number of virtual machines needed to operate their environment and increase the number of applications that can run on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

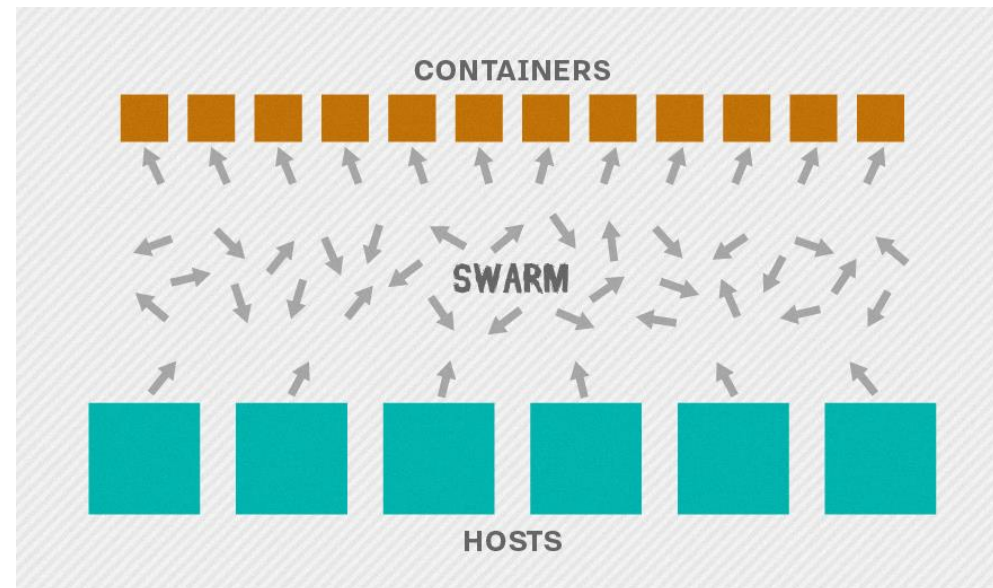
System containers are designed to run multiple processes, like virtual machines.

**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.

## **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

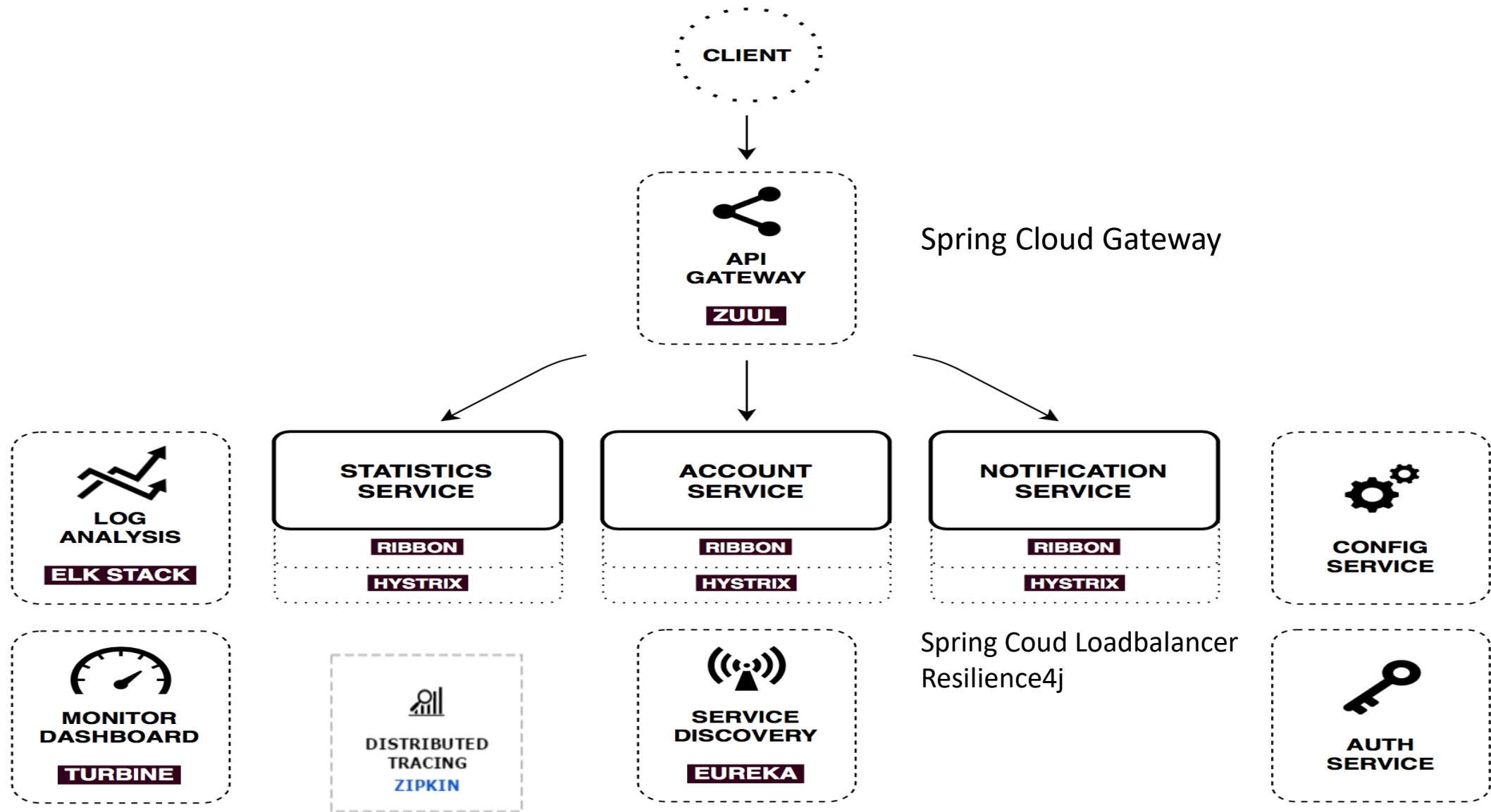


## **spring-cloud-netflix**

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)



### Service Discovery:

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

### Circuit Breaker:

**Hystrix** clients can be built with a simple annotation-driven method decorator  
Embedded Hystrix 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**

### 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** filters, and a simple convention over configuration approach to reverse proxy creation

### **Hystrix stream**

Netflix Hystrix has a neat feature called the Hystrix stream that provides real-time metrics on the state of the Hystrix commands in an application. This data tends to be very raw though, a very cool interface called the Hystrix Dashboard consumes this raw data and presents a graphical information based on the underlying raw data.

### **Turbine stream**

Hystrix stream provides information on a single application, Turbine provides a way to aggregate this information across all installations of an application in a cluster. Integrating turbine into a Spring-Cloud based application is straightforward, all it requires is information on which clusters to expose information on and how to aggregate information about the specific clusters.

### **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.

Resilience4j



Resilience4j is a lightweight fault tolerance library designed for Java 8 and functional programming.

The library uses Vavr, which does not have any other external library dependencies.

Resilience4j allows picking what you need.

The library helps with implementing resilient systems by managing fault tolerance for remote communications.

The library is inspired by Hystrix but offers a much more convenient API and a number of other features like Rate Limiter (block too frequent requests), Bulkhead (avoid too many concurrent requests) etc.

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. Netflix Hystrix, by contrast, has a dependency on Archaius which has several other external library dependencies such as Guava and Apache Commons

## 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.

## **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.

## **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.

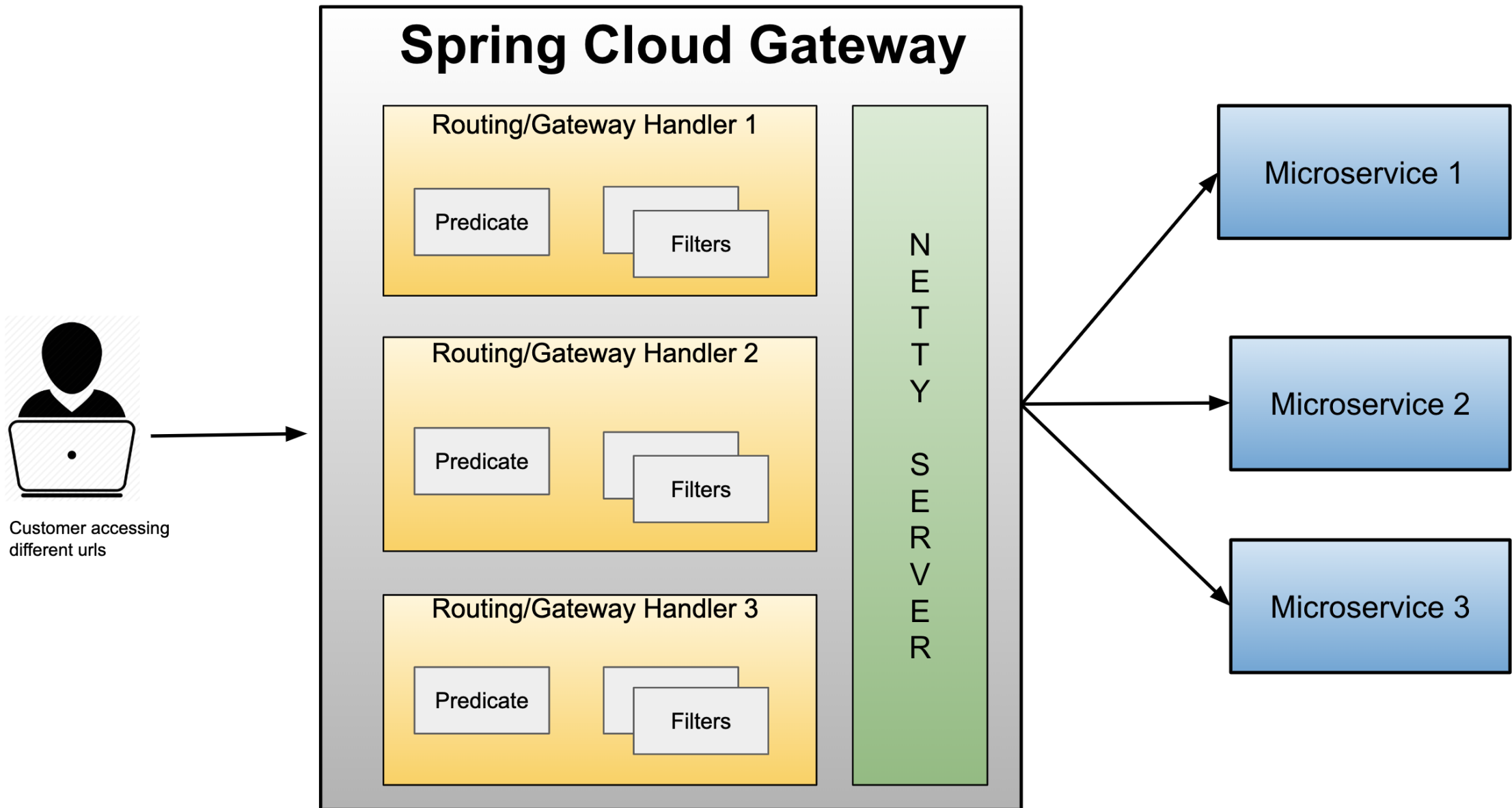
## **Spring Cloud Gateway**



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 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.

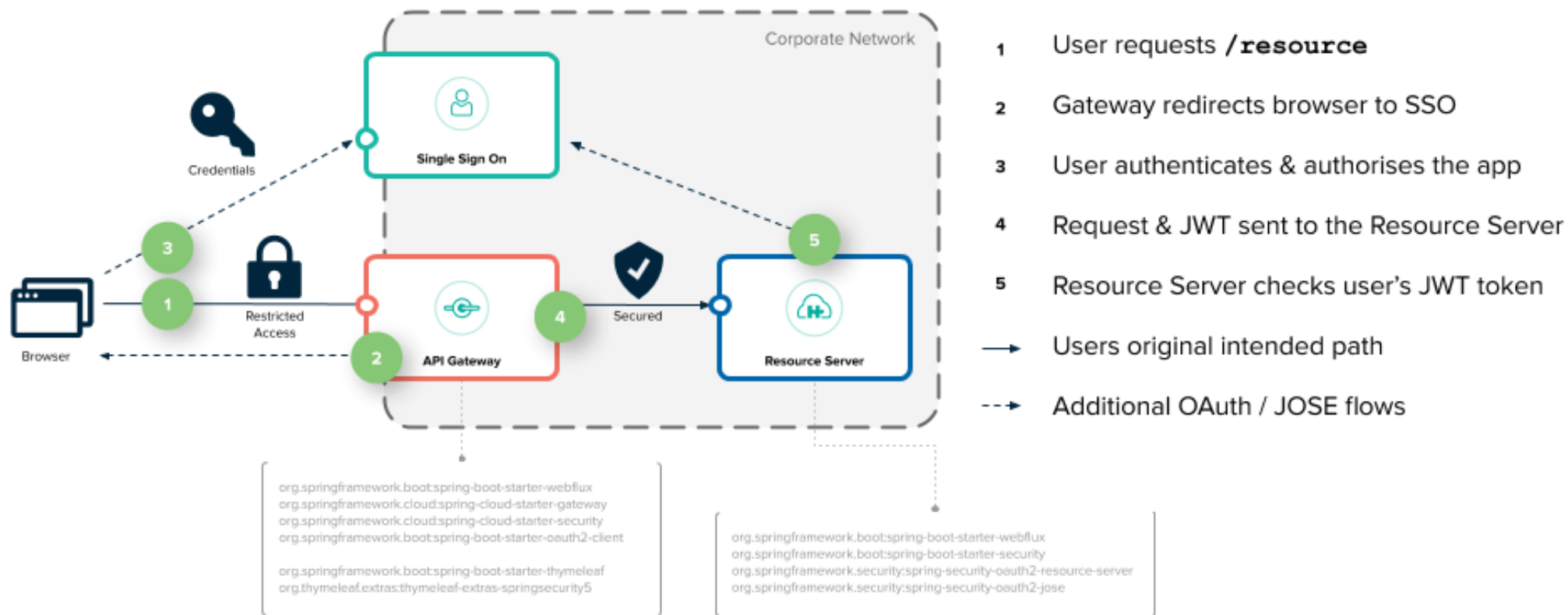
There are quite a lot of out of box WebFilter that framework provides

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



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 <code>slowCallDurationThreshold</code> . 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.
permittedNumberOfCallsInHalfOpenState	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.

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

# 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.