



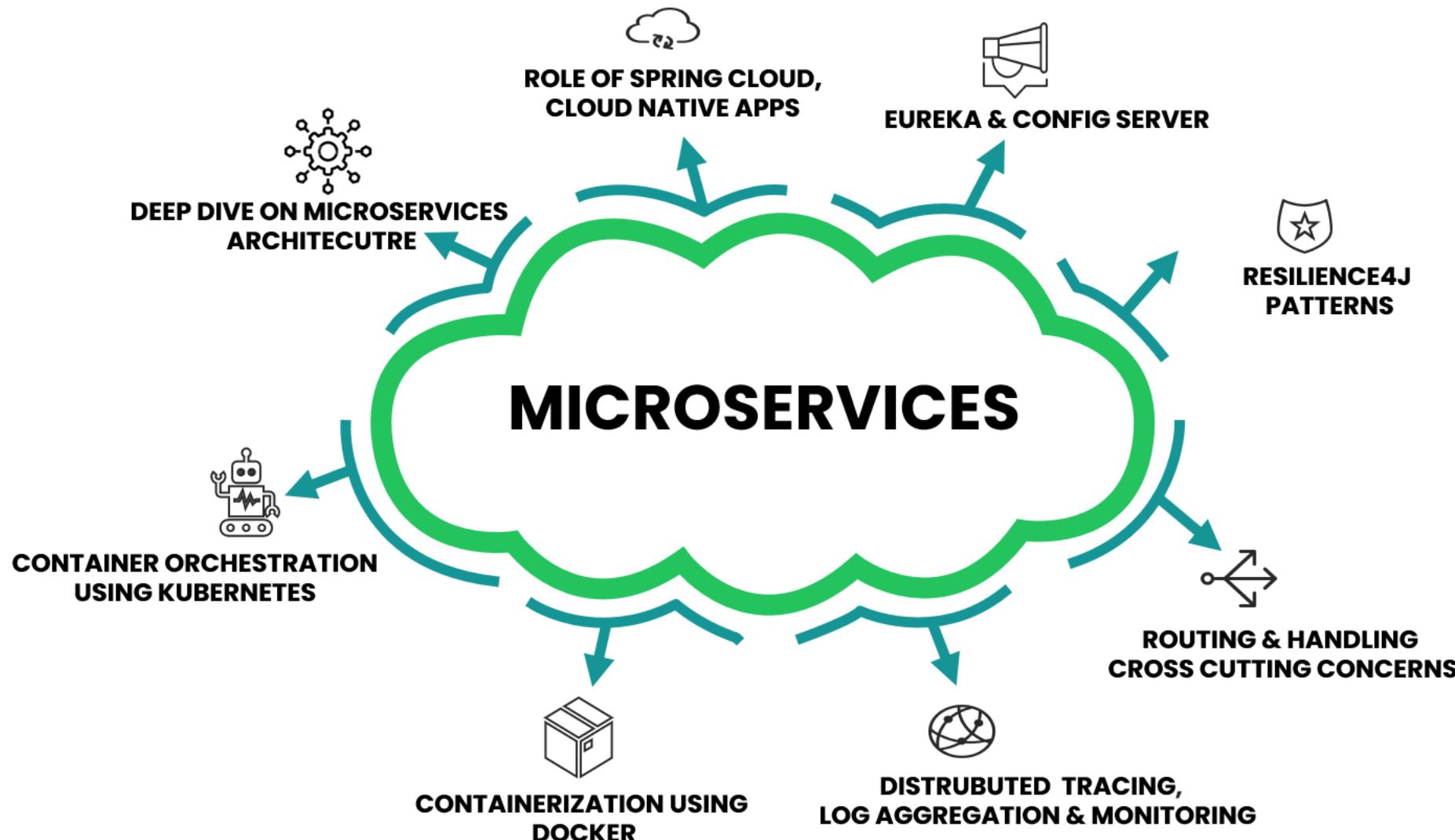
# MICROSERVICES

USING SPRING, DOCKER, KUBERNETES

# MICROSERVICES WITH SPRING, DOCKER, KUBERNETES

eazy  
bytes

## WHAT WE COVER IN THIS COURSE

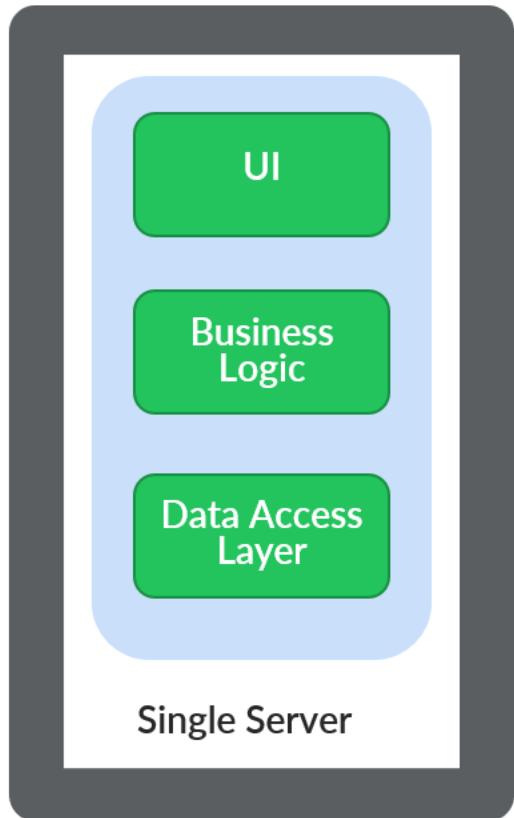


# MONOLITHIC VS SOA VS MICROSERVICES

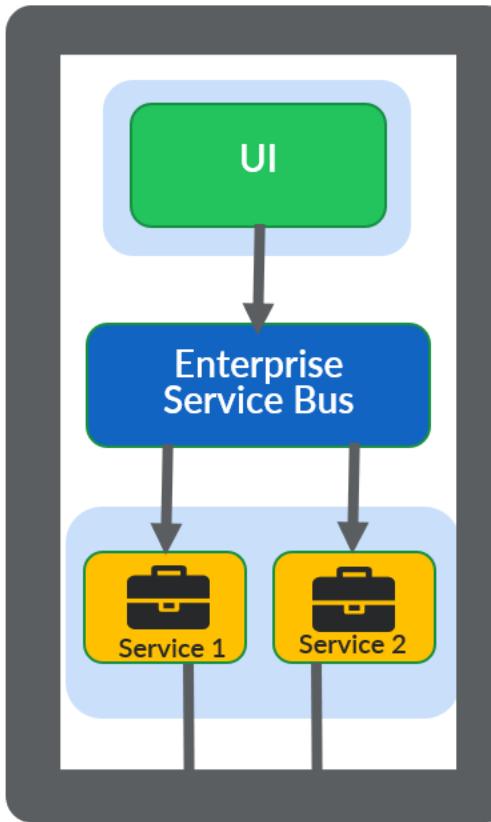
EVOLUTION OF MICROSERVICES

eazy  
bytes

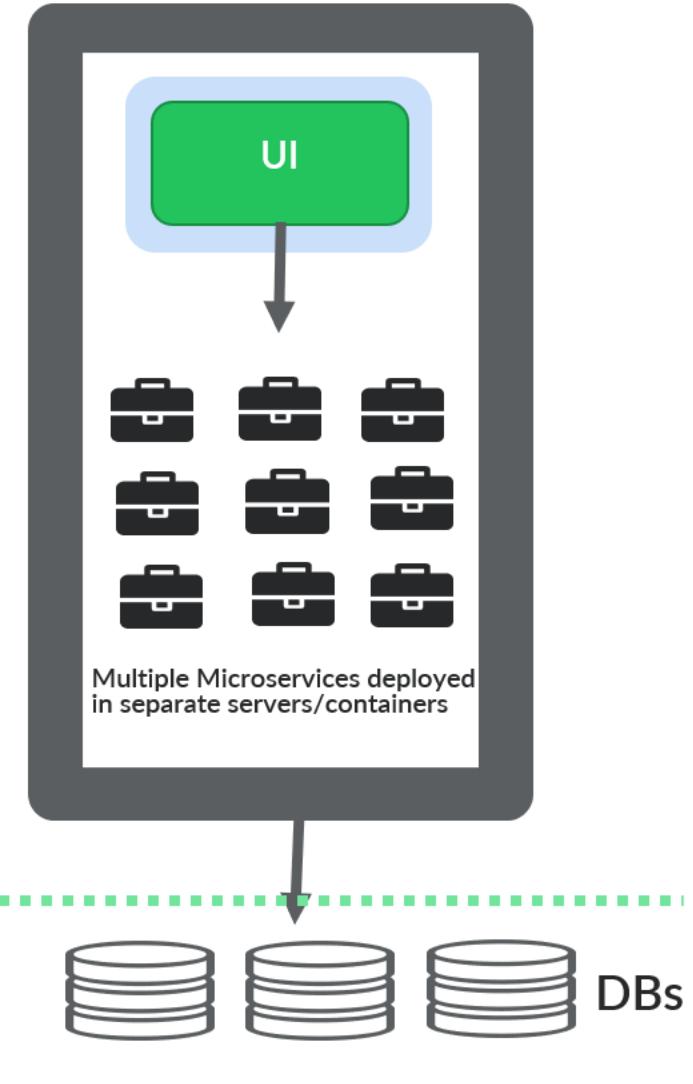
MONOLITHIC



SOA



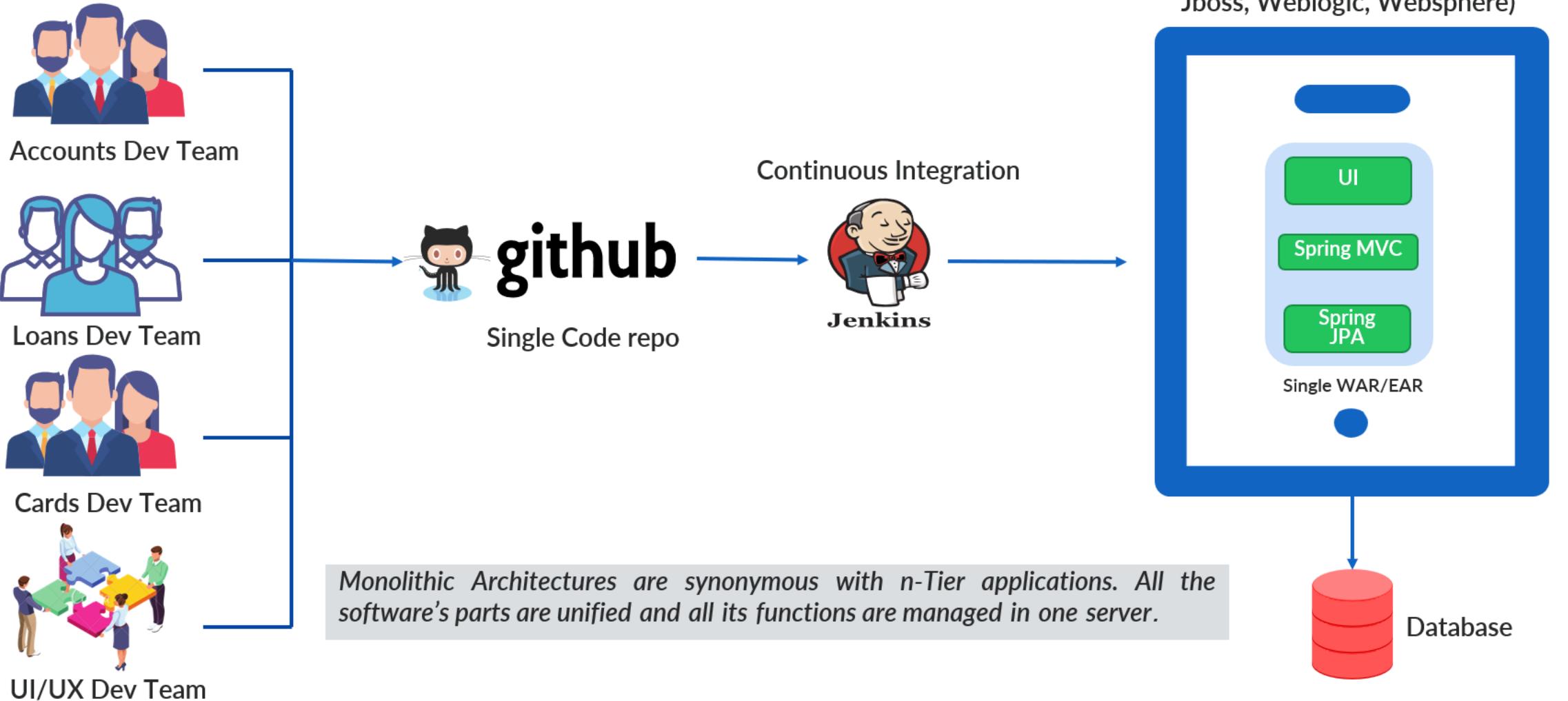
MICROSERVICES



# MONOLITHIC ARCHITECTURE

## SAMPLE BANK APPLICATION

eazy  
bytes

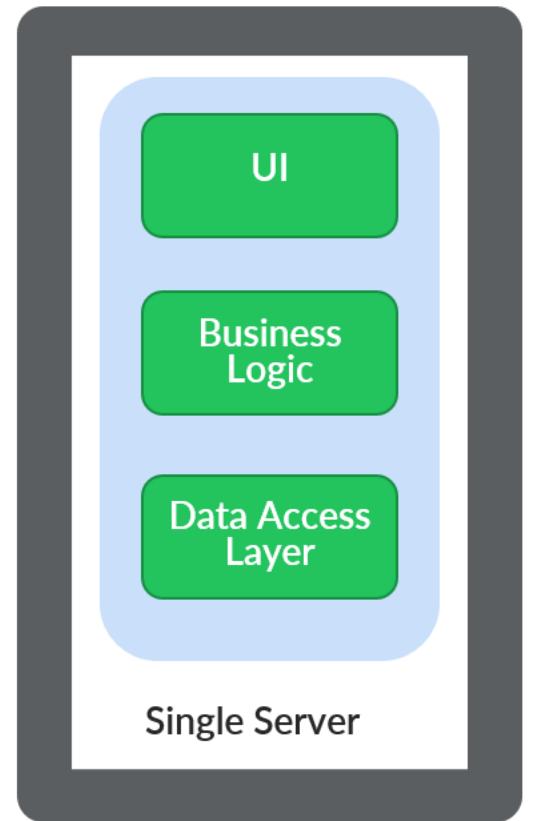


# MONOLITHIC ARCHITECTURE

## PROS & CONS

eazy  
bytes

### MONOLITHIC



#### Pros

- Simpler development and deployment for smaller teams and applications
- Fewer cross-cutting concerns
- Better performance due to no network latency

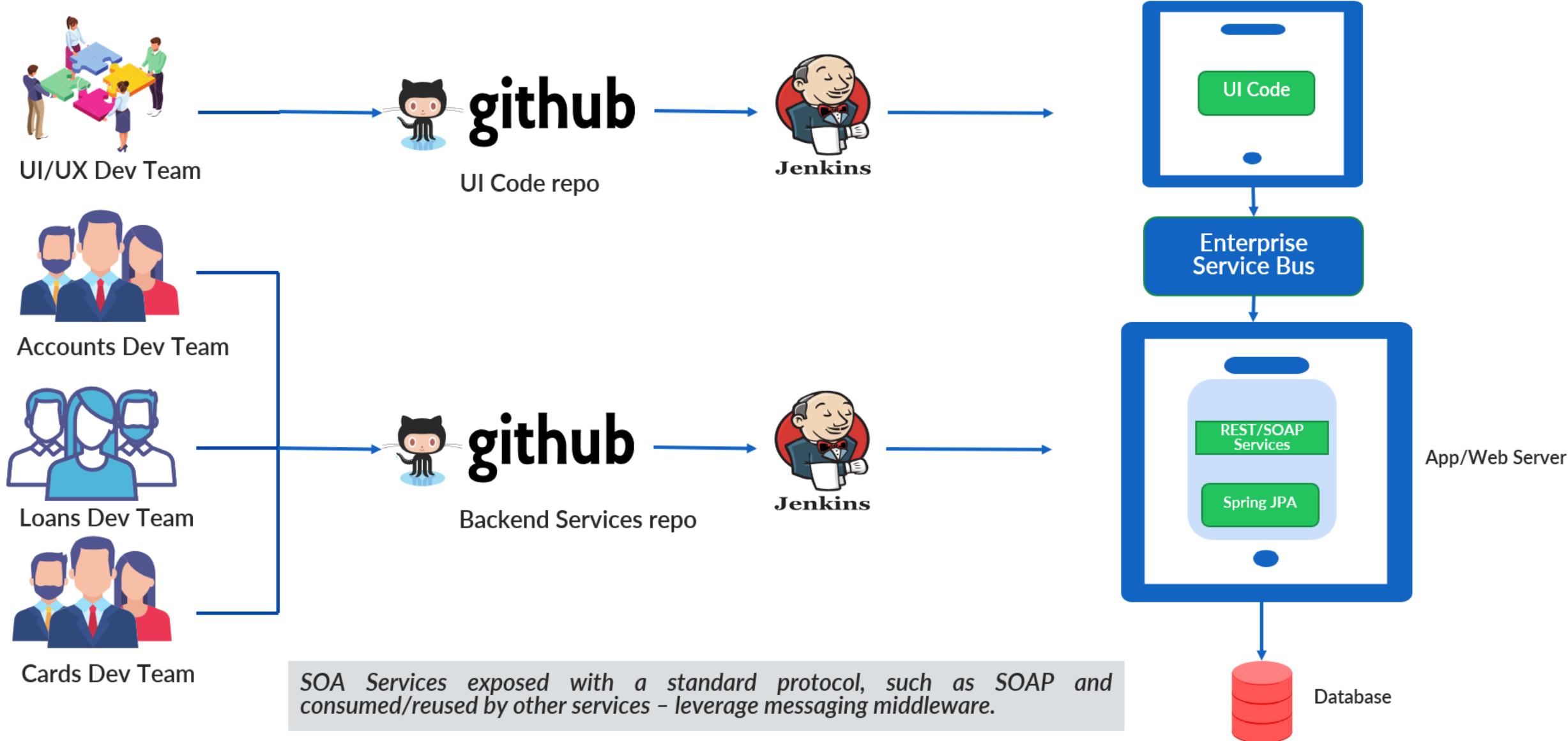
#### Cons

- Difficult to adopt new technologies
- Limited agility
- Single code base and difficult to maintain
- Not Fault tolerance
- Tiny update and feature development always need a full deployment

# SOA ARCHITECTURE

## SAMPLE BANK APPLICATION

eazy  
bytes

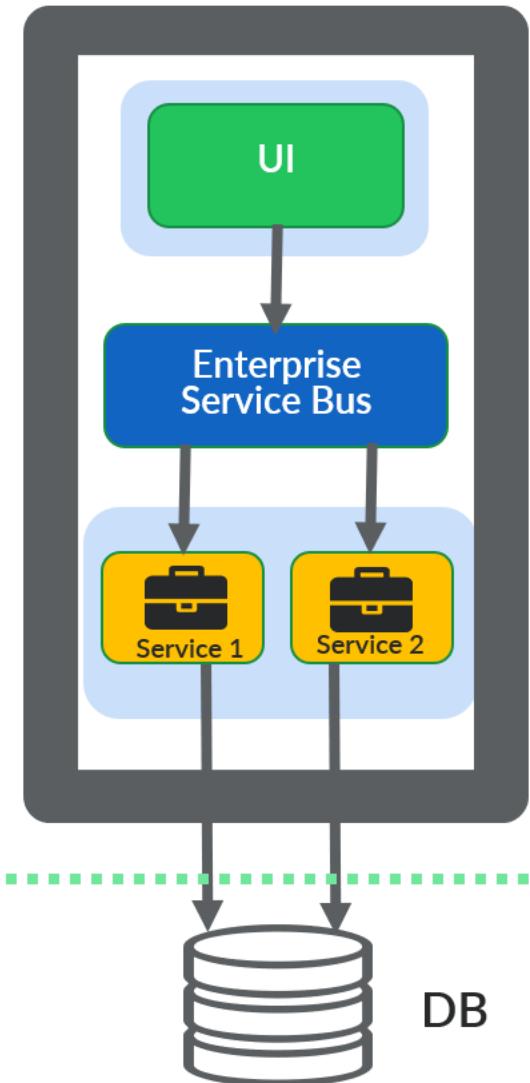


# SOA ARCHITECTURE

## PROS & CONS

eazy  
bytes

SOA



### Pros

- Reusability of services
- Better maintainability
- Higher reliability
- Parallel development

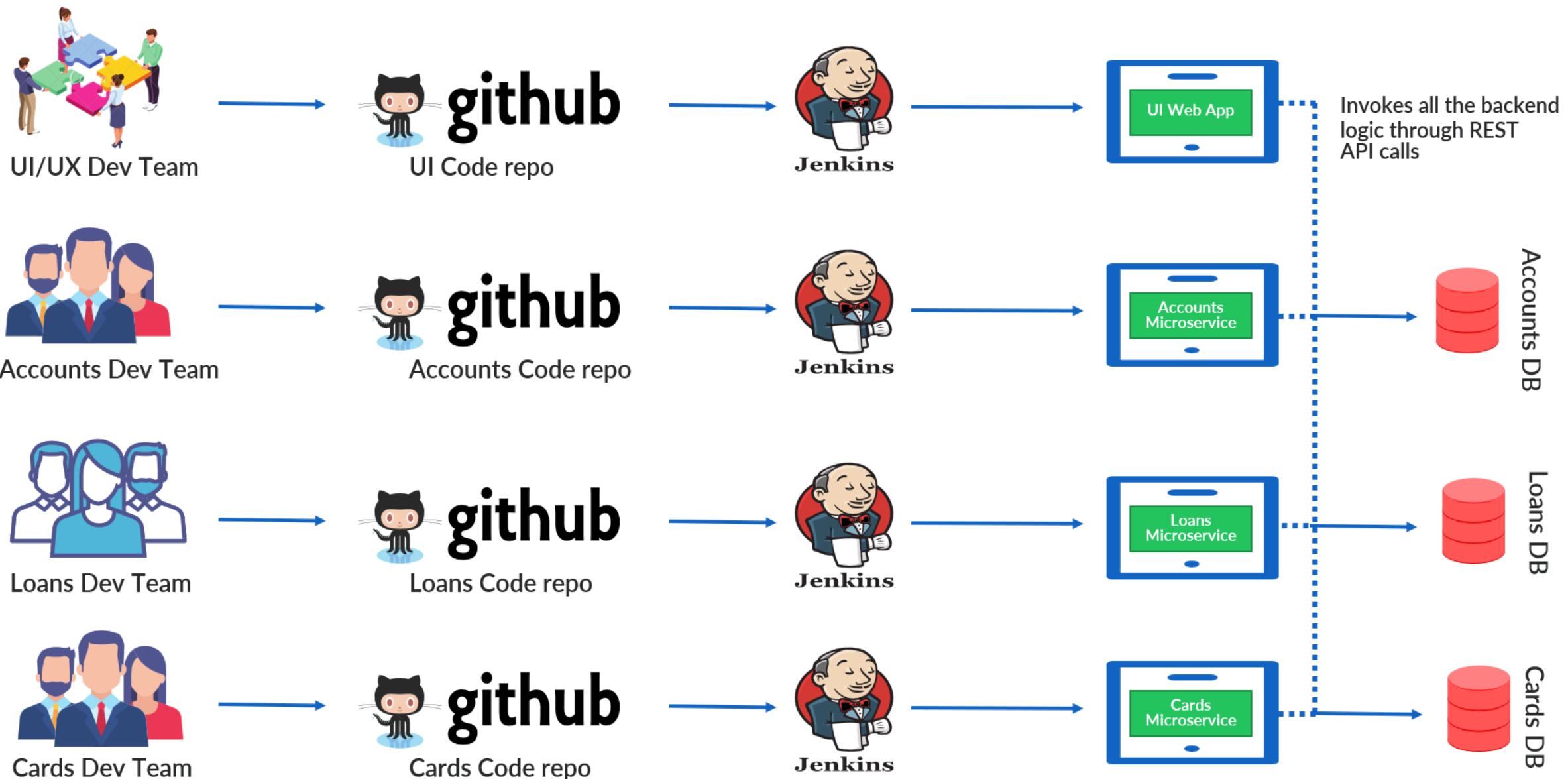
### Cons

- Complex management
- High investment costs
- Extra overload

# MICROSERVICES ARCHITECTURE

SAMPLE BANK APPLICATION

eazy  
bytes

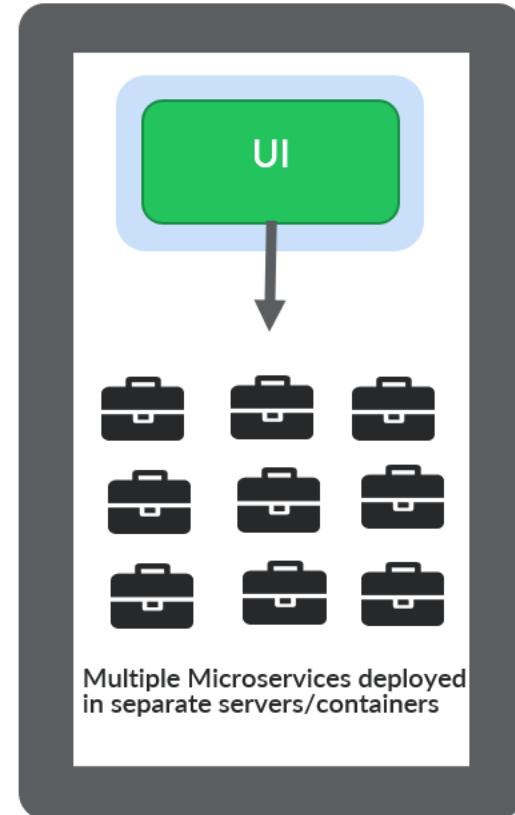


# MICROSERVICES ARCHITECTURE

## PROS & CONS

eazy  
bytes

### MICROSERVICES



#### Pros

- Easy to develop, test, and deploy
- Increased agility
- Ability to scale horizontally
- Parallel development

#### Cons

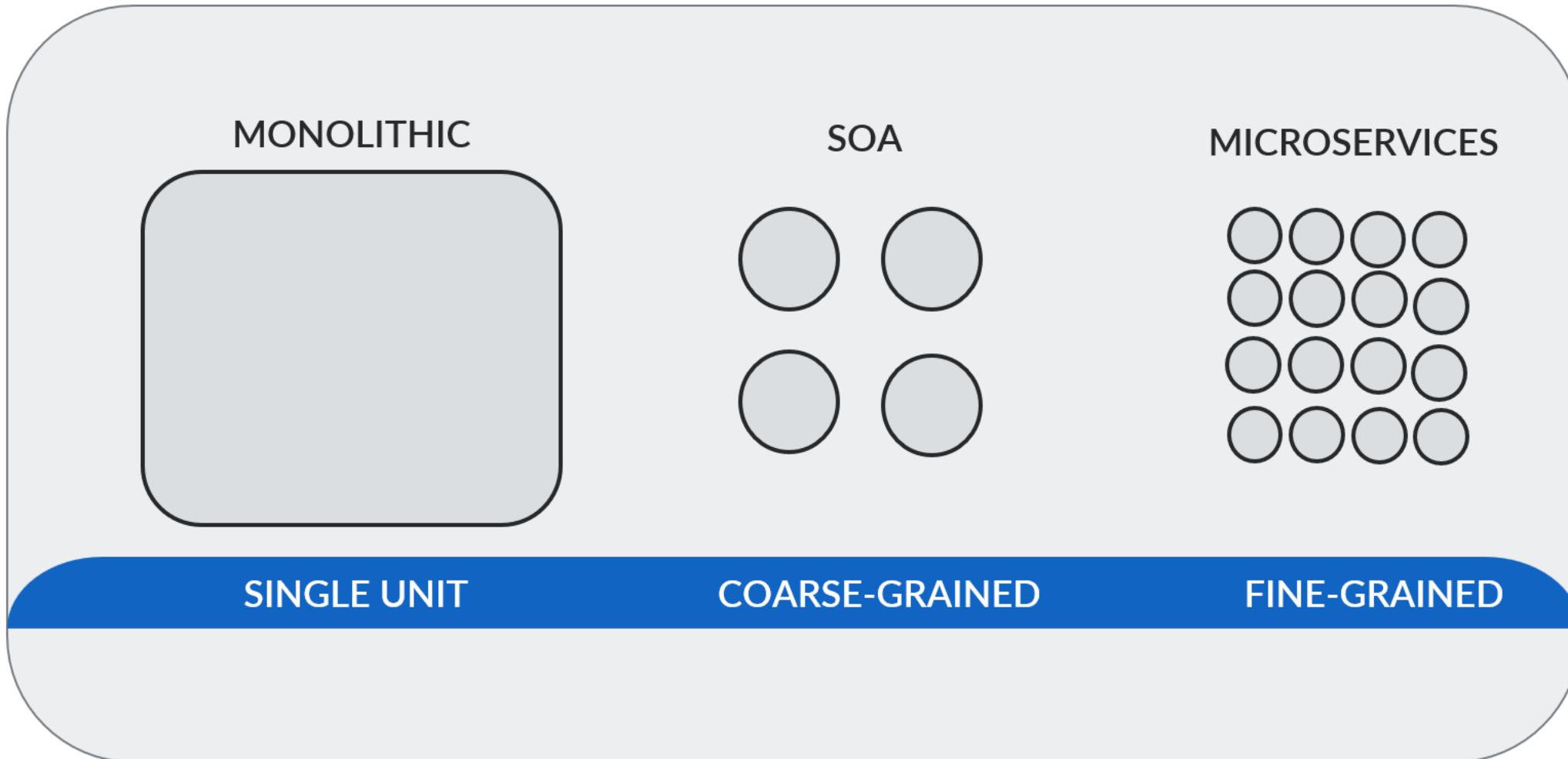
- Complexity
- Infrastructure overhead
- Security concerns



# MONOLITHIC VS SOA VS MICROSERVICES

COMPARISON

eazy  
bytes



# MONOLITHIC VS SOA VS MICROSERVICES

COMPARISON

eazy  
bytes

FEATURES	MONOLITHIC	SOA	MICROSERVICES
Parallel Development			
Agility			
Scalability			
Usability			
Complexity & Operational overhead			
Security Concerns & Performance			

# WHAT ARE MICROSERVICES?

DEFINITION OF MICROSERVICES

eazy  
bytes



*"Microservices is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, built around business capabilities and independently deployable by fully automated deployment machinery."*

- From Article by James Lewis and Martin Fowler's

# WHY SPRING FOR MICROSERVICES?

## WHY SPRING IS THE BEST FRAMEWORK TO BUILD MICROSERVICE

eazy  
bytes

Spring is the most popular development framework for building java-based web applications & services. From the Day1, Spring is working on building our code based on principles like loose coupling by using dependency injection. Over the years, Spring framework is evolving by staying relevant in the market.

**01**

Building small services using SpringBoot is super easy & fast

**02**

Spring Cloud provides tools for dev to quickly build some of the common patterns in Microservices

**03**

Provides production ready features like metrics, security, embedded servers

**04**

Spring Cloud makes deployment of microservices to cloud easy

**05**

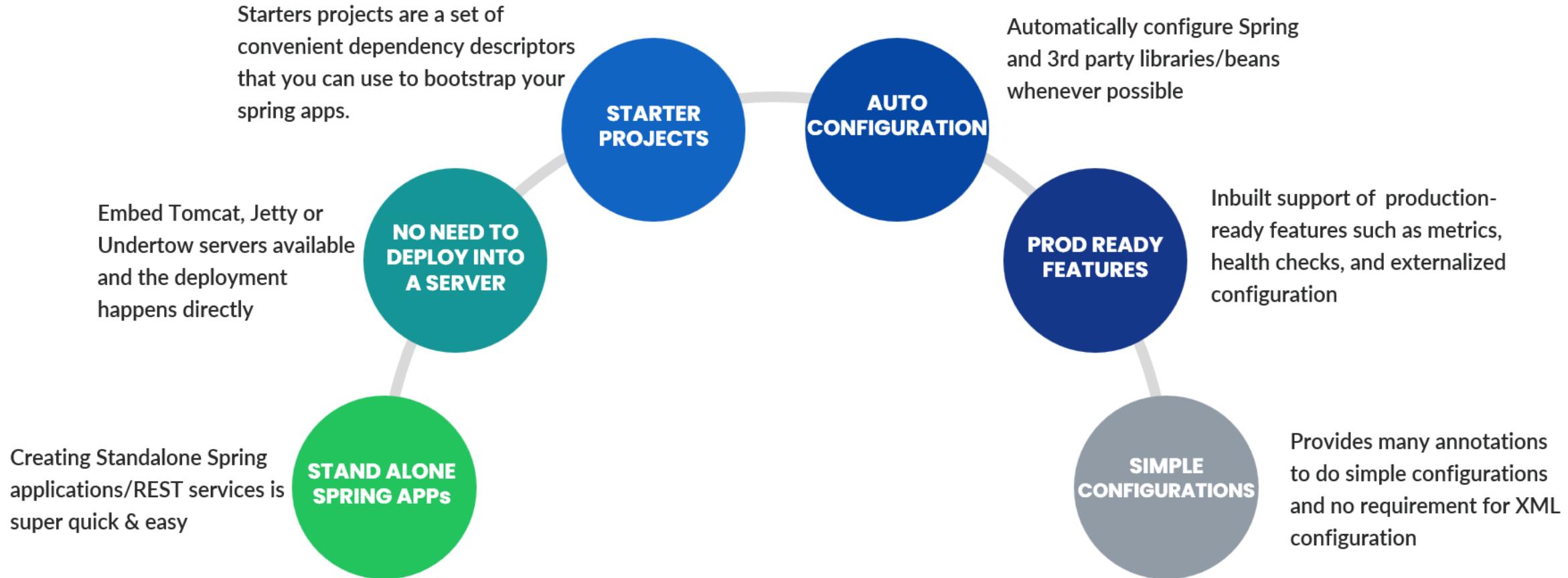
There is a large community of Spring developers who can help & adapt easily



# WHAT IS SPRING BOOT?

## USING SPRING BOOT FOR MICROSERVICES DEVELOPMENT

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

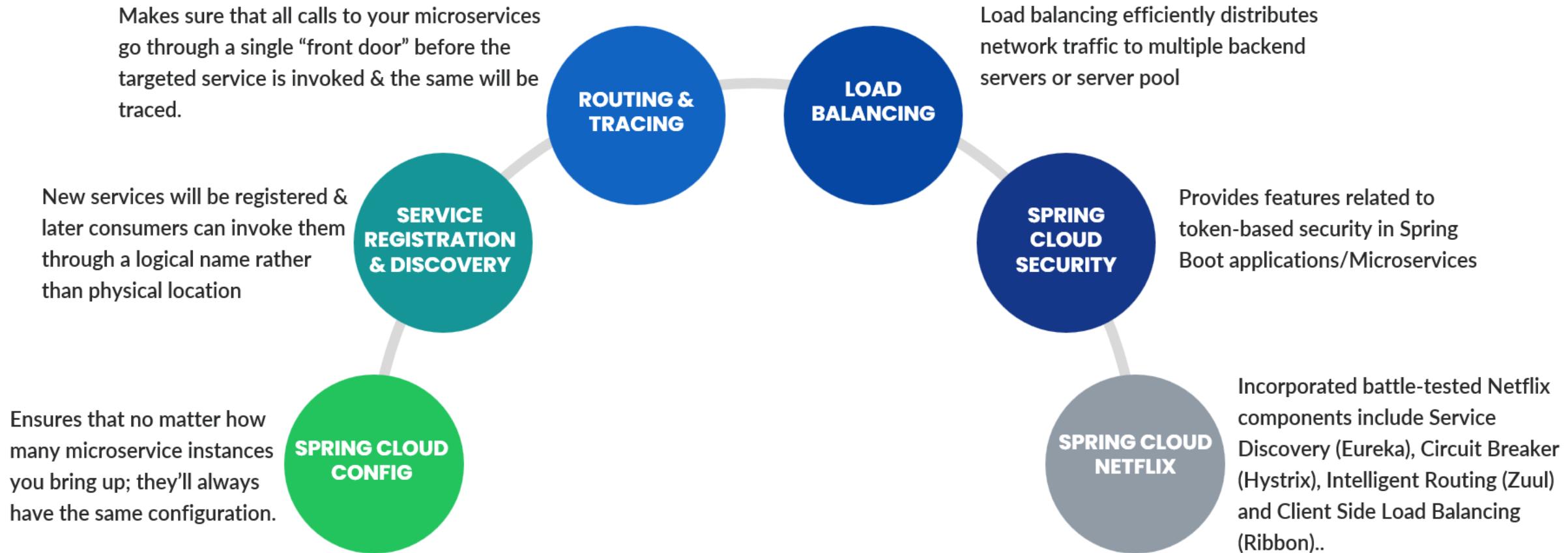


# WHAT IS SPRING CLOUD?

USING SPRING CLOUD FOR MICROSERVICES DEVELOPMENT

eazy  
bytes

Spring Cloud provides tools for developers to quickly build some of the common patterns of Microservices



# CHALLENGE 1 WITH MICROSERVICES

RIGHT SIZING & IDENTIFYING SERVICE BOUNDARIES

eazy  
bytes

- One of the most challenging aspects of building a successful microservices system is the identification of proper microservice boundaries and defining the size of each microservice
- Below are the most common followed approaches in the industry,
  - ✓ **Domain-Driven Sizing** - Since many of our modifications or enhancements driven by the business needs, we can size/define boundaries of our microservices that are closely aligned with Domain-Driven design & Business capabilities. But this process takes lot of time and need good domain knowledge.
  - ✓ **Event Storming Sizing** - Conducting an interactive fun session among various stake holder to identify the list of important events in the system like 'Completed Payment', 'Search for a Product' etc. Based on the events we can identify 'Commands', 'Reactions' and can try to group them to a domain-driven services.

Reference for Event Storming Session : <https://www.lucidchart.com/blog/ddd-event-storming>

# RIGHT SIZING MICROSERVICES

## IDENTIFYING SERVICE BOUNDARIES

eazy  
bytes

Now let's take an example of a Bank application that needs to be migrated/build based on a microservices architecture and try to do sizing of the services.

Saving Account & Trading Account



Cards & Loans



NOT CORRECT SIZING AS WE CAN SEE  
INDEPENDENT MODULES LIKE CARDS & LOANS  
CLUBBED TOGETHER

THIS MIGHT BE THE MOST REASONABLE CORRECT  
SIZING AS WE CAN SEE ALL INDEPENDENT  
MODULES HAVE SEPARATE SERVICE MAINTAINING  
LOOSELY COUPLED & HIGHLY COHESIVE

Saving Account



Cards



Trading Account



Loans



Saving Account



Debit Card



Home Loan



Trading Account



Credit Card



Vehicle Loan



Personal Loan



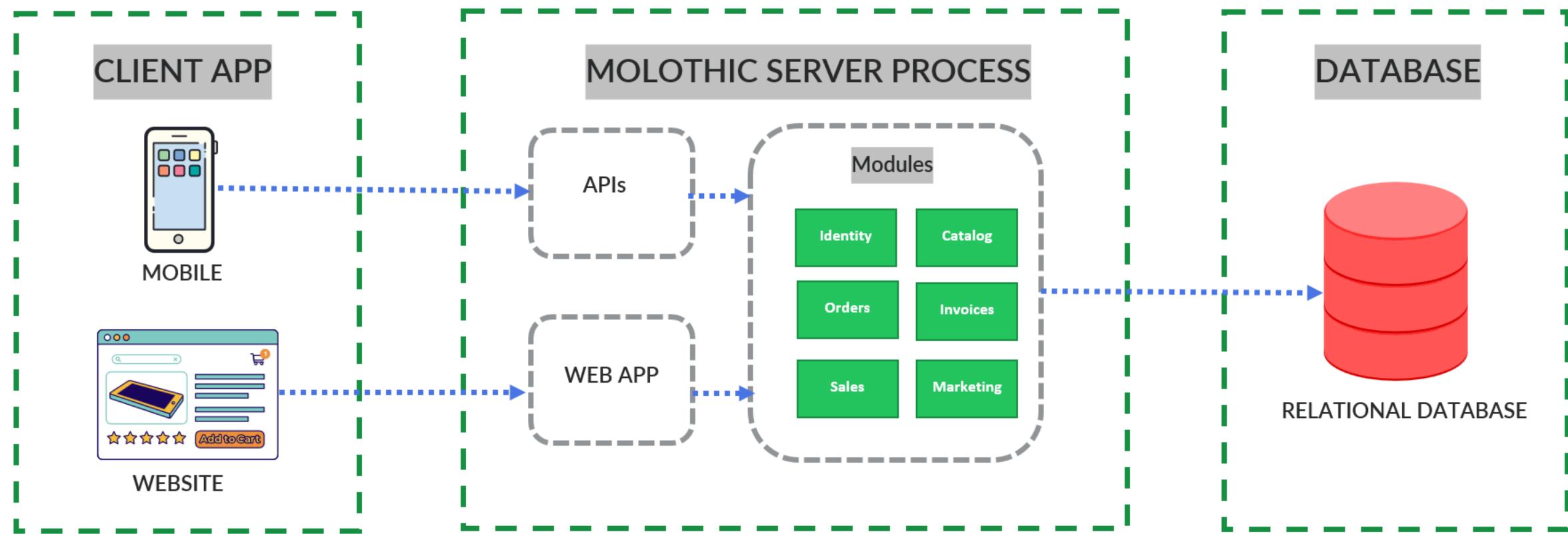
NOT CORRECT SIZING AS WE CAN SEE TOO  
MANY SERVICES UNDER LOANS & CARDS

# MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy  
bytes

Now let's take a scenario where an E-Commerce startup is following monolithic architecture and try to understand what's the challenges with it.



# MONOLOTHIC TO MICROSERVICES

## Migration UseCase

eazy  
bytes

Problem that E-Commerce team is facing due to traditional monolithic design

### Initial Days

- It is straightforward to build, test, deploy, troubleshoot and scale during the launch and when the team size is less

Later after few days the app/site is a super hit and started evolving a lot. Now team has below problems,

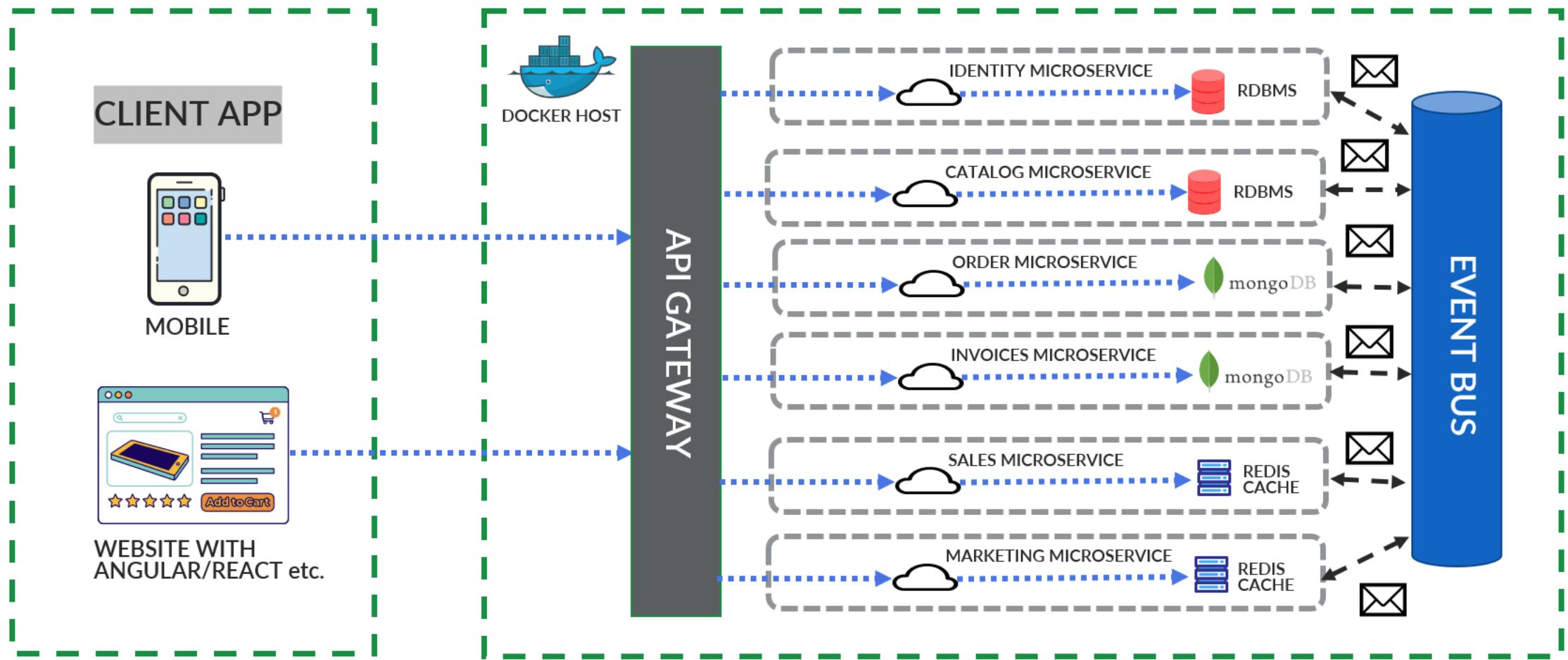
- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as possible and requires a full deployment of the entire application.
- One unstable component can crash the entire system.
- New technologies and frameworks aren't an option.
- It's difficult to maintain small isolated teams and implement agile delivery methodologies.

# MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy  
bytes

So the Ecommerce company decided and adopted the below cloud-native design by leveraging Microservices architecture to make their life easy and less risk with the continuous changes.



# CHALLENGE 2 WITH MICROSERVICES

## DEPLOYMENT, PORTABILITY & SCALABILITY

eazy  
bytes



### DEPLOYMENT

How do we deploy all the tiny 100s of microservices with less effort & cost?



### PORTABILITY

How do we move our 100s of microservices across environments with less effort, configurations & cost?



### SCALABILITY

How do we scale our applications based on the demand on the fly with minimum effort & cost?

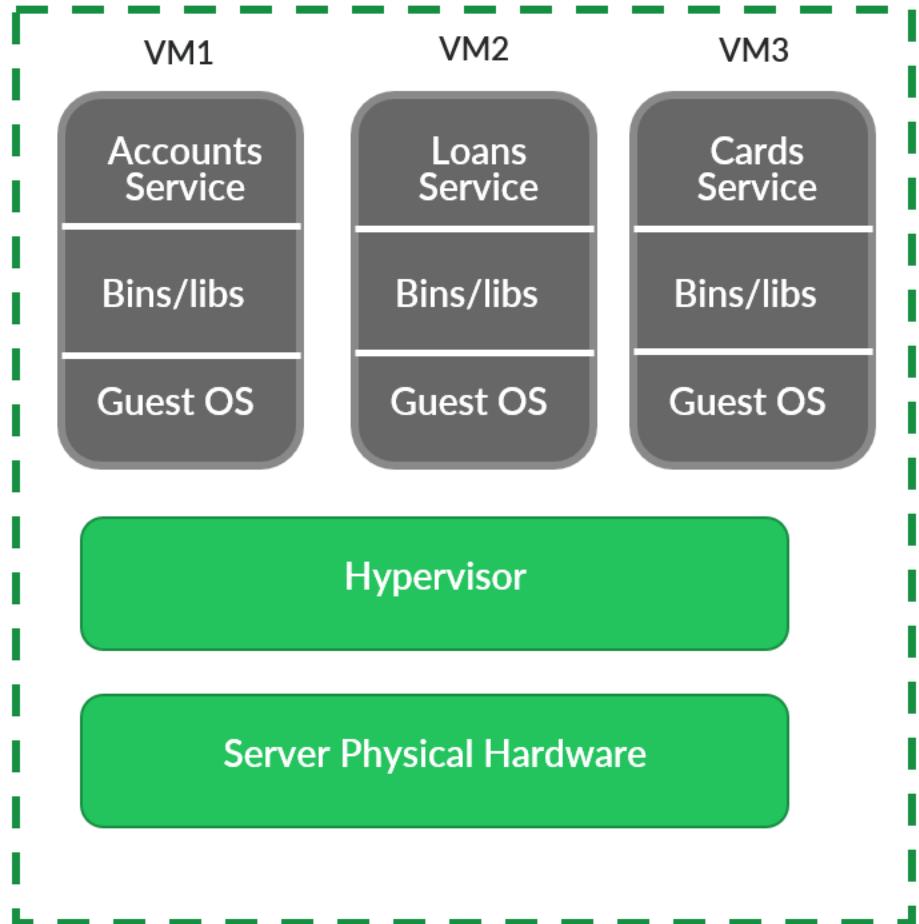


# CONTAINERIZATION TECHNOLOGY

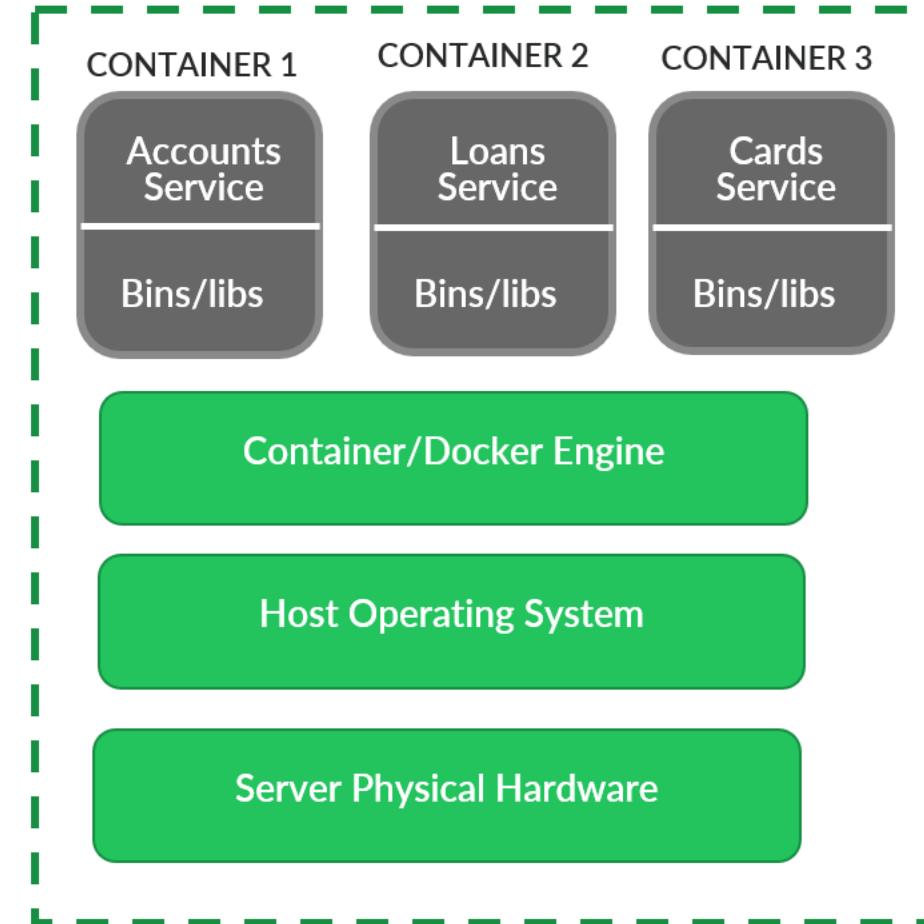
eazy  
bytes

USING DOCKER

## VIRTUAL MACHINES



## CONTAINERS



Main differences between virtual machines and containers. Containers don't need the Guest Os nor the hypervisor to assign resources; instead, they use the container engine.

# INTRO TO DOCKER

WHAT ARE CONTAINERS & DOCKER?

eazy  
bytes

## What is a container ?

A container is a loosely isolated environment that allows us to build and run software packages. These software packages include the code and all dependencies to run applications quickly and reliably on any computing environment. We call these packages container images.

## What is software containerization?

Software containerization is an OS virtualization method that is used to deploy and run containers without using a virtual machine (VM). Containers can run on physical hardware, in the cloud, VMs, and across multiple OSs.

## What is Docker?

Docker is one of the tools that used the idea of the isolated resources to create a set of tools that allows applications to be packaged with all the dependencies installed and ran wherever wanted.

# INTRO TO DOCKER

eazy  
bytes

## DOCKER ARCHITECTURE

### DOCKER CLIENT

Docker Remote API

We can issue commands to Docker Daemon using either CLI or APIs

Docker CLI

### DOCKER HOST/SERVER

Using Docker Daemon we can create and manage the docker images

Docker Daemon

#### CONTAINERS

Container 1

Container 2

Container 3

#### IMAGES

Image of App1

Image of App2

### DOCKER REGISTRY

Docker Hub

The docker images can be maintained and pulled from the docker hub or private registries.

Private Registry

# CLOUD-NATIVE APPLICATIONS

## INTRODUCTION

eazy  
bytes

- Cloud-native applications are a collection of small, independent, and loosely coupled services. They are designed to deliver well-recognized business value, like the ability to rapidly incorporate user feedback for continuous improvement. Its goal is to deliver apps users want at the pace a business needs.
- If an app is "cloud-native," it's specifically designed to provide a consistent development and automated management experience across private, public, and hybrid clouds. So it's about how applications are created and deployed, not where.
- When creating cloud-native applications, the developers divide the functions into microservices, with scalable components such as containers in order to be able to run on several servers. These services are managed by virtual infrastructures through DevOps processes with continuous delivery workflows. It's important to understand that these types of applications do not require any change or conversion to work in the cloud and are designed to deal with the unavailability of downstream components.

# CLOUD-NATIVE APPLICATIONS

eazy  
bytes

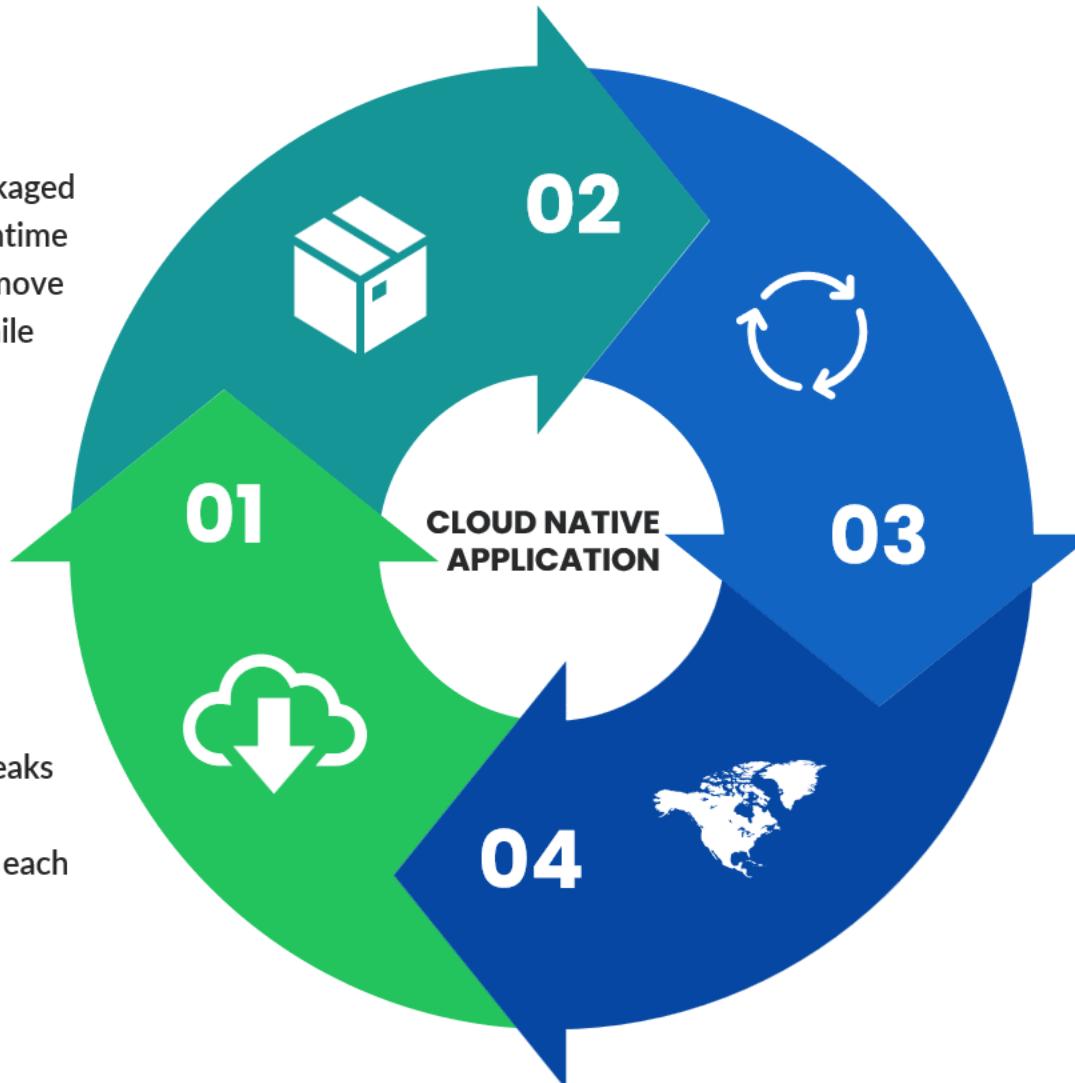
## PRINCIPLES OF CLOUD-NATIVE APPLICATIONS

### CONTAINERS

Containers allow apps to be packaged and isolated with their entire runtime environment, making it easy to move them between environments while retaining full functionality.

### MICROSERVICE

A microservices architecture breaks apps down into their smallest components, independent from each other.



### DEVOPS

DevOps is an approach to culture, automation, and platform design intended to deliver increased business value and responsiveness.

### CONTINUOUS DELIVERY

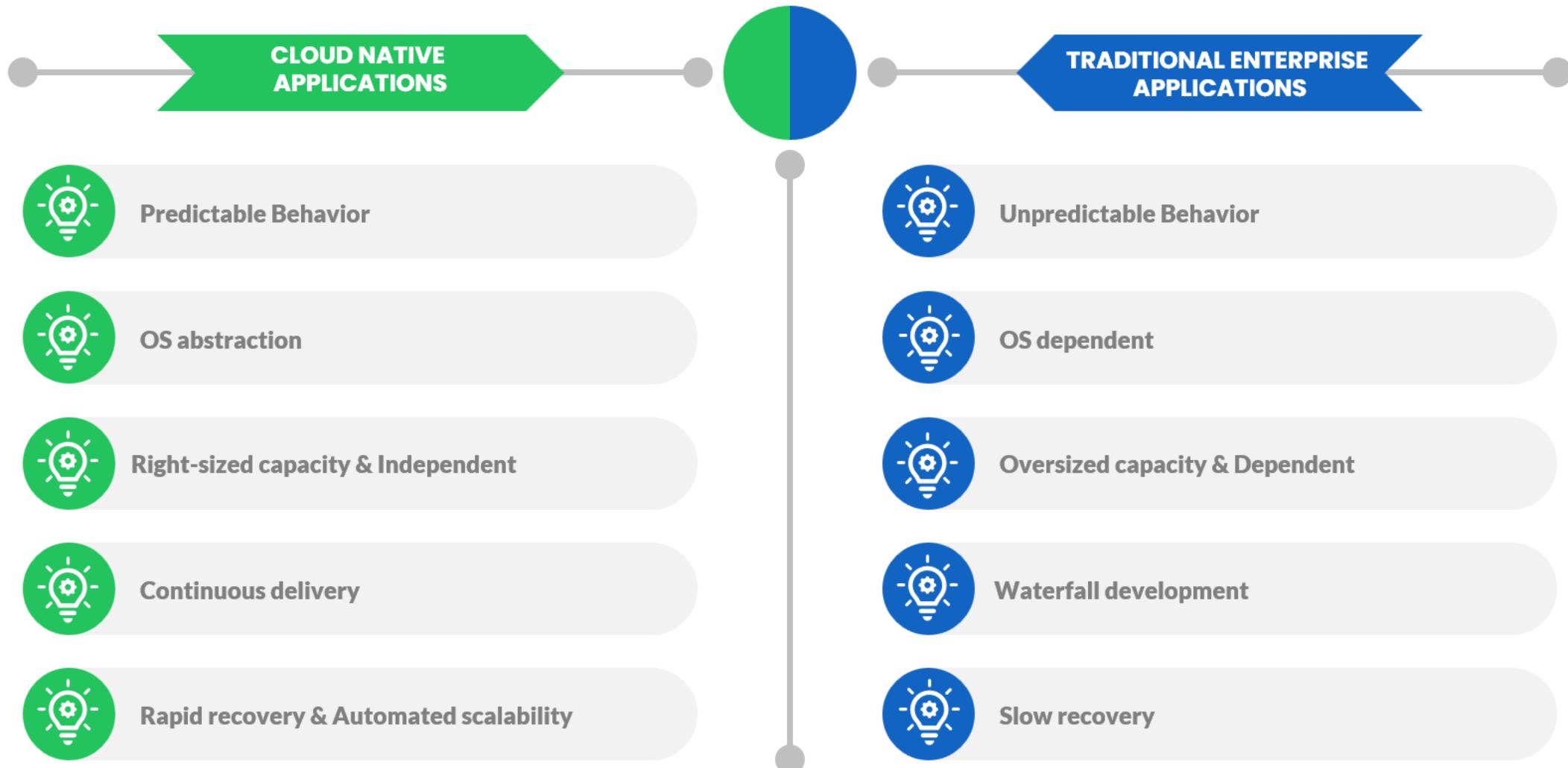
It's a software development practice in which the process of delivering software is automated to allow short-term deliveries into a production environment.

To build and develop cloud native applications (microservices), we need to follow the best practises mentioned in the twelve factor app methodology (<https://12factor.net/>)

# CLOUD-NATIVE APPLICATIONS

DIFFERENCE B/W CLOUD-NATIVE & TRADITIONAL APPS

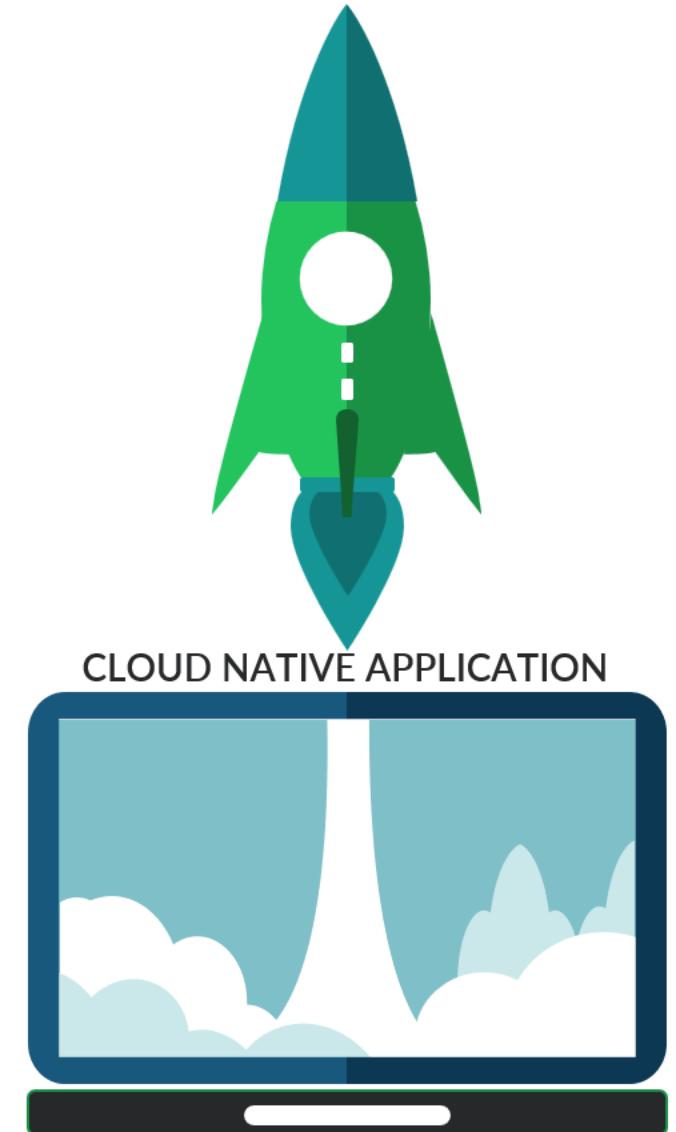
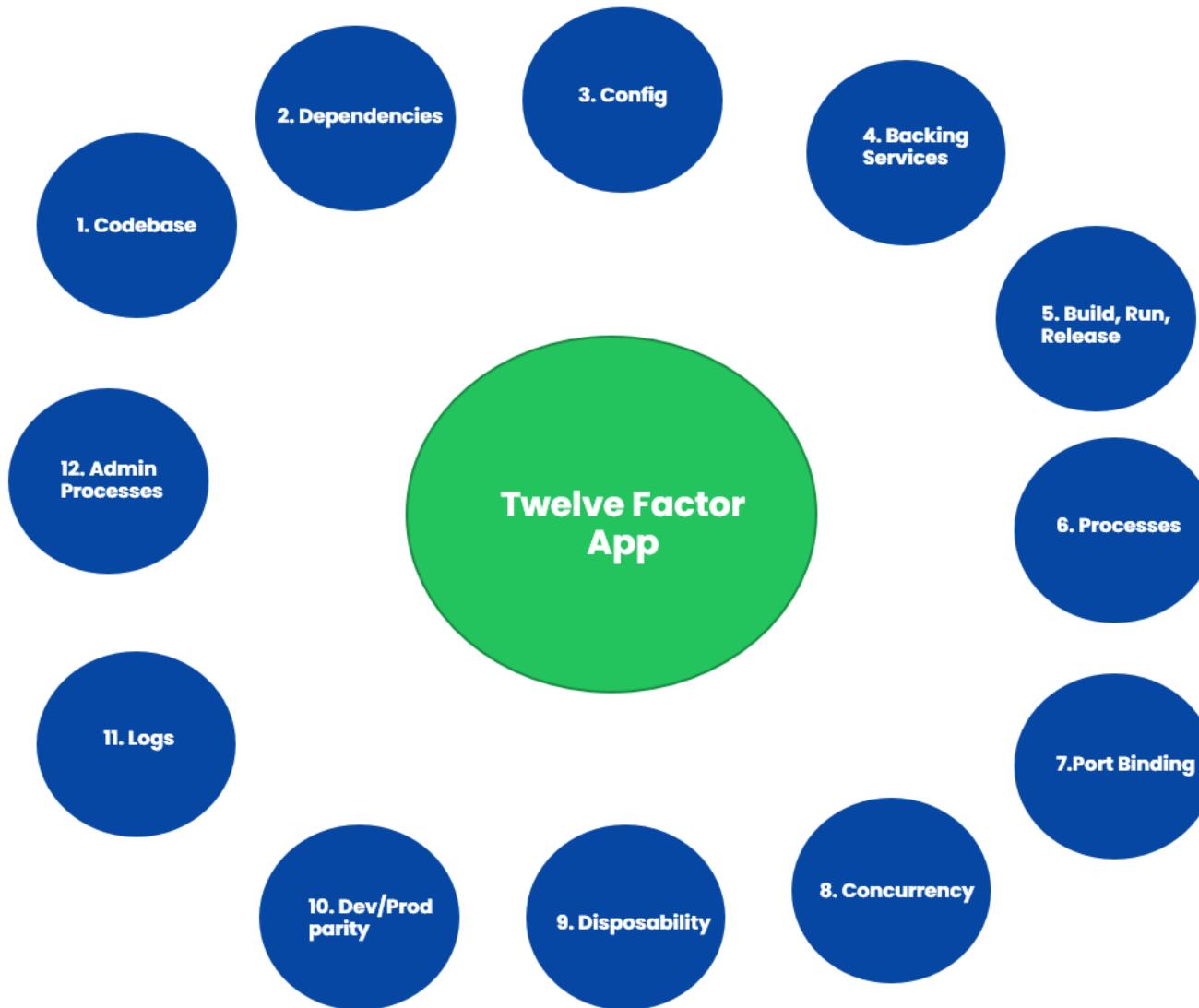
eazy  
bytes



# TWELVE FACTOR APP

## BEST PRACTICES

eazy  
bytes

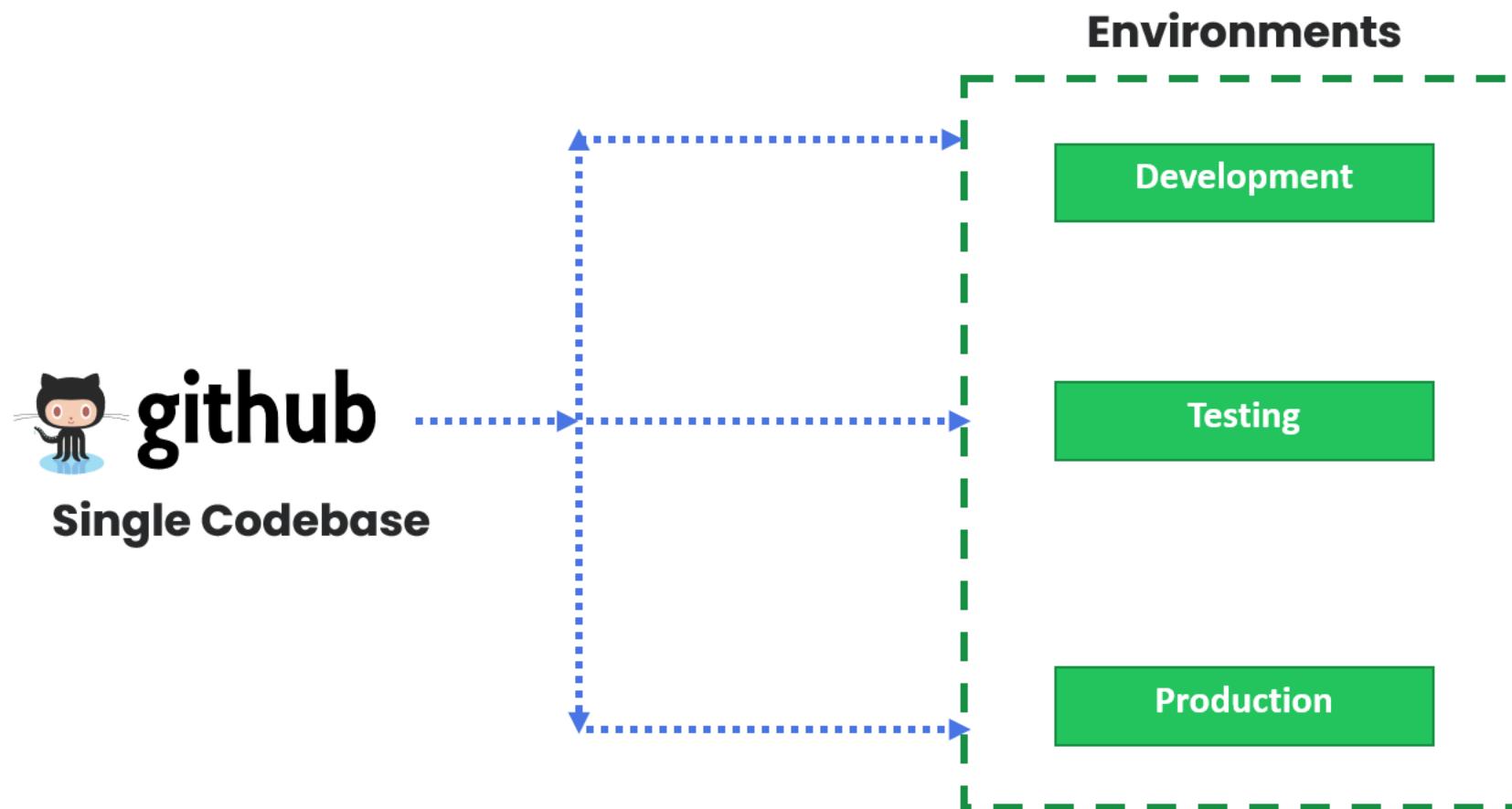


# TWELVE FACTOR APP

## 1. CODEBASE

eazy  
bytes

- Each microservice should have a single codebase, managed in source control. The code base can have multiple instances of deployment environments such as development, testing, staging, production, and more but is not shared with any other microservice.

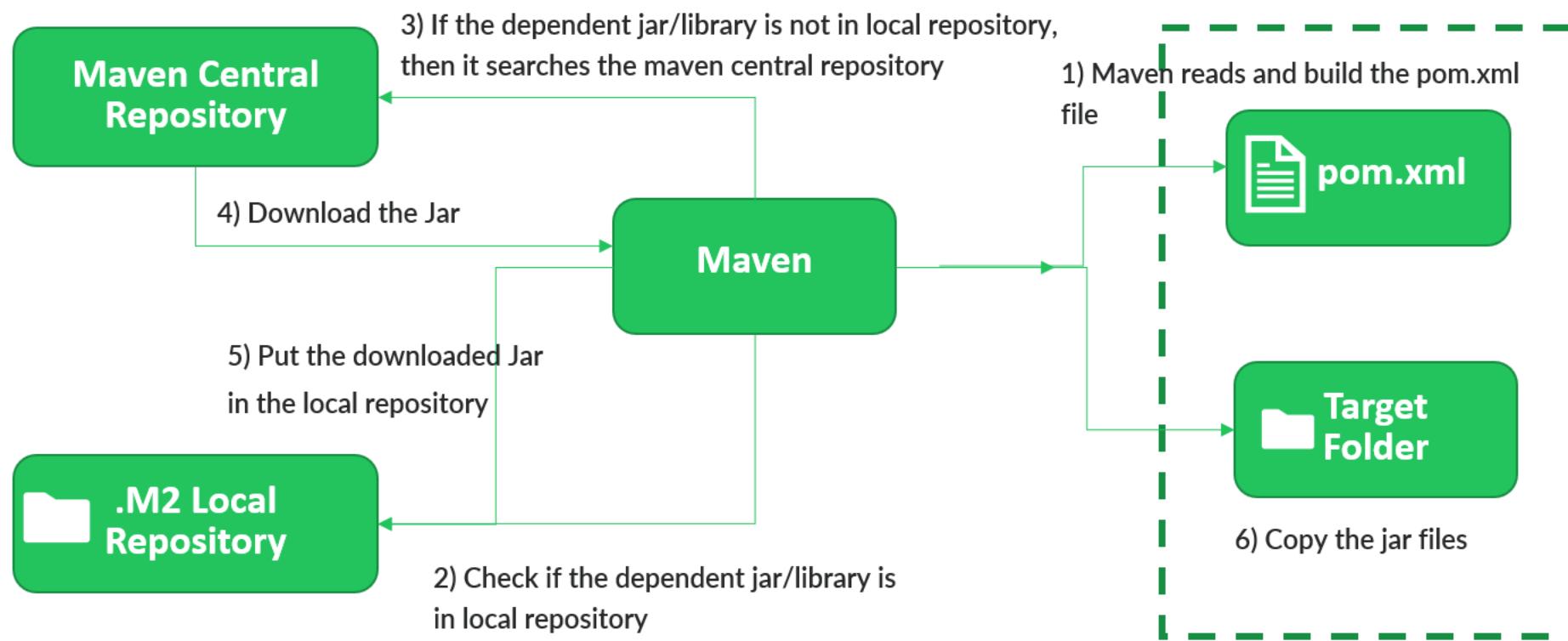


# TWELVE FACTOR APP

## 2. DEPENDENCIES

eazy  
bytes

- Explicitly declare the dependencies your application uses through build tools such as Maven, Gradle (Java). Third-party JAR dependence should be declared using their specific versions number. This allows your microservice to always be built using the same version of libraries.
- A twelve-factor app never relies on implicit existence of system-wide packages.

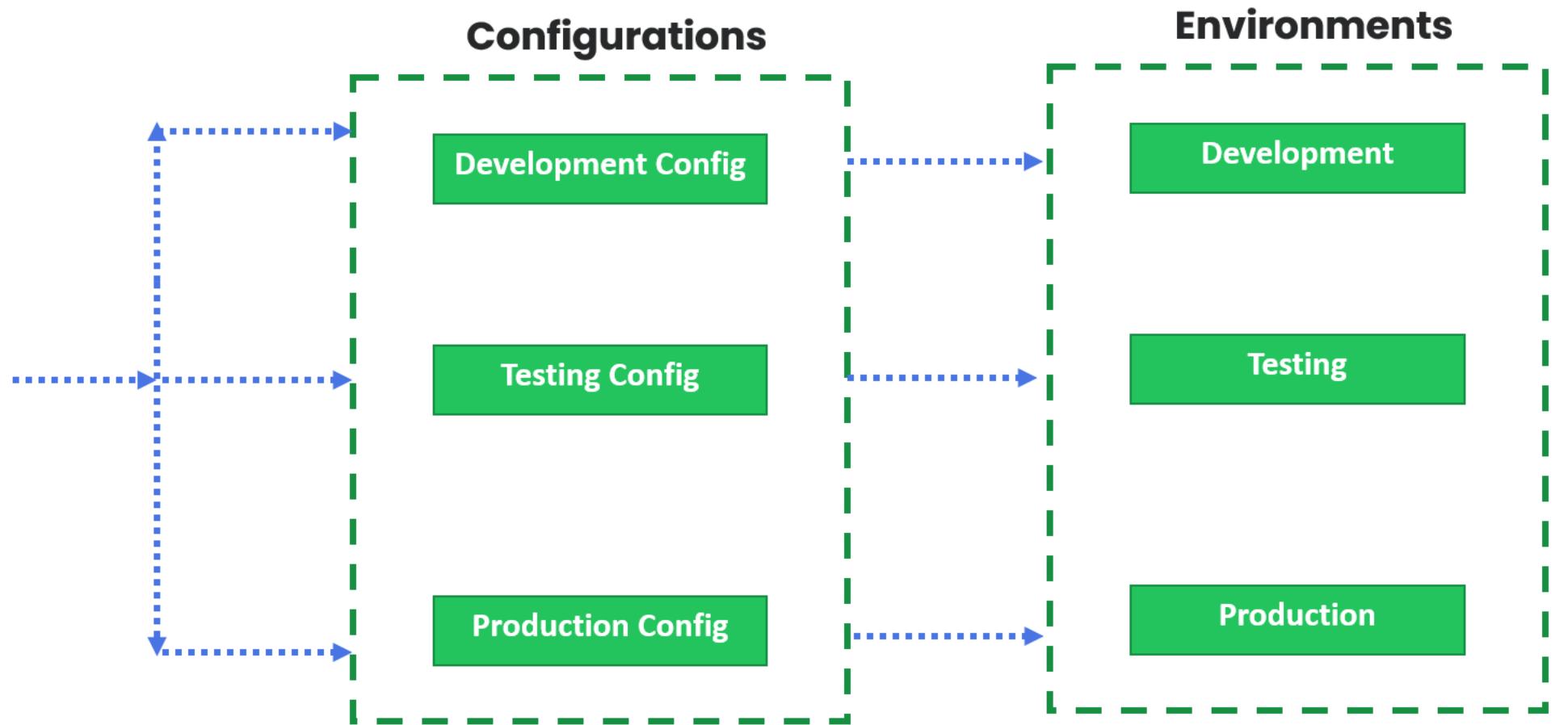


# TWELVE FACTOR APP

## 3. CONFIG

eazy  
bytes

- Store environment-specific configuration independently from your code. Never add embedded configurations to your source code; instead, maintain your configuration completely separated from your deployable microservice. If we keep the configuration packaged within the microservice, we'll need to redeploy each of the hundred instances to make the change.

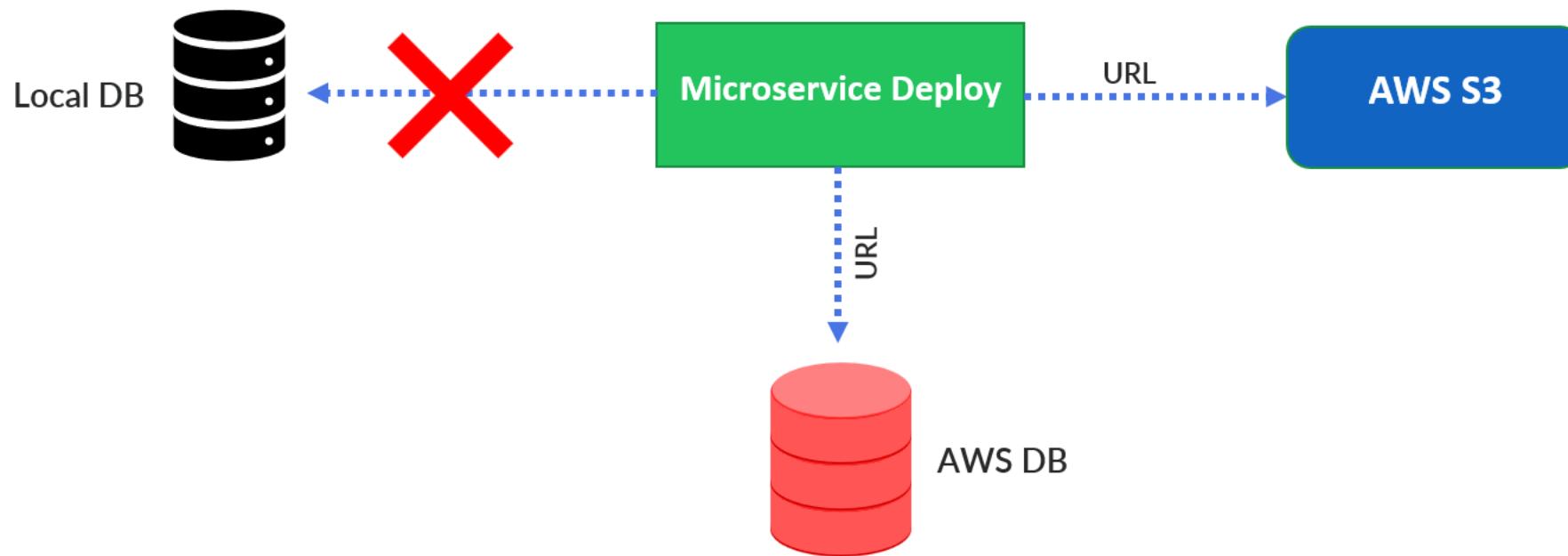


# TWELVE FACTOR APP

## 4. BACKING SERVICES

eazy  
bytes

- Backing Services best practice indicates that a microservices deploy should be able to swap between local connections to third party without any changes to the application code.
- In the below example, we can see that a local DB can be swapped easily to a third-party DB which is AWS DB here with out any code changes.

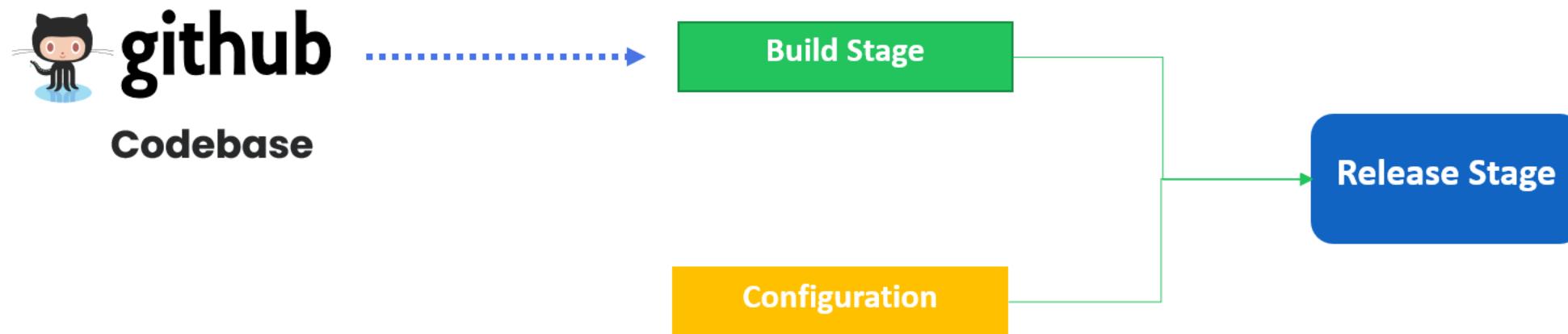


# TWELVE FACTOR APP

## 5. BUILD, RELEASE, RUN

eazy  
bytes

- Keep your build, release, and run stages of deploying your application completely separated. We should be able to build microservices that are independent of the environment which they are running.

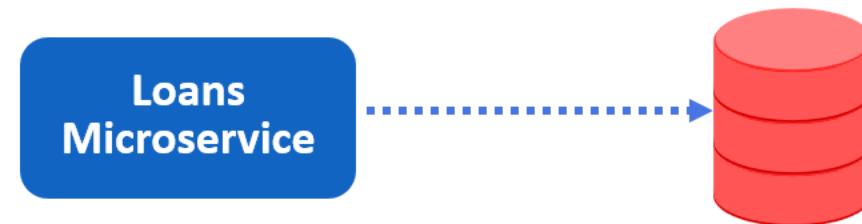


# TWELVE FACTOR APP

## 6. PROCESSES

eazy  
bytes

- Execute the app as one or more stateless processes. Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.
- Microservices can be killed and replaced at any time without the fear that a loss of a service-instance will result in data loss.



We can store the data of the loans  
microservice inside a SQL or NoSQL DB

# TWELVE FACTOR APP

## 7. PORT BINDING

eazy  
bytes

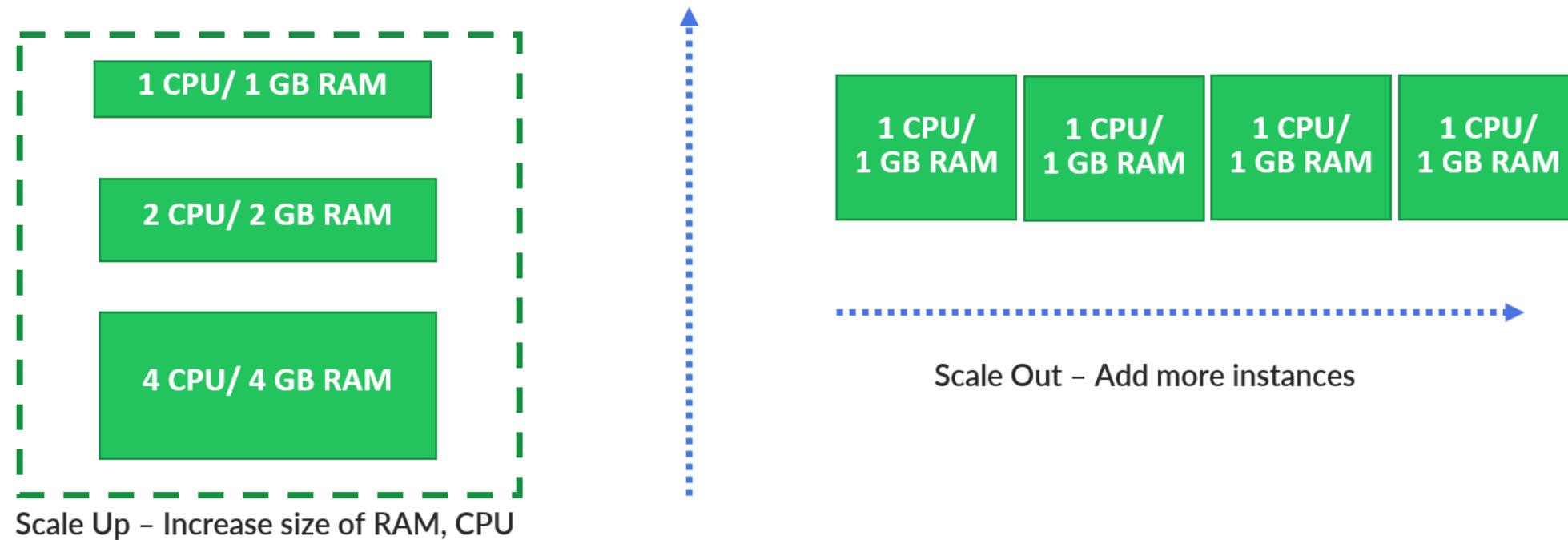
- Web apps are sometimes executed inside a webserver container. For example, PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat. But each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices.
- We will develop an application using Spring Boot. Spring Boot, apart from many other benefits, provides us with a default embedded application server. Hence, the JAR we generated earlier using Maven is fully capable of executing in any environment just by having a compatible Java runtime.

# TWELVE FACTOR APP

## 8. CONCURRENCY

eazy  
bytes

- Services scale out across a large number of small identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available.
- Vertical scaling (Scale up) refers to increase the hardware infrastructure (CPU, RAM). Horizontal scaling (Scale out) refers to adding more instances of the application. When you need to scale, launch more microservice instances and scale out and not up.



# TWELVE FACTOR APP

## 9. DISPOSABILITY

eazy  
bytes

- Service instances should be disposable, favoring fast startups to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement.
- For example, if one of the instances of the microservice is failing because of a failure in the underlying hardware, we can shut down that instance without affecting other microservices and start another one somewhere else if needed.

# TWELVE FACTOR APP

## 10. Dev/Prod parity

eazy  
bytes

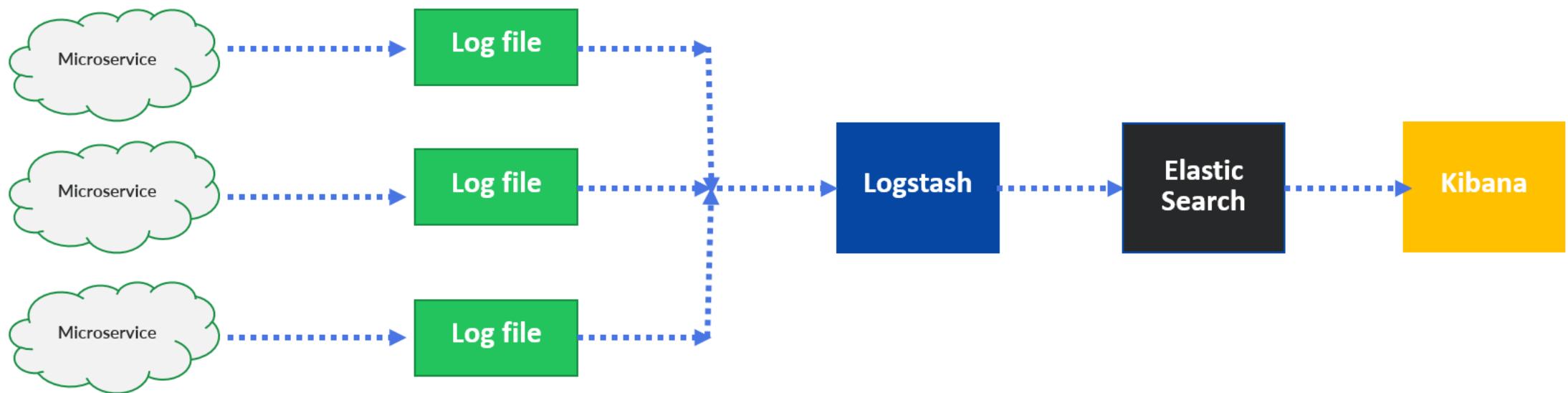
- Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.
- As soon as a code is committed, it should be tested and then promoted as quickly as possible from development all the way to production. This guideline is essential if we want to avoid deployment errors. Having similar development and production environments allows us to control all the possible scenarios we might have while deploying and executing our application.

# TWELVE FACTOR APP

## 11. Logs

eazy  
bytes

- Treat logs generated by microservices as event streams. As logs are written out, they should be managed by tools, such as Logstash(<https://www.elastic.co/products/logstash>) that will collect the logs and write them to a central location.
- The microservice should never be concerned about the mechanisms of how this happens, they only need to focus on writing the log entries into the stdout. We will discuss on how to provide an autoconfiguration for sending these logs to the ELK stack (Elasticsearch, Logstash and Kibana) in the coming sections.



# TWELVE FACTOR APP

## 12. Admin processes

eazy  
bytes

- Run administrative/management tasks as one-off processes. Tasks can include data cleanup and pulling analytics for a report. Tools executing these tasks should be invoked from the production environment, but separately from the application.
- Developers will often have to do administrative tasks related to their microservices like Data migration, clean up activities. These tasks should never be ad hoc and instead should be done via scripts that are managed and maintained through source code repository. These scripts should be repeatable and non-changing across each environment they're run against. It's important to have defined the types of tasks we need to take into consideration while running our microservice, in case we have multiple microservices with these scripts we are able to execute all of the administrative tasks without having to do it manually.

# CHALLENGE 3 WITH MICROSERVICES

## CONFIGURATION MANAGEMENT

eazy  
bytes



### SEPARATION OF CONFIGs/PROPERTIES

How do we separate the configurations/properties from the microservices so that same Docker image can be deployed in multiple envs.



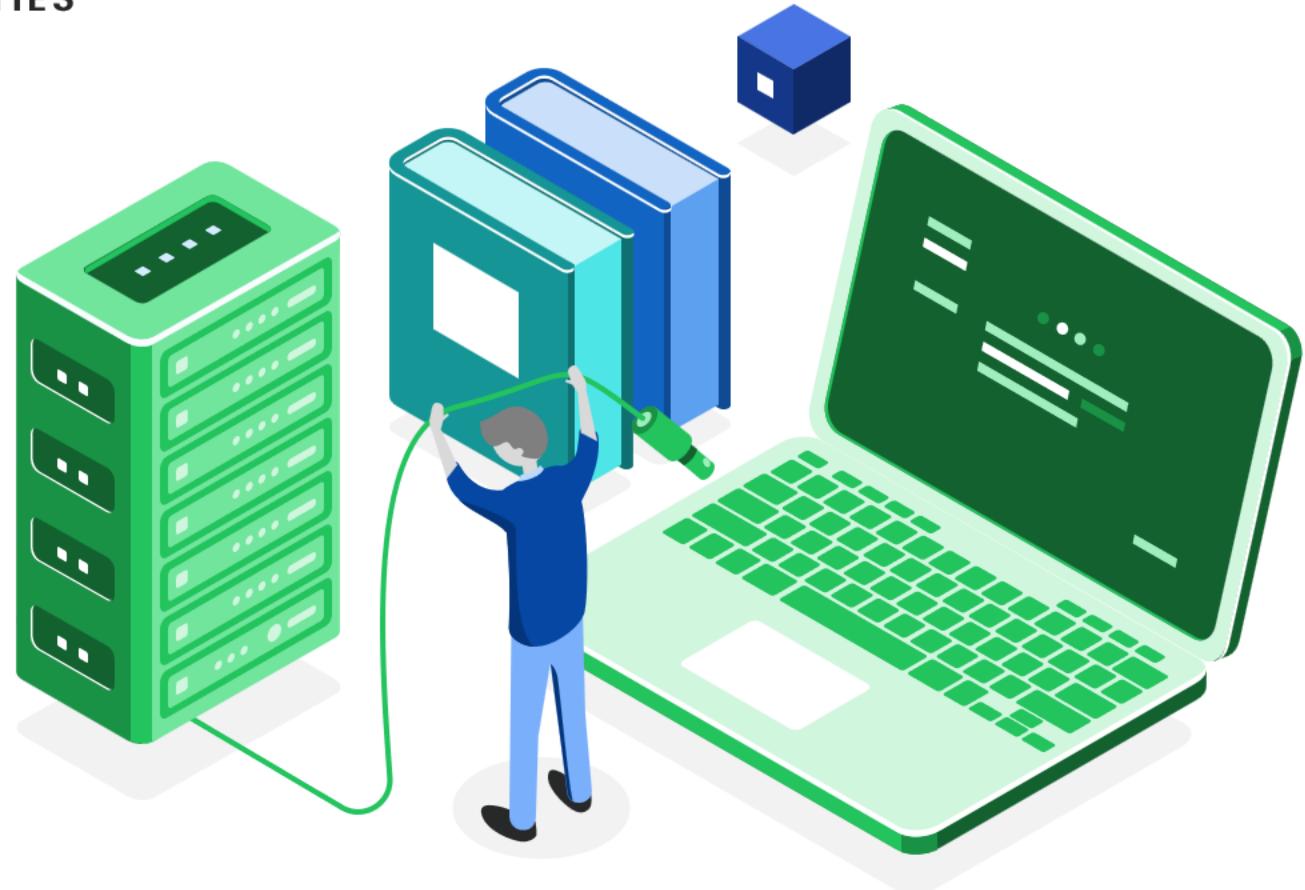
### INJECT CONFIGs/PROPERTIES

How do we inject configurations/properties that microservice needed during start up of the service



### MAINTAIN CONFIGs/PROPERTIES

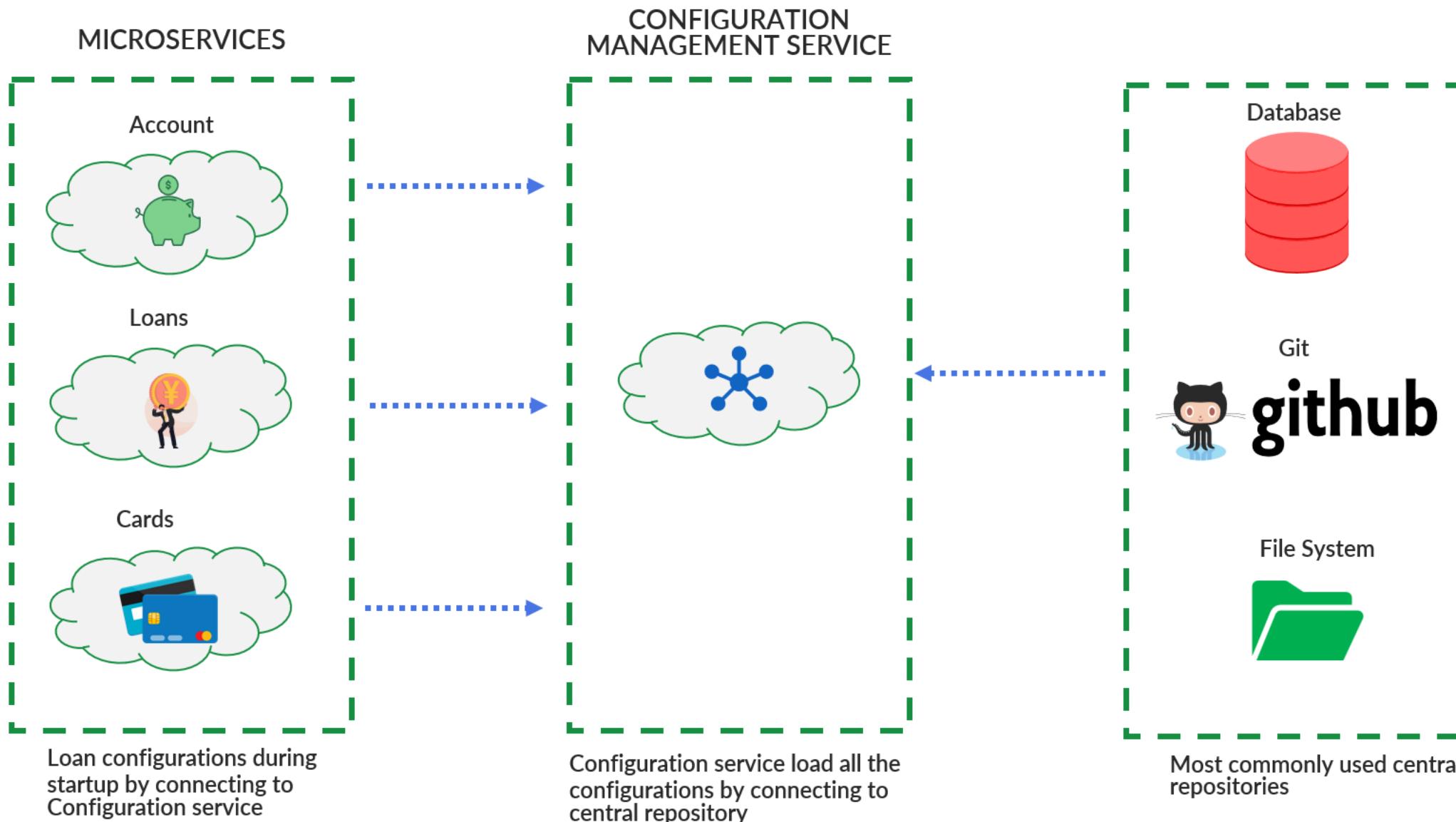
How do we maintain configurations/properties in a centralized repository along with versioning of them



# CONFIGURATION MANAGEMENT

ARCHITECTURE INSIDE MICROSERVICES

eazy  
bytes

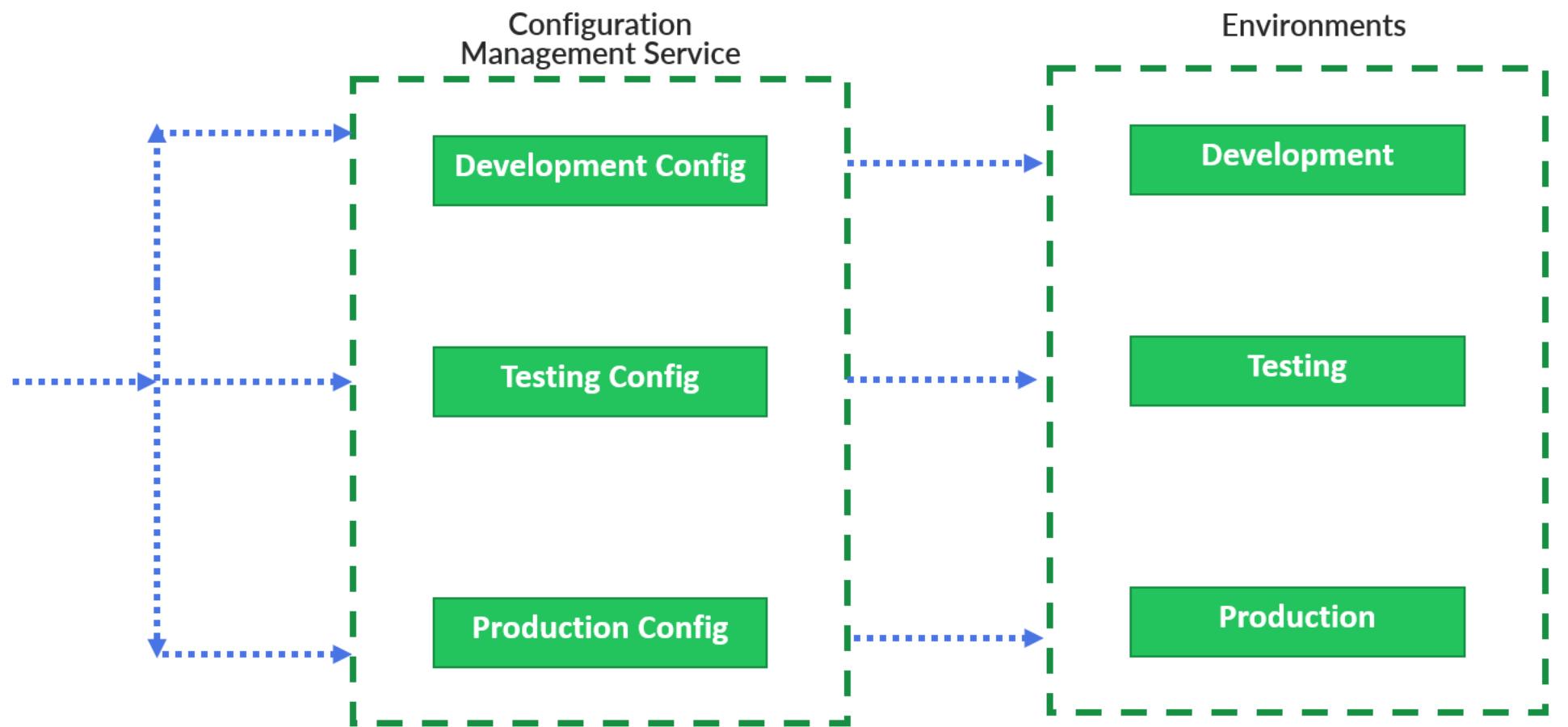


# SPRING CLOUD CONFIG

FOR CONFIGURATION MANAGEMENT IN MICROSERVICES

eazy  
bytes

- 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.



# SPRING CLOUD CONFIG

FOR CONFIGURATION MANAGEMENT IN MICROSERVICES

eazy  
bytes

## Spring Cloud Config Server features:

- *HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)*
- *Encrypt and decrypt property values*
- *Embeddable easily in a Spring Boot application using @EnableConfigServer*

## Config Client features (for Microservices):

- *Bind to the Config Server and initialize Spring Environment with remote property sources*
- *Encrypt and decrypt property values*

# CHALLENGE 4 WITH MICROSERVICES

## SERVICE DISCOVERY & REGISTRATION

eazy  
bytes



### HOW DO SERVICES LOCATE EACH OTHER INSIDE A NETWORK?

Each instance of a microservice exposes a remote API with its own host and port. How do other microservices & clients know about these dynamic endpoint URLs to invoke them? So where is my service?



### HOW DO NEW SERVICE INSTANCES ENTER INTO THE NETWORK?

If a microservice instance fails, new instances will be brought online to ensure constant availability. This means that the IP addresses of the instances can be constantly changing. So how does these new instances start serving to the clients?



### LOAD BALANCE, INFO SHARING B/W MICROSERVICE INSTANCES

How do we make sure to properly load balance b/w the multiple microservice instances especially a microservice is invoking another microservice? How do specific service information share across the network?



# SERVICE DISCOVERY & REGISTRATION

INSIDE MICROSERVICES NETWORK

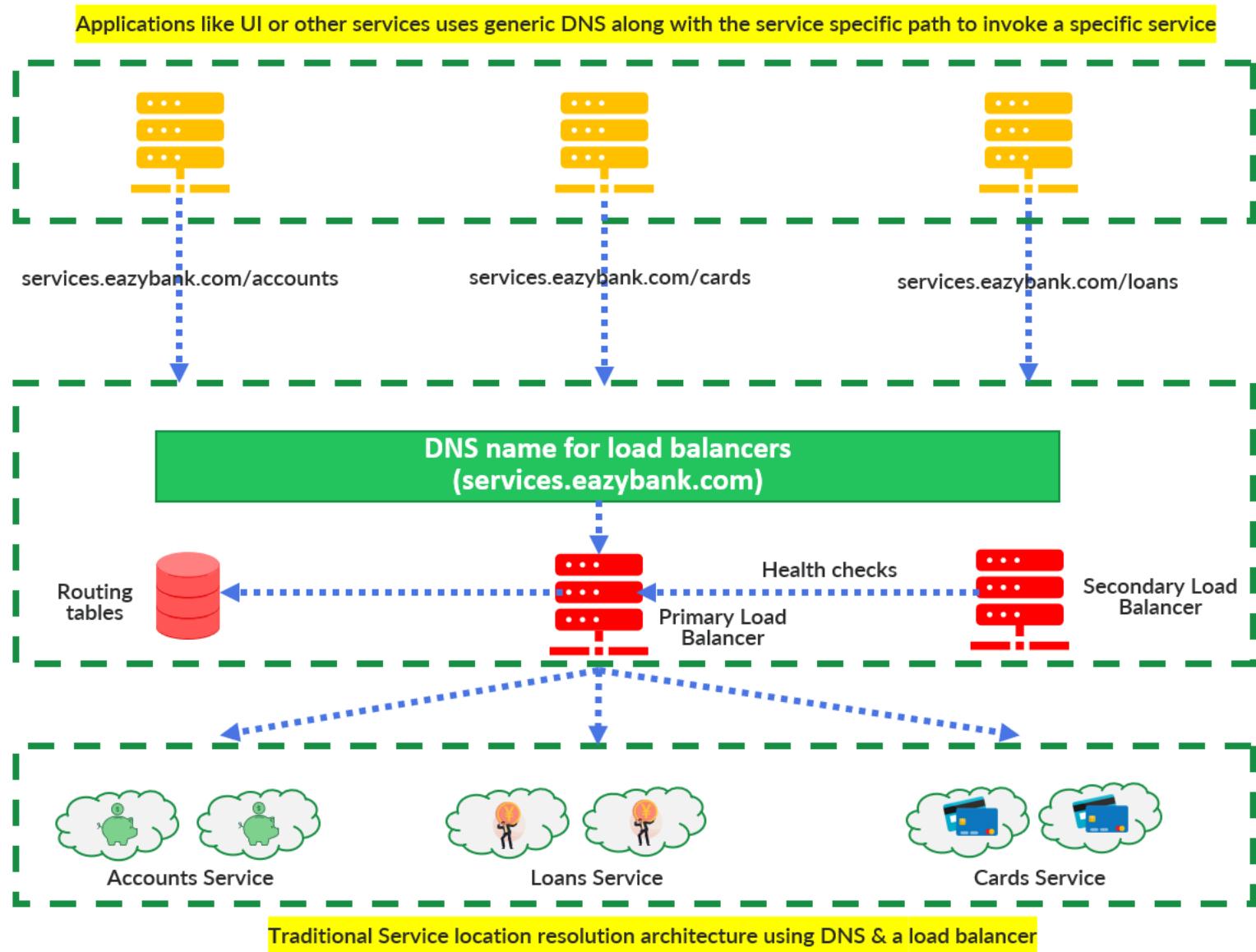
eazy  
bytes

- Service discovery & registrations deals with the problems about how microservices talk to each other, i.e. perform API calls.
- In a traditional network topology, applications have static network locations. Hence IP addresses of relevant external locations can be read from a configuration file, as these addresses rarely change.
- In a modern microservice architecture, knowing the right network location of an application is a much more complex problem for the clients as service instances might have dynamically assigned IP addresses. Moreover the number instances may vary due to autoscaling and failures.
- Microservices service discovery & registration is a way for applications and microservices to locate each other on a network. This includes,
  - ✓ A central server (or servers) that maintain a global view of addresses.
  - ✓ Microservices/clients that connect to the central server to register their address when they start & ready
  - ✓ Microservices/clients need to send their heartbeats at regular intervals to central server about their health

# WHY NOT TRADITIONAL LOAD BALANCERS

## FOR SERVICE DISCOVERY & REGISTRATION

eazy  
bytes



# WHY NOT TRADITIONAL LOAD BALANCERS

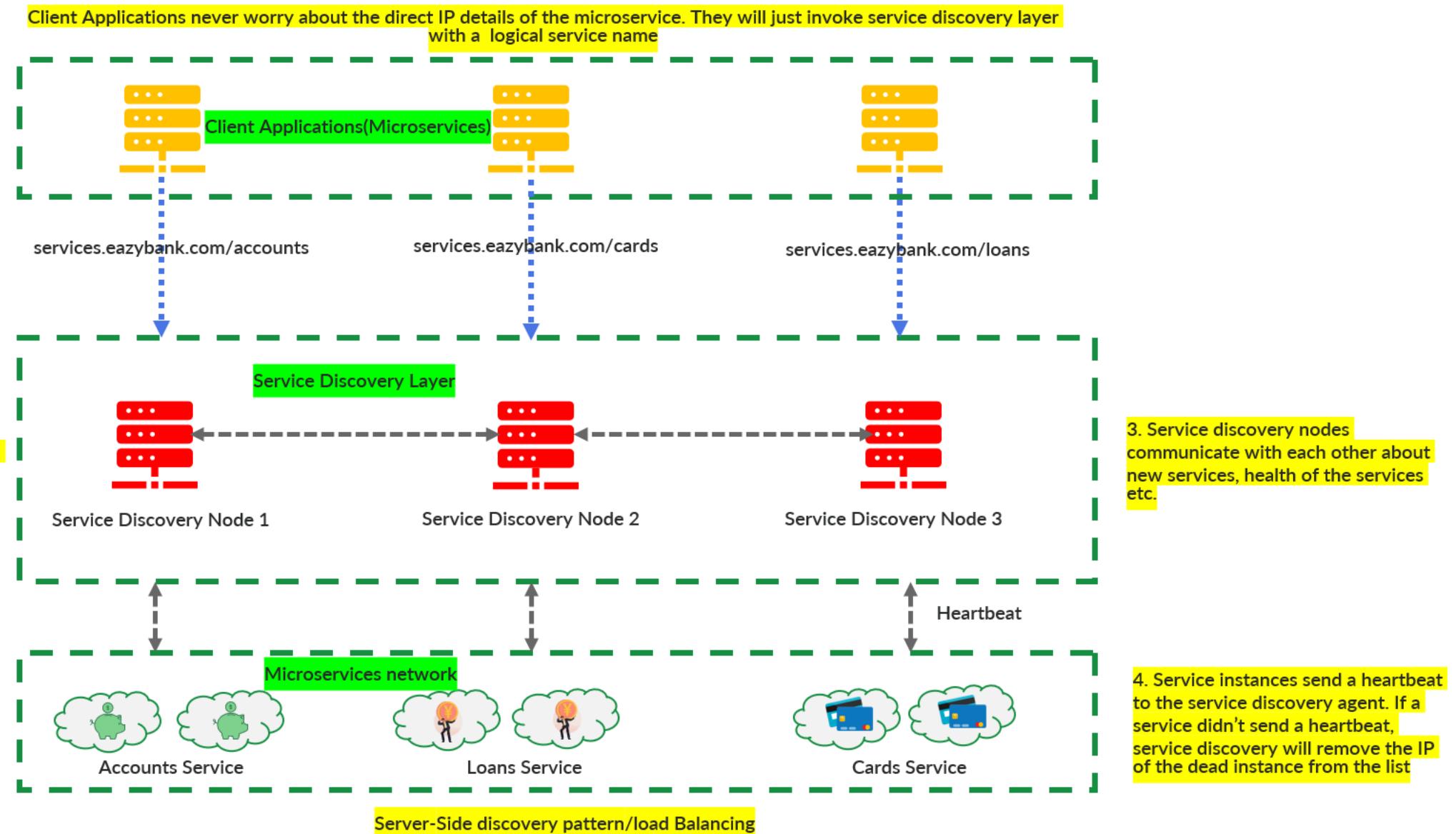
FOR SERVICE DISCOVERY & REGISTRATION

eazy  
bytes

- With traditional approach each instance of a service used to be deployed in one or more application servers. The number of these application servers was often static and even in the case of restoration it would be restored to the same state with the same IP and other configurations.
- While this type of model works well with monolithic and SOA based applications with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications for the following reasons,
  - Limited horizontal scalability & licenses costs
  - Single point of failure & Centralized chokepoints
  - Manually managed to update any IPs, configurations
  - Not containers friendly
  - Complex in nature

# ARCHITECTURE OF SERVICE DISCOVERY IN MICROSERVICES

eazy bytes



# ARCHITECTURE OF SERVICE DISCOVERY

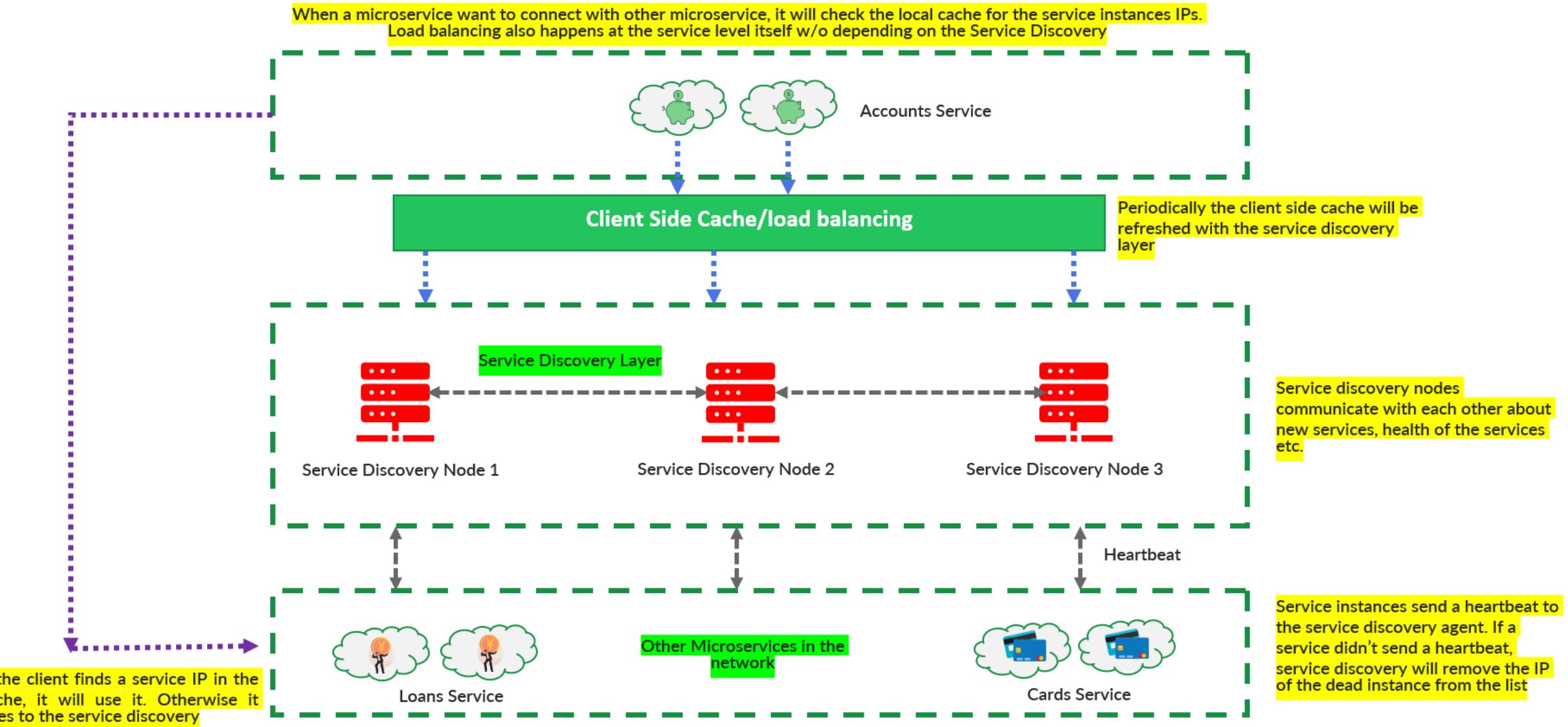
IN MICROSERVICES

eazy  
bytes

- Service discovery tools and patterns are developed to overcome the challenges with traditional load balancers.
- Mainly service discovery consists of a key-value store (Service Registry) and an API to read from and write to this store. New instances of applications are saved to this service registry and deleted when the service is down or not healthy.
- Clients, that want to communicate with a certain service are supposed to interact with the service registry to know the exact network location(s).
- Advantages of Service Discovery approach,
  - No limitations on availability
  - Peer to peer communication b/w Service Discovery agents
  - Dynamically managed IPs, configurations & Load balanced
  - Fault-tolerant & Resilient in nature

# CLIENT-SIDE LOAD BALANCING IN MICROSERVICES

eazy  
bytes



# SPRING CLOUD SUPPORT

## FOR SERVICE DISCOVERY & REGISTRATION

eazy  
bytes

- Spring Cloud project makes Service Discovery & Registration setup trivial to undertake with the help of the below components,
  - Spring Cloud Netflix's Eureka service which will act as a service discovery agent\*
  - Spring Cloud Load Balancer library for client-side load balancing\*\*
  - Netflix Feign client to look up for a service b/w microservices

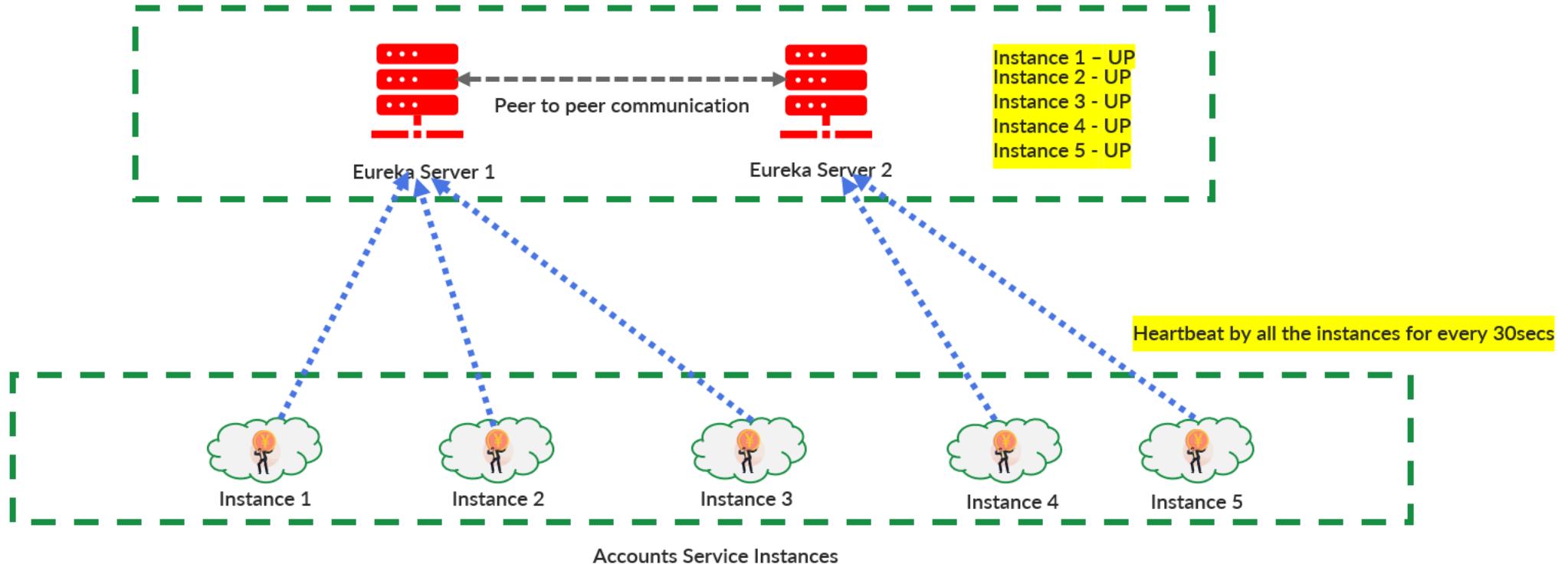
*\* Though in this course we use Eureka since it is mostly used but there are other service registries such as etcd, Consul, and Apache Zookeeper which are also good.*

*\*\* Though Netflix Ribbon client-side is also good and stable product, we are going to use Spring Cloud Load Balancer for client-side load balancing. This is because Ribbon has entered a maintenance mode and unfortunately, it will not be developed anymore*

# EUREKA SELF-PRESERVATION

TO AVOID TRAPS IN NETWORK

eazy  
bytes

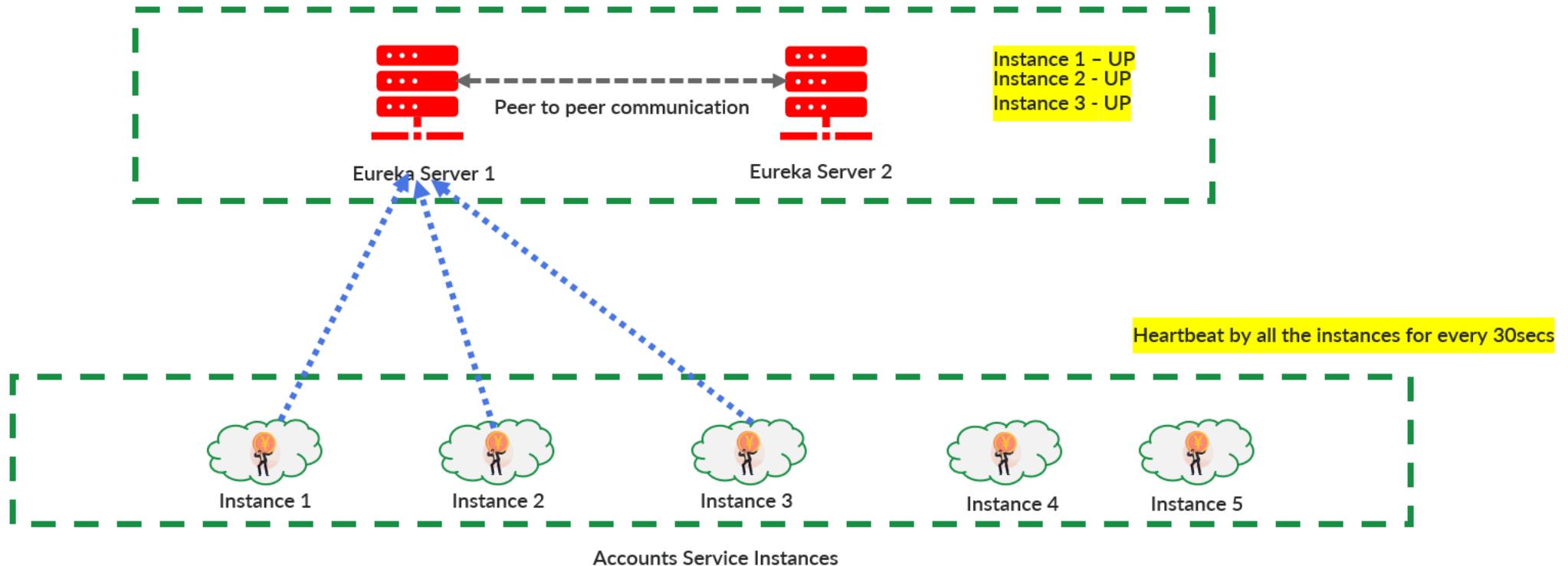


Healthy Microservices System with all 5 instances up before encountering network problems

# EUREKA SELF-PRESERVATION

TO AVOID TRAPS IN NETWORK

eazy  
bytes

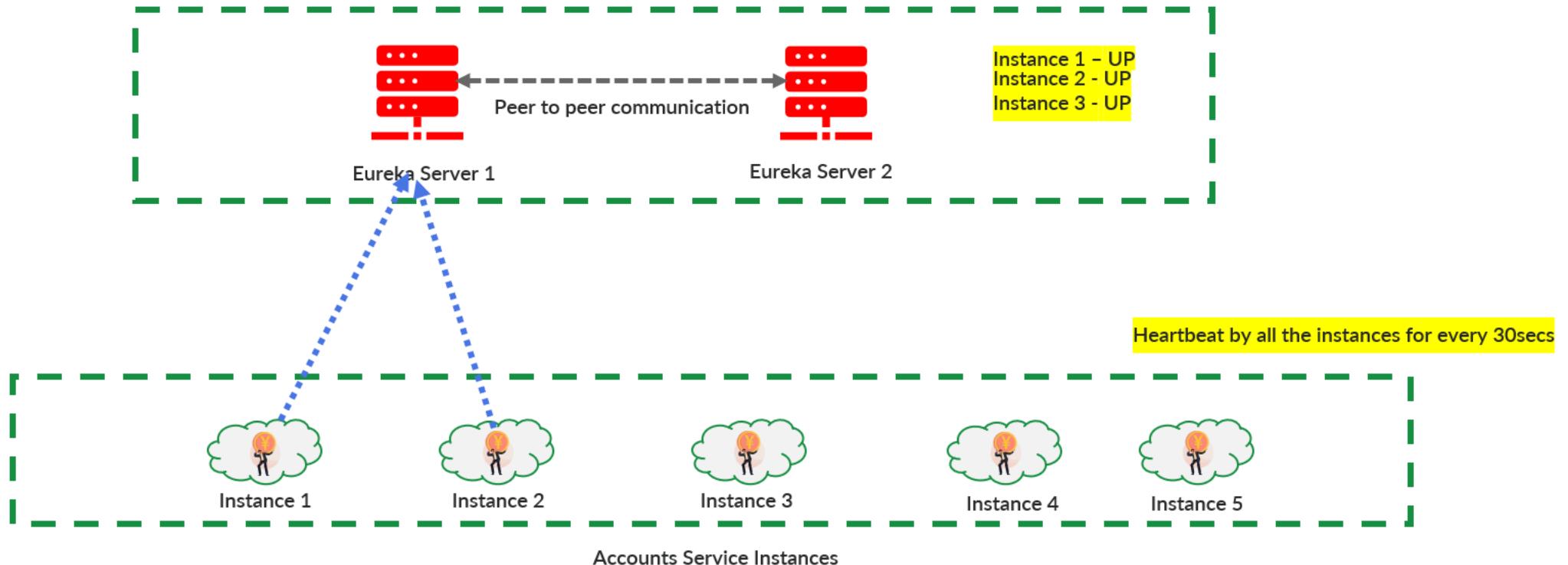


2 of the instances not sending heartbeat. Eureka enters self-preservation mode since it met threshold percentage

# EUREKA SELF-PRESERVATION

TO AVOID TRAPS IN NETWORK

eazy  
bytes



During Self-preservation, eureka will stop expiring the instances though it is not receiving heartbeat from instance 3

# EUREKA SELF-PRESERVATION

TO AVOID TRAPS IN NETWORK

eazy  
bytes

- The reason behind self-preservation mode in eureka
  - ✓ Servers not receiving heartbeats could be due to a poor network issue but does not necessarily mean the clients are down which may be resolved sooner. So with out self-preservation we will end up have zero instances up with Eureka though the instances might be up and running.
  - ✓ Even though the connectivity is lost between servers and some clients, clients might have connectivity with each other. With their local cache registration details they can keep communicating with each other
- Self-preservation mode never expires, until and unless the down microservices are brought back or the network glitch is resolved. This is because eureka will not expire the instances till it is above the threshold limit.
- Self-preservation will be a savior where the networks glitches are common and help us to handle false-positive alarms.

# EUREKA SELF-PRESERVATION

TO AVOID TRAPS IN NETWORK

eazy  
bytes

- Configurations which will directly or indirectly impact self-preservation behavior of eureka

- ✓ `eureka.instance.lease-renewal-interval-in-seconds = 30`

*Indicates the frequency the client sends heartbeats to server to indicate that it is still alive.*

- ✓ `eureka.instance.lease-expiration-duration-in-seconds = 90`

*Indicates the duration the server waits since it received the last heartbeat before it can evict an instance*

- ✓ `eureka.server.eviction-interval-timer-in-ms = 60 * 1000`

*A scheduler(EvictionTask) is run at this frequency which will evict instances from the registry if the lease of the instances are expired as configured by lease-expiration-duration-in-seconds. It will also check whether the system has reached self-preservation mode (by comparing actual and expected heartbeats) before evicting.*

- ✓ `eureka.server.renewal-percent-threshold = 0.85`

*This value is used to calculate the expected % of heartbeats per minute eureka is expecting.*

- ✓ `eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000`

*A scheduler is run at this frequency which calculates the expected heartbeats per minute*

- ✓ `eureka.server.enable-self-preservation = true`

*By default self-preservation mode is enabled but if you need to disable it you can change it to 'false'*

# CHALLENGE 5 WITH MICROSERVICES

## RESILIENCY

eazy  
bytes



### HOW DO WE AVOID CASCADING FAILURES?

One failed or slow service should not have a ripple effect on the other microservices. Like in the scenarios of multiple microservices are communicating, we need to make sure that the entire chain of microservices does not fail with the failure of a single microservice.



### HOW DO WE HANDLE FAILURES GRACEFULLY WITH FALLBACKS?

In a chain of multiple microservices, how do we build a fallback mechanism if one of the microservice is not working. Like returning a default value or return values from cache or call another service/DB to fetch the results etc.



### HOW TO MAKE OUR SERVICES SELF-HEALING CAPABLE

In the cases of slow performing services, how do we configure timeouts, retries and give time for a failed services to recover itself.



# SPRING SUPPORT

## FOR RESILIENCY USING RESILIENCE4J

eazy  
bytes

- Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming. Lightweight, because the library only uses Vavr, which does not have any other external library dependencies. Netflix Hystrix, in contrast, has a compile dependency to Archaius which has many more external library dependencies such as Guava and Apache Commons Configuration.
- Resilience4j offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:
  - ✓ Circuit breaker - Used to stop making requests when a service invoked is failing.
  - ✓ Fallback - Alternative paths to failing requests.
  - ✓ Retry - Used to make retries when a service has temporarily failed.
  - ✓ Rate limit - Limits the number of calls that a service receives in a time.
  - ✓ Bulkhead - Limits the number of outgoing concurrent requests to a service to avoid overloading.
- Before using Resilience4j, Developers used to use Hystrix, one of the most common java libraries to implement the resiliency patterns in microservices. But now Hystrix is in maintenance mode and no new features are developed. Due to this reason now everyone uses Resilience4j which has more features than Hystrix.

# TYPICAL SCENARIO IN MICROSERVICES

eazy  
bytes

App 1 which needs response from all 3 services

App 2 which needs response from Accounts



Accounts Microservice



Accounts Database



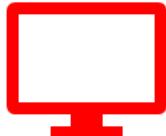
Loans Microservice



Loans Database



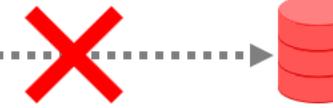
App 3 which needs  
response from Loans



Cards Microservice



Cards Database



App 4 which needs  
response from Cards

In my microservices network both Accounts, Loans are working fine but Cards is not responding properly due to DB connectivity issues

# CIRCUIT BREAKER PATTERN

## FOR RESILIENCY IN MICROSERVICES

eazy  
bytes

- In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them.
- The Circuit Breaker pattern which inspired from electrical circuit breaker will monitor the remote calls. If the calls take too long, the circuit breaker will intercede and kill the call. Also, the circuit breaker will monitor all calls to a remote resource, and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.
- The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.
- The advantages with circuit breaker pattern are,
  - ✓ Fail fast
  - ✓ Fail gracefully
  - ✓ Recover seamlessly

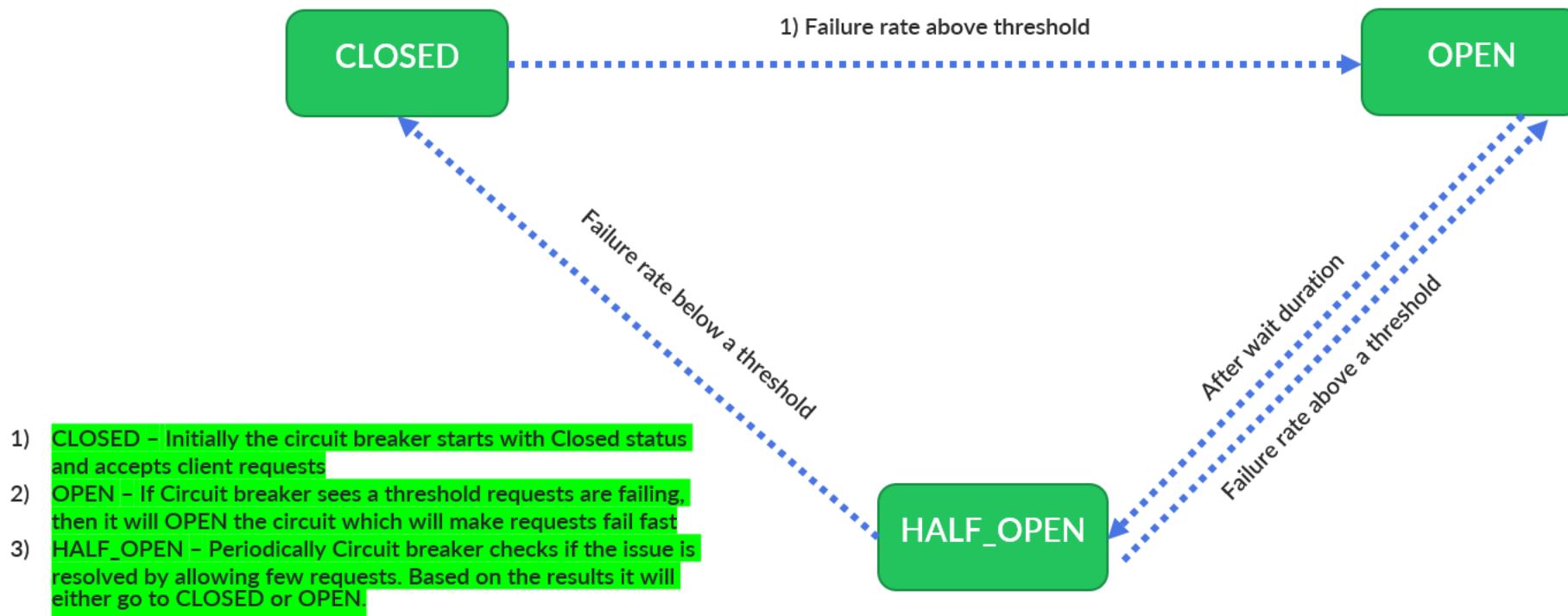
```
@CircuitBreaker(name="detailsForCustomerSupportApp",  
fallbackMethod= "myCustomerDetailsFallBack")
```

# CIRCUIT BREAKER PATTERN

FOR RESILIENCY IN MICROSERVICES

eazy  
bytes

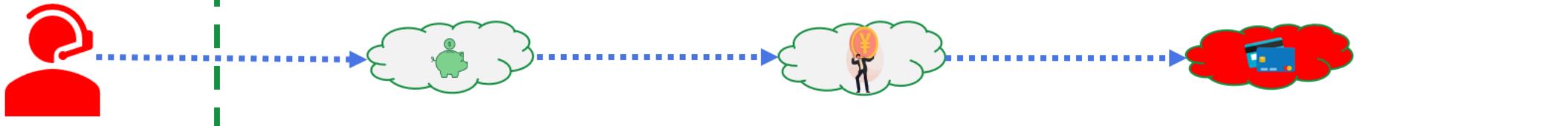
In Resilience4j the circuit breaker is implemented via a finite state machine with the following states



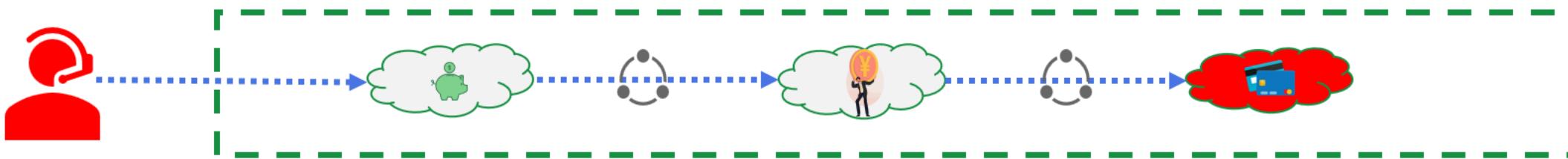
# CIRCUIT BREAKER PATTERN

## FOR RESILIENCY IN MICROSERVICES

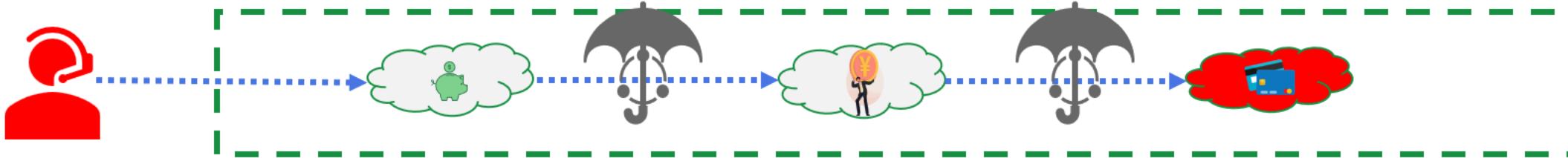
eazy  
bytes



Scenario 1 – If Cards microservice is responding slowly, then without circuit breaker it will start eating up all the resources threads on the Loans and Accounts microservices which will make them also slow/down eventually



Scenario 2 – If Cards microservice is responding slowly, then with circuit breakers in between it will start acting and failing the services fast with the states OPEN, HALF\_OPEN, CLOSED. This way at least Accounts and Loans services will not have any issues for other Apps.



Scenario 3 – If Cards microservice is responding slowly then with circuit breakers and fallback mechanism, we can make sure that at least we are failing gracefully with some default response is being returned and at the same time other microservices will not get impacted

# RETRY PATTERN

## FOR RESILIENCY IN MICROSERVICES

eazy  
bytes

- The retry pattern will make configured multiple retry attempts when a service has temporarily failed. This pattern is very helpful in the scenarios like network disruption where the client request may successful after a retry attempt.
- For retry pattern we can configure the following values,
  - ✓ maxAttempts - The maximum number of attempts
  - ✓ waitDuration - A fixed wait duration between retry attempts
  - ✓ retryExceptions - Configures a list of Throwable classes that are recorded as a failure and thus are retried.
  - ✓ ignoreExceptions - Configures a list of Throwable classes that are ignored and thus are not retried.
- We can also define a fallback mechanism if the service call fails even after multiple retry attempts. Below is a sample configuration,

```
@Retry(name="detailsForCustomerSupportApp", fallbackMethod= "myCustomerDetailsFallBack")
```

# RATE LIMITTER PATTERN

## FOR RESILIENCY IN MICROSERVICES

eazy  
bytes

- The rate limiter pattern will help to stop overloading the service with more calls more than it can consume in a given time. This is an imperative technique to prepare our API for high availability and reliability.
- This pattern protect APIs and service endpoints from harmful effects, such as denial of service, cascading failure.
- For rate limiter pattern we can configure the following values,
  - ✓ timeoutDuration - The default wait time a thread waits for a permission
  - ✓ limitForPeriod - The number of permissions available during one limit refresh period
  - ✓ limitRefreshPeriod - The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
- We can also define a fallback mechanism if the service call fails due to rate limiter configurations. Below is a sample configuration,

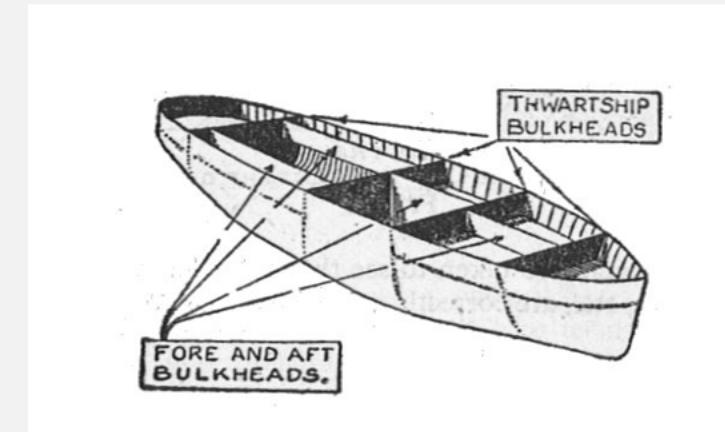
```
@RateLimiter(name="detailsForCustomerSupportApp", fallbackMethod="myCustomerDetailsFallBack")
```

# BULKHEAD PATTERN

## FOR RESILIENCY IN MICROSERVICES

eazy  
bytes

- 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 microservices resources should be isolated in such a way that failure of one component is not affecting the entire microservice.
- Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced.
- For Bulkhead pattern we can configure the following values,
  - ✓ maxConcurrentCalls - Max amount of parallel executions allowed by the bulkhead
  - ✓ maxWaitDuration - Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

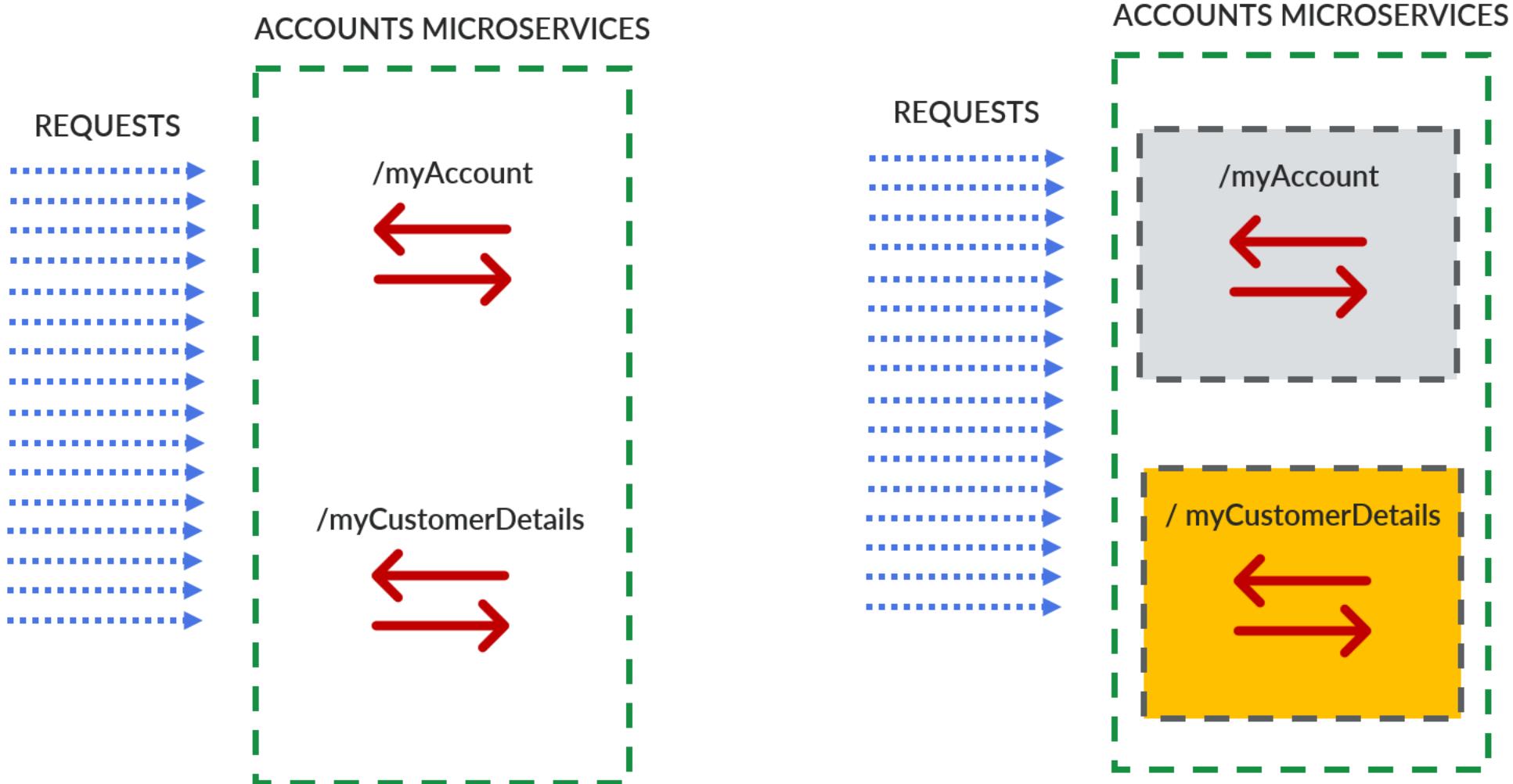


```
@Bulkhead(name="bulkheadAccounts", fallbackMethod= "bulkheadAccountsFallBack")
```

# BULKHEAD PATTERN

ARCHITECTURE INSIDE MICROSERVICES

eazy  
bytes



Without Bulkhead, `/myCustomerDetails` will start eating all the threads, resources available which will effect the performance of `/myAccount`

With Bulkhead, `/myCustomerDetails` and `/myAccount` will have their own resources, threads pool defined

# CHALLENGE 6 WITH MICROSERVICES

## ROUTING, CROSS CUTTING CONCERNS

eazy  
bytes



### HOW DO WE ROUTE BASED ON CUSTOM REQUIREMENTS

If we have a custom requirements to route the incoming requests to the appropriate destination both in static and dynamic way, how do we do that?



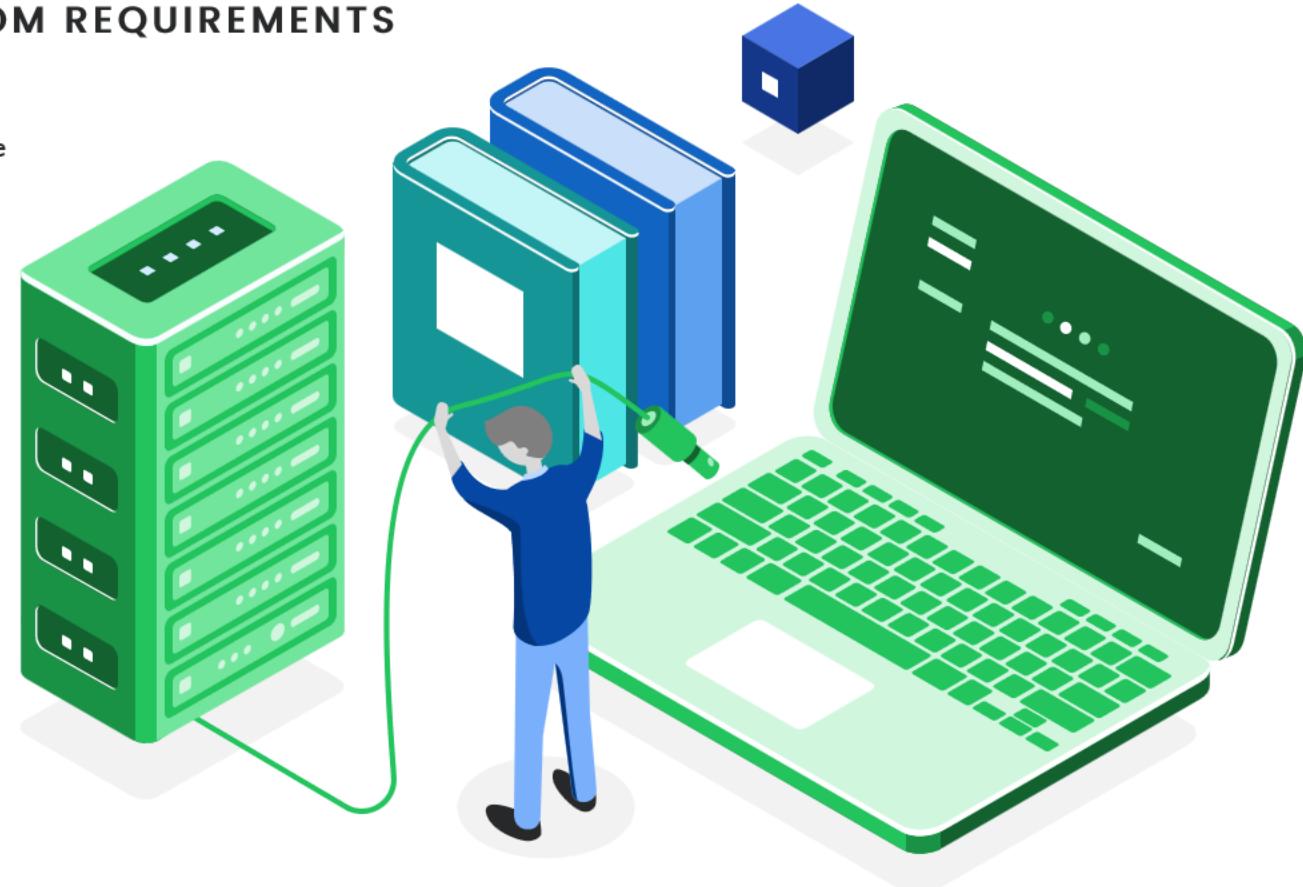
### HOW DO WE HANDLE CROSS CUTTING CONCERNS?

In a distributed microservices architecture, how do we make sure to have a consistently enforced cross cutting concerns like logging, auditing, tracing, security and metrics collection across multiple microservices



### HOW DO WE BUILD A SINGLE GATEKEEPER?

How do we build a single gatekeeper for all the inbound traffic to our microservices which will act as a central Policy Enforcement Point (PEP) for all service calls?



# SPRING CLOUD SUPPORT

FOR ROUTING, CROSS CUTTING CONCERNS

eazy  
bytes

- 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.
- The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.
- The service gateway sits between all calls from the client to the individual services, it also acts as a central Policy Enforcement Point (PEP) like below for service calls.
  - ✓ Routing (Both Static & Dynamic)
  - ✓ Security (Authentication & Authorization)
  - ✓ Logging, Auditing and Metrics collection

# SPRING CLOUD SUPPORT

FOR ROUTING, CROSS CUTTING CONCERNS

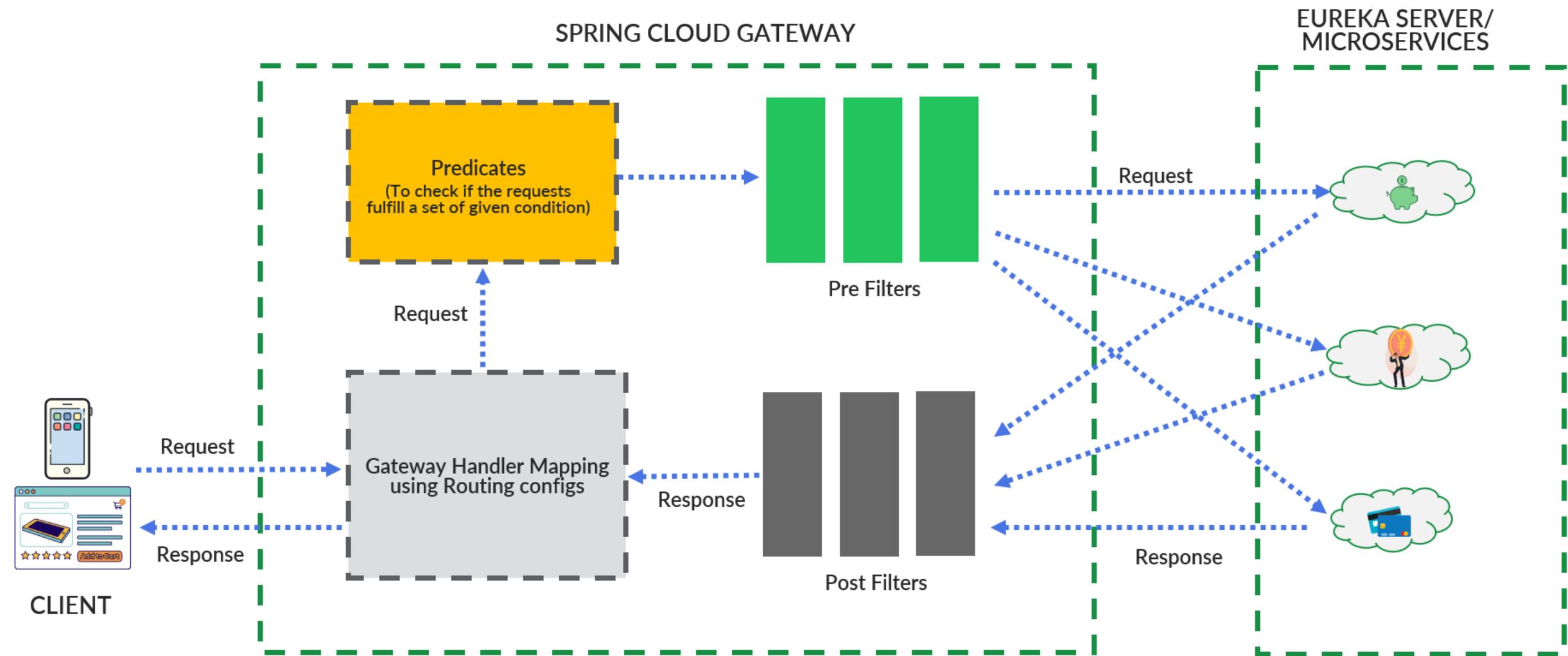
eazy  
bytes

- Spring Cloud Gateway is a library for building an API gateway, so it looks like any another Spring Boot application. If you're a Spring developer, you'll find it's very easy to get started with Spring Cloud Gateway with just a few lines of code.
- Spring Cloud Gateway is intended to sit between a requester and a resource that's being requested, where it intercepts, analyzes, and modifies every request. That means you can route requests based on their context. Did a request include a header indicating an API version? We can route that request to the appropriately versioned backend. Does the request require sticky sessions? The gateway can keep track of each user's session.
- Spring Cloud Gateway is replacement of Zuul for the following reasons and advantages,
  - Spring Cloud Gateway is the preferred API gateway implementation from the Spring Cloud Team. It's built on Spring 5, Reactor, and Spring WebFlux. Not only that, it also includes circuit breaker integration, service discovery with Eureka
  - ✓ Spring Cloud Gateway is non-blocking in nature. Though later Zuul 2 also supports it, but still Spring Cloud Gateway has an edge here
  - ✓ Spring Cloud Gateway has a superior performance compared to that of Zuul.

# SPRING CLOUD GATEWAY

## INTERNAL ARCHITECTURE

eazy  
bytes



When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the filters. Post filter it will send to the actual microservices or Eureka Server

# CHALLENGE 7 WITH MICROSERVICES

## DISTRIBUTED TRACING & LOG AGGREGATION

eazy  
bytes



### HOW DO WE DEBUG WHERE A PROBLEM IN MICROSERVICES?

How do we trace one or more transactions across multiple services, physical machines, and different data stores, and try to find where exactly the problem or bug is?



### HOW DO WE AGGREGATE ALL APPLICATION LOGS?

How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered, and grouped to find bugs that are contributing to a problem?



### HOW DO WE MONITOR OUR CHAIN OF SERVICE CALLS?

How do we understand for a specific chain service call the path it travelled inside our microservices network, time it took at each microservice etc. ?



# SPRING CLOUD SUPPORT

FOR DISTRIBUTED TRACING & LOG AGGREGATION

eazy  
bytes

## Spring Cloud Sleuth (<https://spring.io/projects/spring-cloud-sleuth>)

- Spring Cloud Sleuth provides Spring Boot auto-configuration for distributed tracing.
- It adds trace and span ids to all the logs, so you can just extract from a given trace or span in a log aggregator.
- It does this by adding the filters and interacting with other Spring components to let the correlation IDs being generated pass through to all the system calls.

## Zipkin (<https://zipkin.io/>)

- Zipkin is a is an open-source data-visualization tool that can helps aggregating all the logs and gather timing data needed to troubleshoot latency problems in microservices architectures.
- It allows us to break a transaction down into its component pieces and visually identify where there might be performance hotspots. Thus reducing time in triaging by contextualizing errors and delays.

# SPRING CLOUD SLEUTH

## TRACE FORMAT

eazy  
bytes

- Spring Cloud Sleuth will add three pieces of information to all the logs written by a microservice.

[<App Name>,<Trace ID>,<Span ID>]

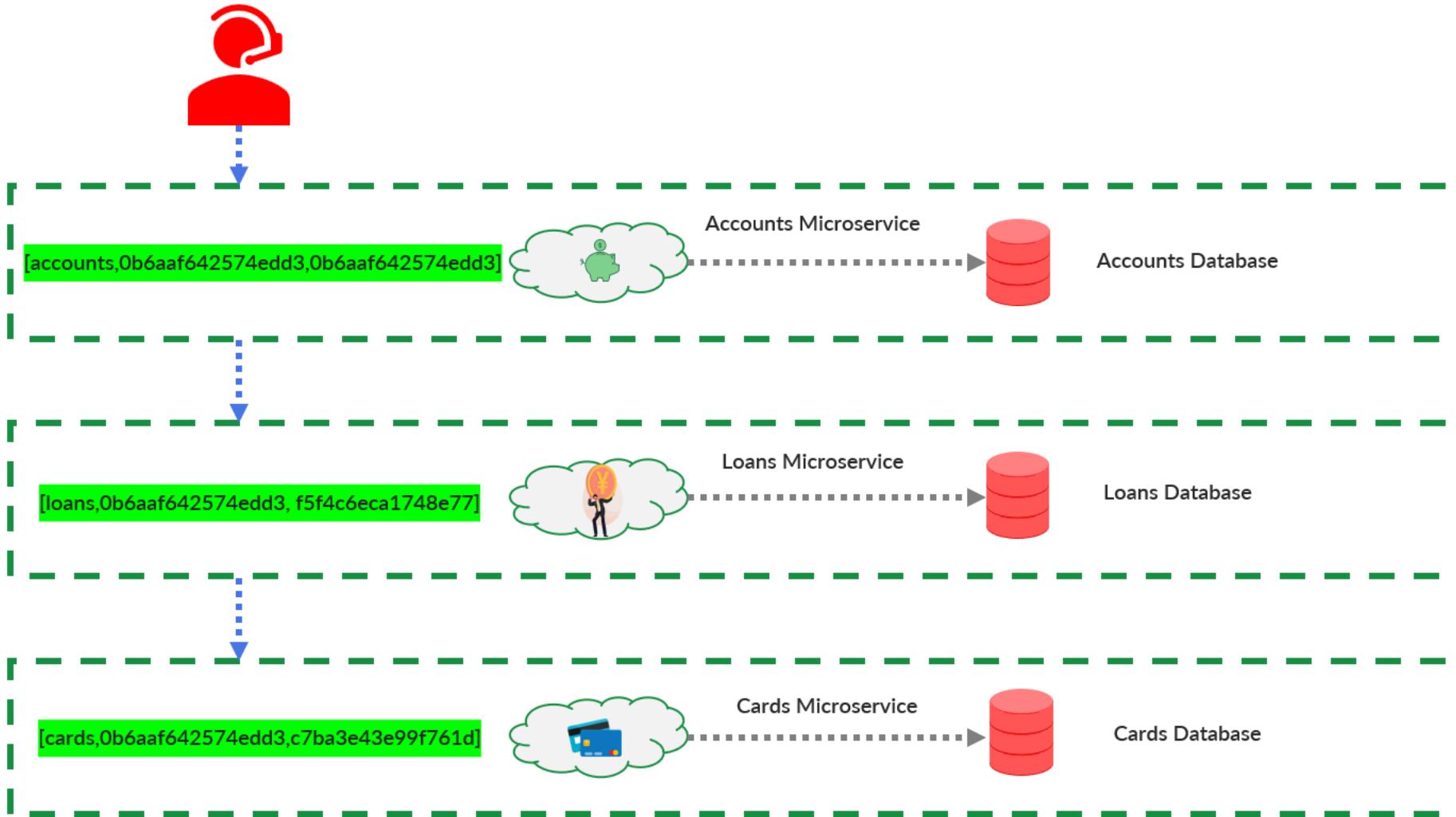
- **Application name of the service:** This is going to be the application name where the log entry is being made. Spring Cloud Sleuth get this name from the 'spring.application.name' property.
- **Trace ID:** Trace ID is the equivalent term for correlation ID. It's a unique number that represents an entire transaction.
- **Span ID:** A span ID is a unique ID that represents part of the overall transaction. Each service participating within the transaction will have its own span ID. Span IDs are particularly relevant when you integrate with Zipkin to visualize your transactions.

# SPRING CLOUD SLEUTH

## TRACE FORMAT

eazy  
bytes

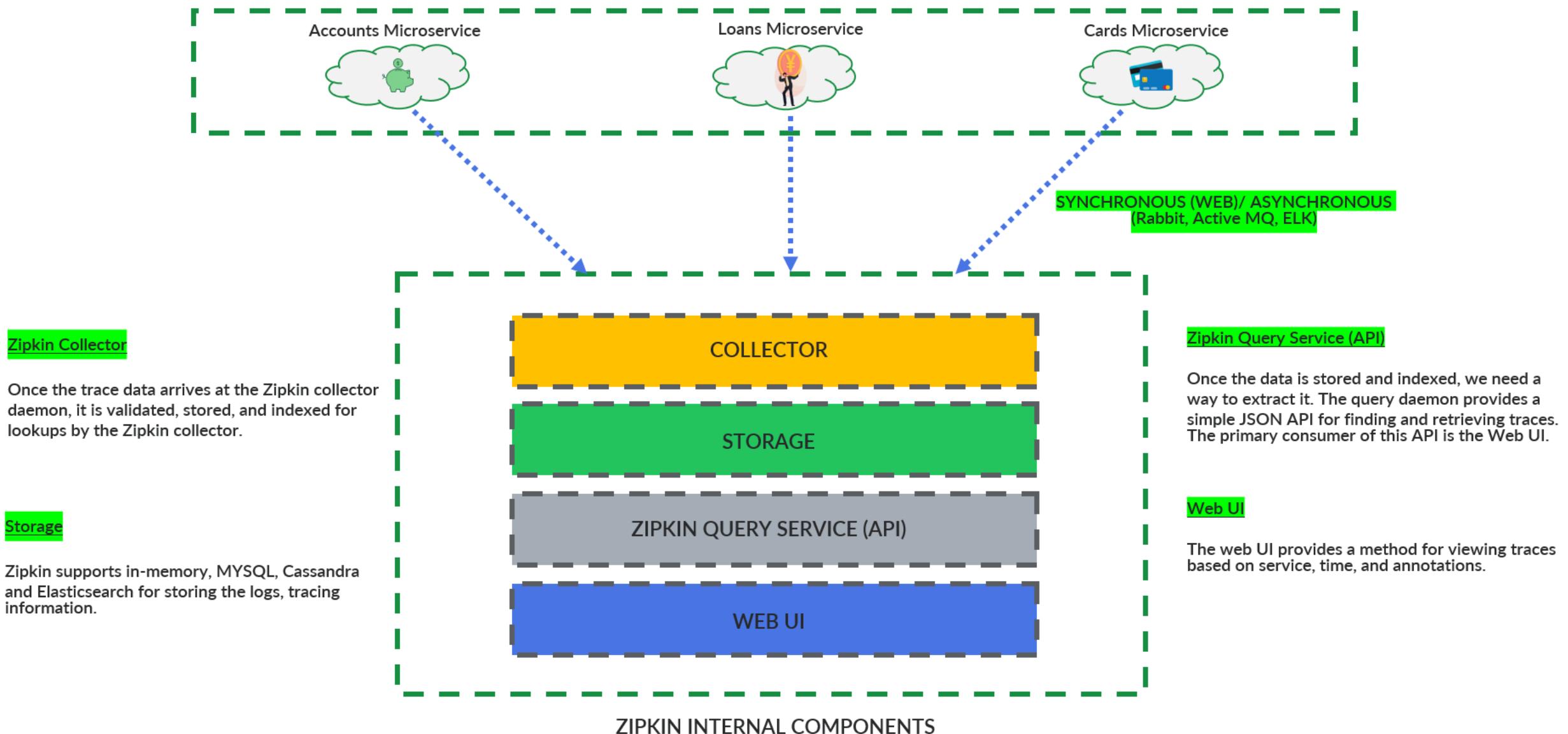
App which needs response from all 3 services



# ZIPKIN

## ARCHITECTURE OVERVIEW

eazy  
bytes



# CHALLENGE 8 WITH MICROSERVICES

## MONITORING MICROSERVICES HEALTH & METRICS

eazy  
bytes



### HOW DO WE MONITOR SERVICES METRICS?

How do we monitor the metrics like CPU usage, JVM metrics etc. for all the microservices applications we have inside our network easily and efficiently?



### HOW DO WE MONITOR SERVICES HEALTH?

How do we monitor the status/health for all the microservices applications we have inside our network in a single place?



### HOW DO WE CREATE ALERTS BASED ON MONITORING ?

How do we create alerts/notifications for any abnormal behavior of the services?



# DIFF APPROACHES TO MONITOR

eazy  
bytes

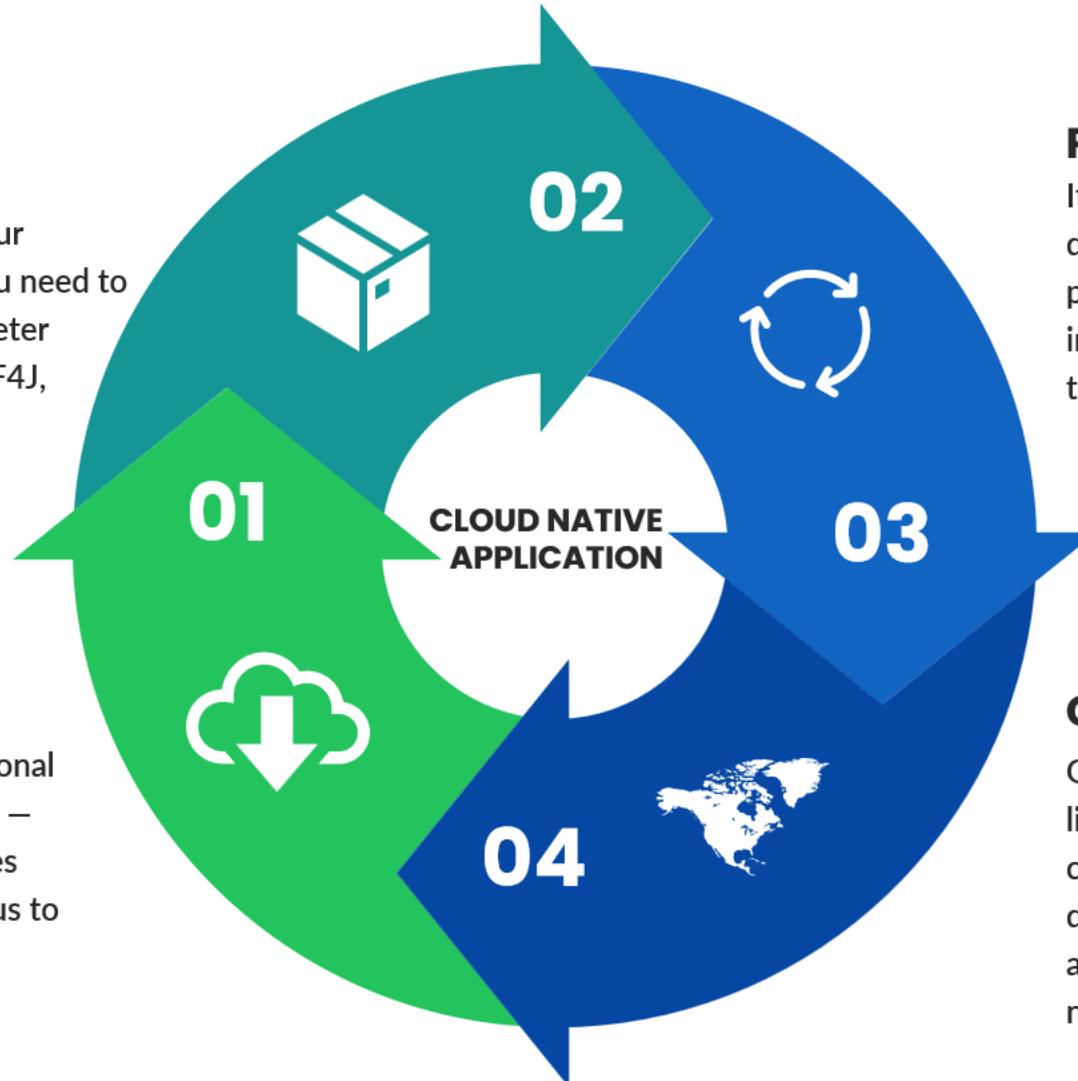
## MICROSERVICES HEALTH & METRICS

### MICROMETER

Micrometer automatically exposes /actuator/metrics data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application. Think SLF4J, but for metrics.

### ACTUATOR

Actuator is mainly used to expose operational information about the running application – health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.



### PROMETHEUS

It is a time-series database that stores our metric data by pulling it (using a built-in data scraper) periodically over HTTP. It also has a simple user interface where we can visualize/query on all of the collected metrics.

### GRAFANA

Grafana can pull data from various data sources like Prometheus and offers a rich UI where you can build up custom graphs quickly and create a dashboard out of many graphs in no time. It also allows you to set rule-based alerts, for notifications.

# CHALLENGE 9 WITH MICROSERVICES

## CONTAINER ORCHESTRATION

eazy  
bytes



### HOW DO WE AUTOMATE THE DEPLOYMENTS, ROLLOUTS & ROLLBACKS?

How do we automate deployment of the containers into a complex cluster env and perform rollout of new versions of the containers without down time along with an option of automatic rollback in case of any issues?



### HOW DO WE MAKE SURE OUR SERVICES ARE SELF-HEALING?

How do we automatically restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.



### HOW DO WE AUTO SCALE OUR SERVICES ?

How do we monitor our services and scale them automatically based on metrics like CPU Utilization etc. ?





# KUBERNETES (K8S)

FOR CONTAINER ORCHESTRATION

eazy  
bytes

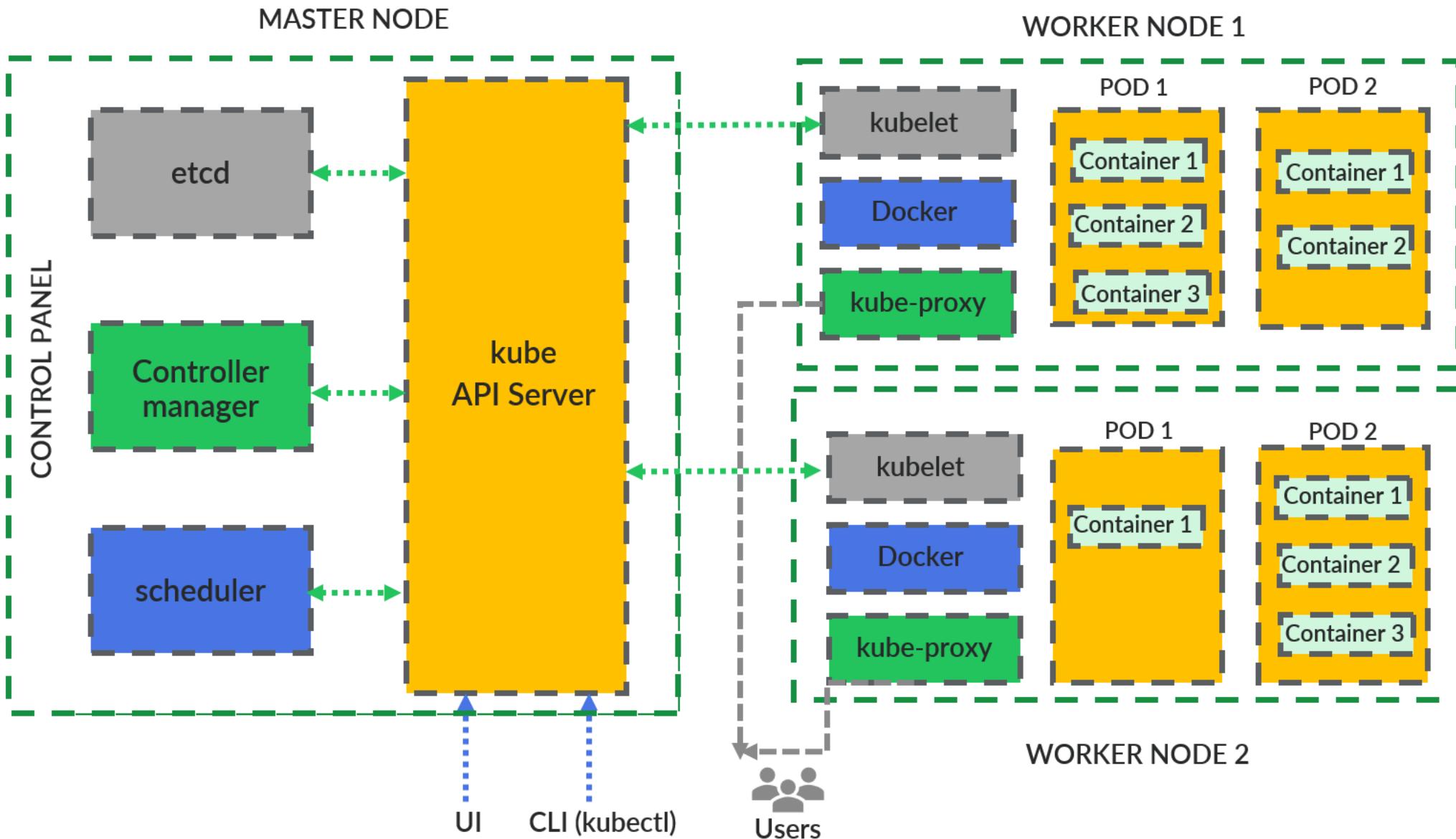
- Kubernetes, is an open-source system for automating deployment, scaling, and managing containerized applications. It is the most famous orchestration platform and it is cloud neutral.
- The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s".
- Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.
- Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. It provides you with:
  - *Service discovery and load balancing*
  - *Storage orchestration*
  - *Automated rollouts and rollbacks*
  - *Automatic bin packing*
  - *Self-healing*
  - *Secret and configuration management*



# KUBERNETES (K8S)

## INTERNAL ARCHITECTURE

eazy  
bytes





## Master Node (Control Plane)

- The master node is responsible for managing an entire cluster. It monitors the health check of all the nodes in the cluster, stores members' information regarding different nodes, plans the containers that are scheduled to certain worker nodes, monitors containers and nodes, etc. So, when a worker node fails, the master moves the workload from the failed node to another healthy worker node.
- The Kubernetes master is responsible for scheduling, provisioning, configuring, and exposing APIs to the client. So, all these are done by a master node using control plane components. Kubernetes takes care of service discovery, scaling, load balancing, self-healing, leader election, etc. Therefore, developers no longer have to build these services inside their applications.



## Master Node (Control Plane)

- Four basic components of the master node (control plane):
  - ✓ **API server** - *The API Server is the front-end of the control plane and the only component in the control plane that we interact with directly. Internal system components, as well as external user components, all communicate via the same API.*
  - ✓ **Scheduler** - *Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.*
  - ✓ **Controller manager** - *The controller manager maintains the cluster. It handles node failures, replicates components, maintains the correct number of pods, etc. It constantly tries to keep the system in the desired state by comparing it with the current state of the system.*
  - ✓ **Etcd** - *Etcd is a data store that stores the cluster configuration. It is a distributed reliable, key-value store; all the configurations are stored in documents, and it's schema-less.*



## Worker Node (Data plane)

- The worker node is nothing but a virtual machine (VM) running in the cloud or on-prem (a physical server running inside your data center). So, any hardware capable of running container runtime can become a worker node. These nodes expose underlying compute, storage, and networking to the applications.
- Worker nodes do the heavy-lifting for the application running inside the Kubernetes cluster. Together, these nodes form a cluster – a workload assign is run to them by the master node component, similar to how a manager would assign a task to a team member. This way, we will be able to achieve fault-tolerance and replication.
- Pods are the smallest unit of deployment in Kubernetes just as a container is the smallest unit of deployment in Docker. To understand in an easy way, we can say that pods are nothing but lightweight VMs in the virtual world. Each pod consists of one or more containers. Pods are ephemeral in nature as they come and go, while containers are stateless in nature. Usually, we run a single container inside a pod. There are some scenarios where we will run multiple containers that are dependent on each other inside a single pod. Each time a pod spins up, it gets a new IP address with a virtual IP range assigned by the pod networking solution.



## Worker Node (Data plane)

- There are three basic components of the worker node (data plane):
  - ✓ **Kubelet** - An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.
  - ✓ **Kube-proxy** - kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
  - ✓ **Container runtime** - Container runtime runs containers like Docker, or containerd. Once you have the specification that describes the image for your application, the container runtime will pull the images and run the containers.



# KUBERNETES (K8S)

SUPPORT BY CLOUD PROVIDERS

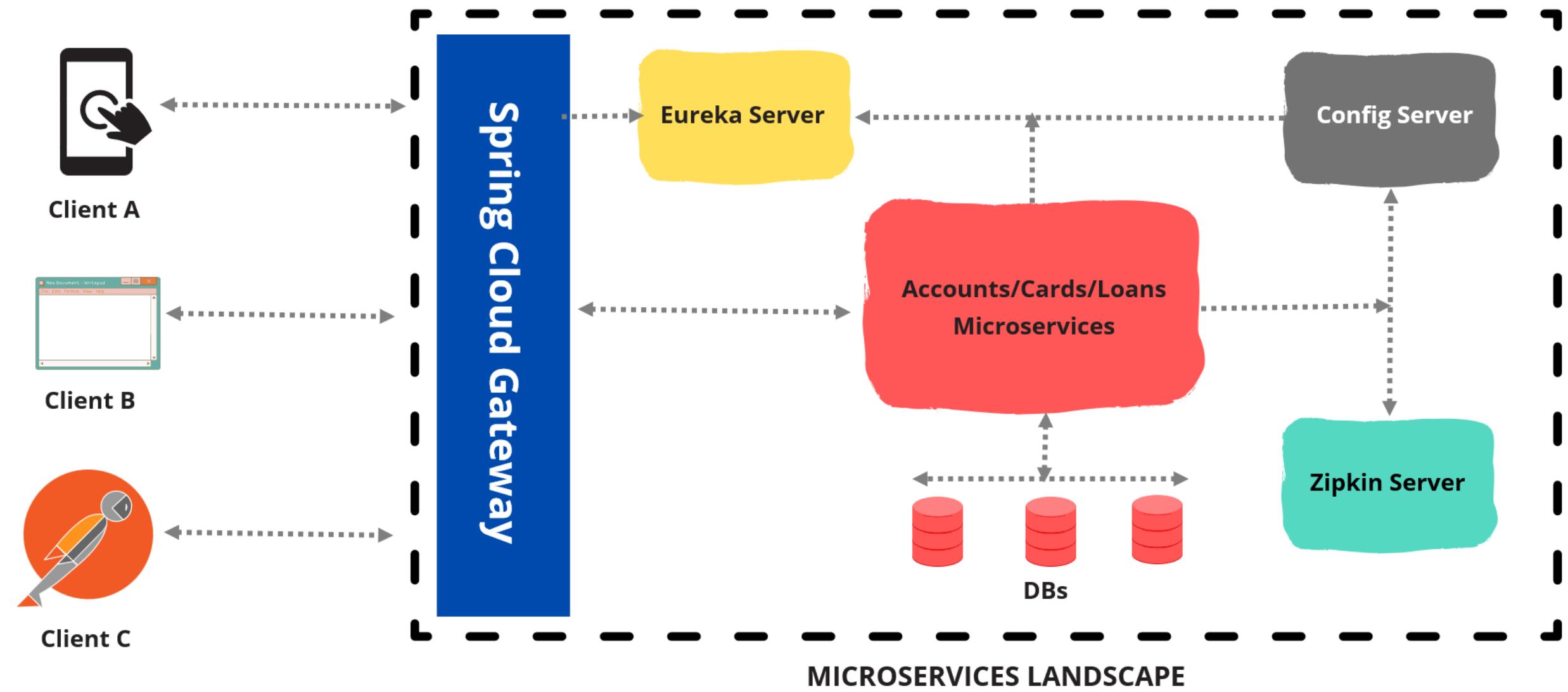
eazy  
bytes

- Kubernetes is so modular, flexible, and extensible that it can be deployed on-prem, in a third-party data center, in any of the popular cloud providers and even across multiple cloud providers.
- Creating and maintaining a K8S cluster can be very challenge in on-prem. Due that many enterprises look for the cloud providers which will make their job easy to maintain their microservice architecture using Kubernetes.
- Below are the different famous cloud providers and their support to Kubernetes with the different names,
  - GCP - GKE (Google Kubernetes Engine)
  - AWS - EKS (Elastic Kubernetes Service)
  - Azure - AKS (Azure Kubernetes Service)

# MICROSERVICES COMPONENTS SO FAR

WITH SPRING CLOUD PROJECTS

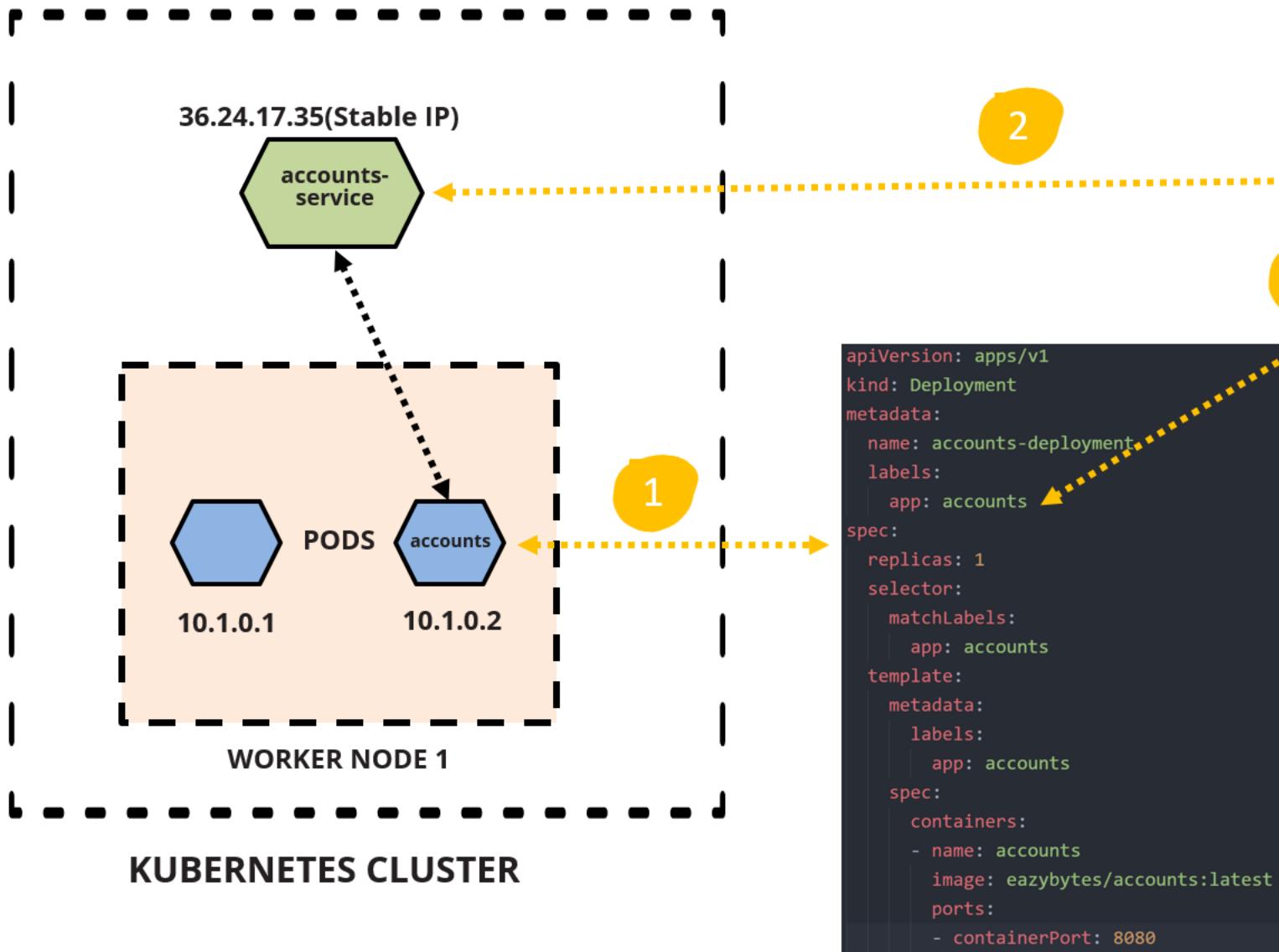
eazy  
bytes



# KUBERNETES SERVICES

## HOW DEPLOYMENT & SERVICES TIED TOGETHER

eazy  
bytes



1. K8s Deployment manifest will give instructions to deploy the containers into a pod inside one of the worker node of the K8s cluster
2. K8s Service manifest will give instructions to create a service inside the K8s cluster which tracks service registration & discovery of a specific service/deployment
3. The binding between a container running inside a Pod and a service will be done based “app” label & selector inside the manifest files.

# INTRODUCTION TO HELM

eazy  
bytes



## What is Helm ?

Helm is widely known as "the package manager for K8s". The goal of Helm was to provide users with a better way to manage all the Kubernetes YAML files we create on Kubernetes projects.

1

With out Helm, we need to maintain all the Kubernetes manifest files for Deployment, Service, ConfigMap etc. for each microservice.

2

With out Helm, the Devops team need to run all the K8s YAML manifest files via kubectl

3

The path Helm took to solve this issue was to create Helm Charts. Each chart is a bundle with one or more Kubernetes manifests – a chart can have child charts and dependent charts as well. This means that Helm installs the whole dependency tree of a project with just a single command.

# PROBLEMS THAT HELM SOLVES

With out Helm, we need to maintain the separate YAML files/manifest of K8s for all microservices inside a project like below. But majority content inside them looks similar except few dynamic values.

```
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

accounts service manifest

```
apiVersion: v1
kind: Service
metadata:
  name: loans-service
spec:
  selector:
    app: loans
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8090
      targetPort: 8090
```

loans service manifest

```
apiVersion: v1
kind: Service
metadata:
  name: cards-service
spec:
  selector:
    app: cards
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 9000
```

cards service manifest

# PROBLEMS THAT HELM SOLVES

eazy  
bytes

With Helm, we can a single template yaml file like shown below. Only the dynamic values will be injected during K8s services setup based on the values mentioned inside the values.yaml present inside each service/chart.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.deploymentLabel }}
spec:
  selector:
    app: {{ .Values.deploymentLabel }}
  type: {{ .Values.service.type }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
```

Helm service template

```
deploymentLabel: accounts
service:
  type: ClusterIP
  port: 8080
  targetPort: 8080
```

values.yaml

# PROBLEMS THAT HELM SOLVES

eazy  
bytes



## HELM SUPPORTS PACKAGING OF YAML FILES

With the help of Helm, we can package all of the YAML manifest files belongs in to an application into a Chart. The same can be distributed into public or private repositories.



## HELM SUPPORTS EASIER INSTALLATION

With the help of Helm, we can set up/upgrade/rollback/remove entire microservices applications into K8s cluster with just 1 command. No need to manually run `kubectl apply` command for each manifest file.



## HELM SUPPORTS RELEASE/VERSION MANAGEMENT

Helm automatically maintains the version history of the installed manifests.

Due to that rollback of entire K8s cluster to the previous working state is just a single command away.



Helm acts as a package manager for K8s. Just like a package manager is a collection of software tools that automates the process of installing, upgrading, version management, and removing computer programs for a computer in a consistent manner, similarly Helm automates the installation, rollback, upgrade of the multiple K8s manifests with a single command.

# HELM CHART STRUCTURE

eazy  
bytes

**helm create <chart-name>** will create a chart folder structure like below.

```
eazybank-common/
|---Chart.yaml
|---values.yaml
|---charts/
|---templates/
```

Top level **eazybank-common** folder is the name of the chart that we have given while creating the chart.

**Chart.yaml** will have meta info about the helm chart  
**values.yaml** will have dynamic values for the chart

**charts** folder will have other charts which the current chart is dependent on.

**templates** folder contains the manifest template yaml files

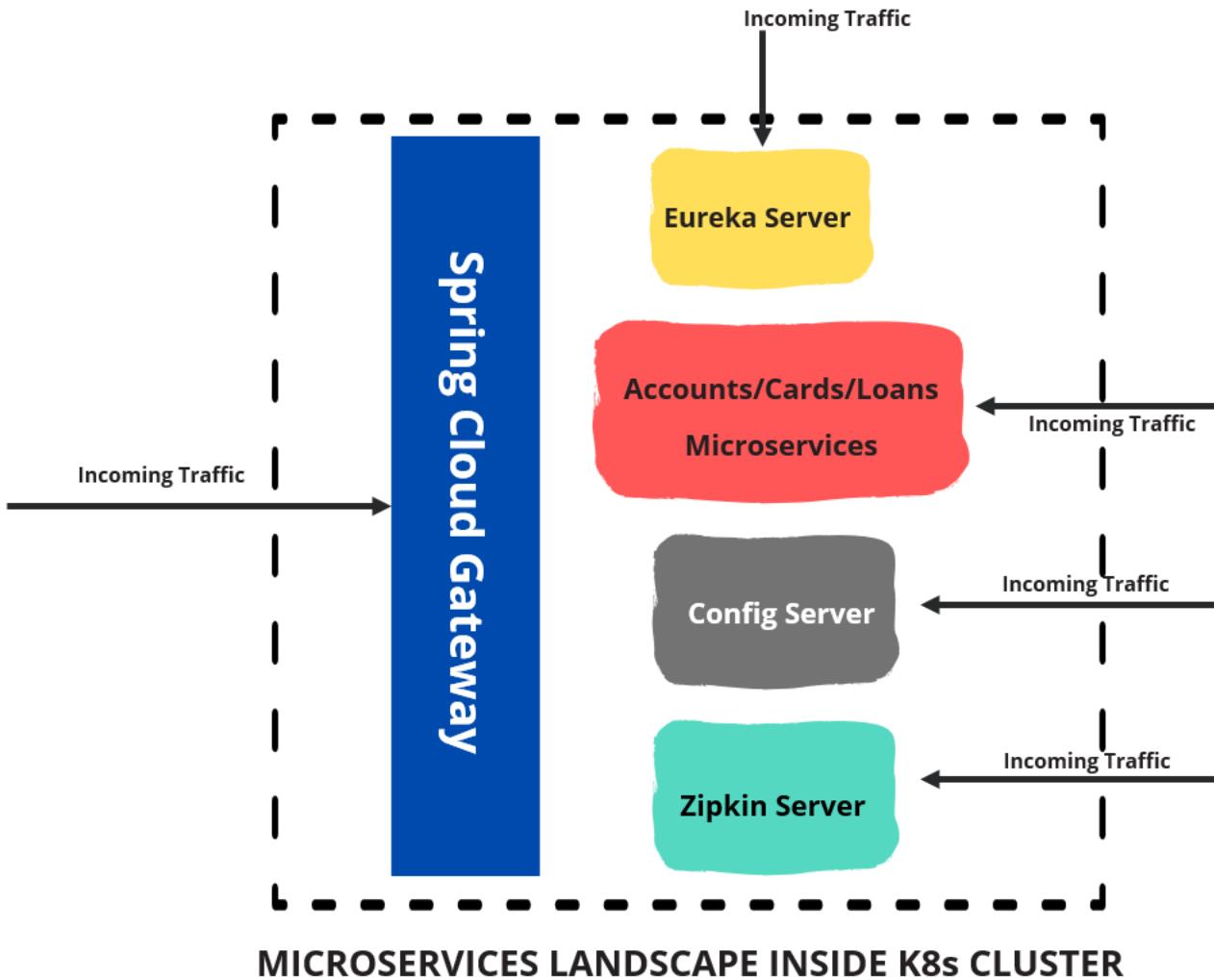
# IMPORTANT HELM COMMANDS

## Helm commands

Command	Description
helm create <b>eazybank</b>	Create a blank chart with the name eazybank. Inside the eazybank folder, we can see Charts.yaml, values.yaml, Charts folder, templates folder etc.
helm dependencies build	Build is used to reconstruct a chart's dependencies
helm install <b>[NAME] [CHART]</b>	Install the manifests mentioned in the [CHART] with a given release name inside [NAME]
helm upgrade <b>[NAME] [CHART]</b>	Upgrades a specified release to a new version of a chart
helm history <b>[NAME]</b>	Prints historical revisions for a given release.
helm rollback <b>[NAME] [REVISION]</b>	Roll back a release to a previous revision. The first argument of the rollback command is the name of a release, and the second is a revision (version) number. If this argument is omitted, it will roll back to the previous release.
helm uninstall <b>[NAME]</b>	Removes all of the resources associated with the last release of the chart as well as the release history
helm template <b>[NAME] [CHART]</b>	Render chart templates locally along with the values and display the output.
helm list	This command lists all of the releases for a specified namespace

# MICROSERVICES WITH K8s LOADBALANCER SERVICE

eazy  
bytes



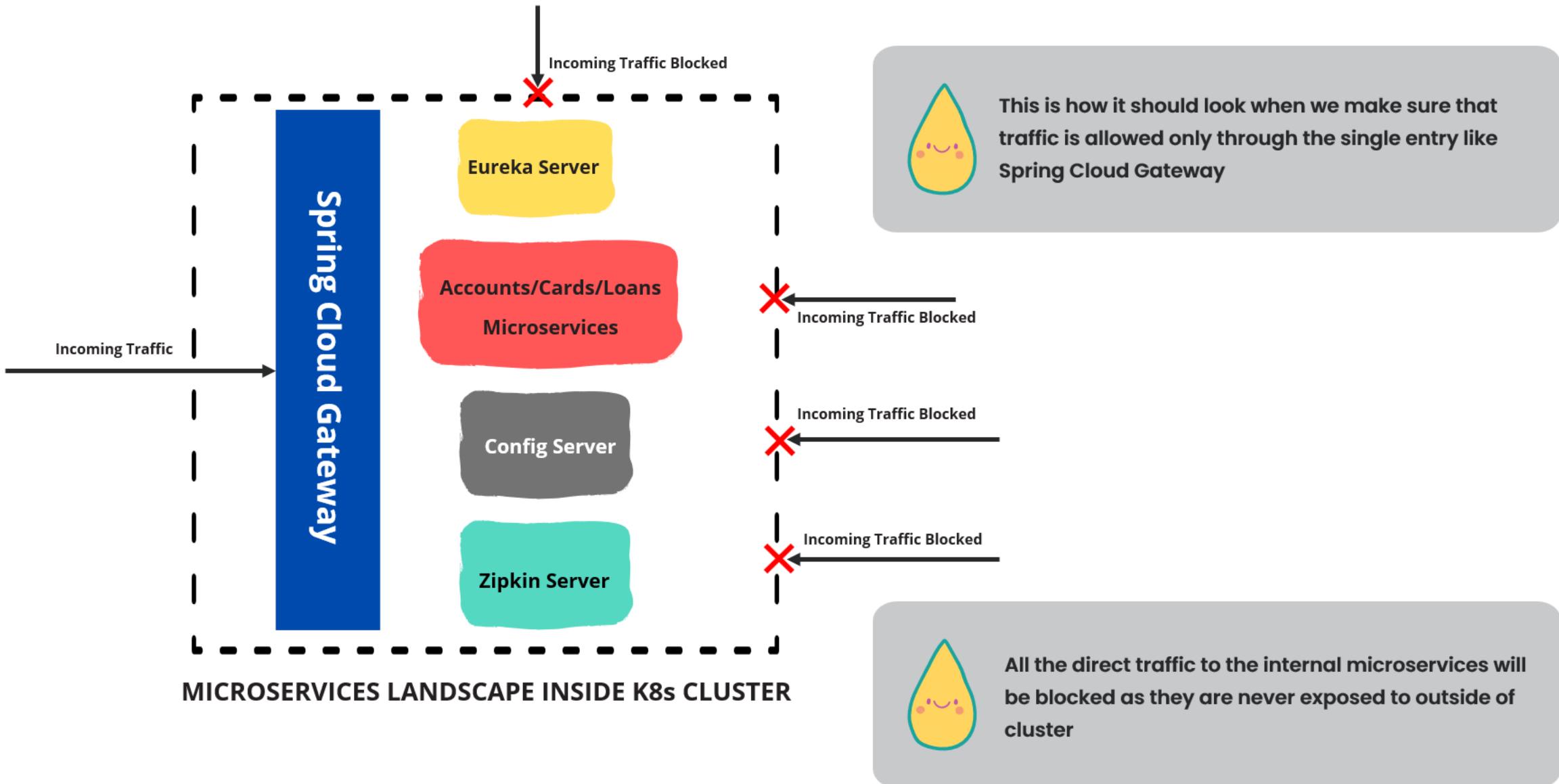
As of now with K8s LoadBalancer Service, all our microservices are being exposed to outside world. This is not a production recommended setup.

All the traffic into K8s cluster need to come from only through a single entry like Spring Cloud Gateway or K8s Ingress component.

As a first step towards microservices security, we should not expose our microservices to public & all the communication should happen through API Gateway

# SECURING MICROSERVICES WITH K8s SERVICES

eazy  
bytes



# KUBERNETES SERVICE TYPES

eazy  
bytes

Below 3 Kubernetes Service types are used majorly inside the K8s clusters.

## ClusterIP Service

This is the default service that uses an internal Cluster IP to expose Pods. In ClusterIP, the services are not available for external access of the cluster and used for internal communications between different Pods or microservices in the cluster.

## NodePort Service

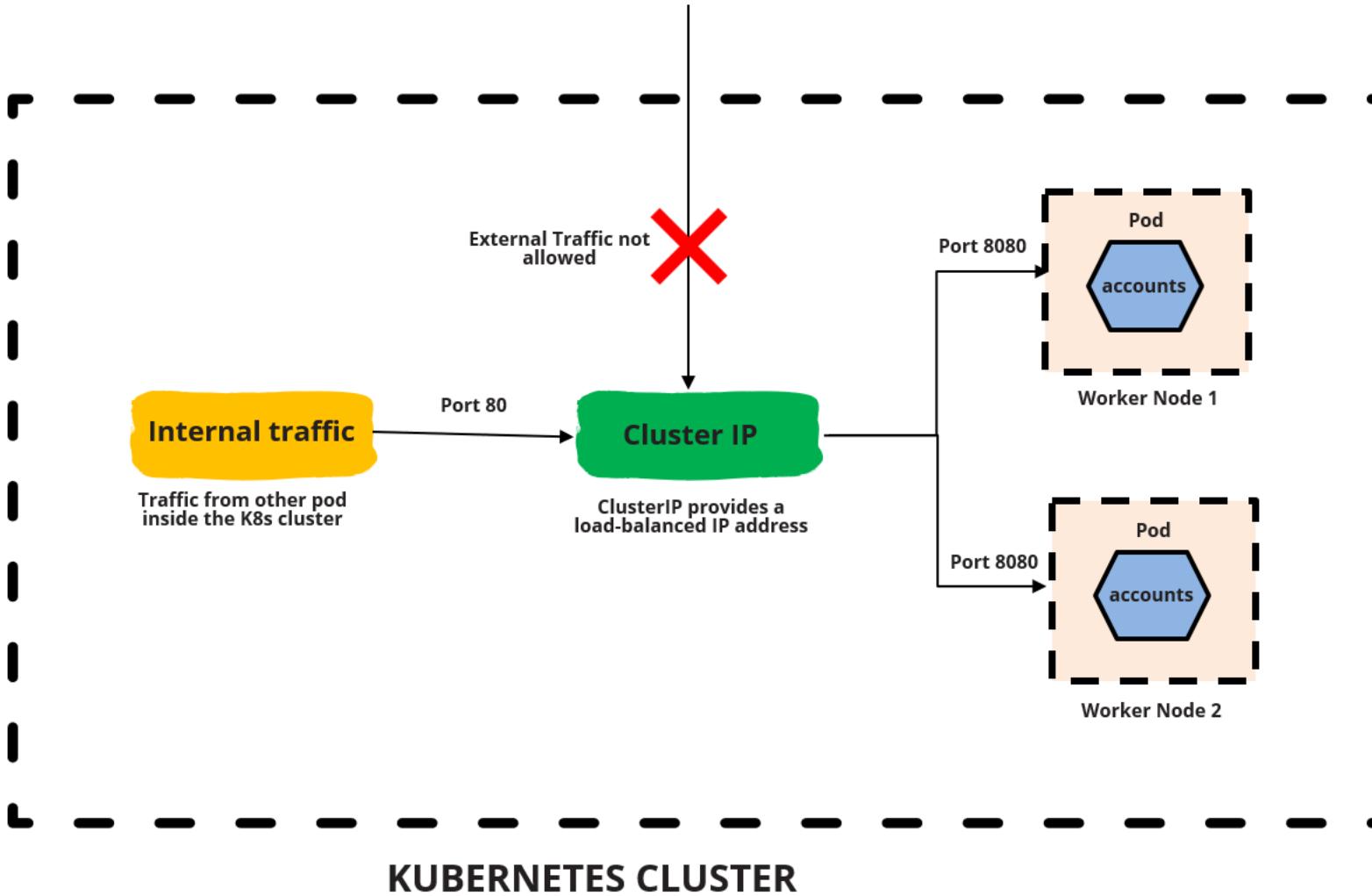
This service exposes outside and allows the outside traffic to connect to K8s Pods through the node port which is the port opened at Node end. The Pods can be accessed from external using `<Nodelp>:<Nodeport>`

## LoadBalancer Service

This service is exposed like in NodePort but creates a load balancer in the cloud where K8s is running that receives external requests to the service. It then distributes them among the cluster nodes using NodePort.

# K8s CLUSTER IP SERVICE

ClusterIP service creates an internal IP address for use within the K8s cluster. Good for internal-only applications that support other workloads within the cluster.

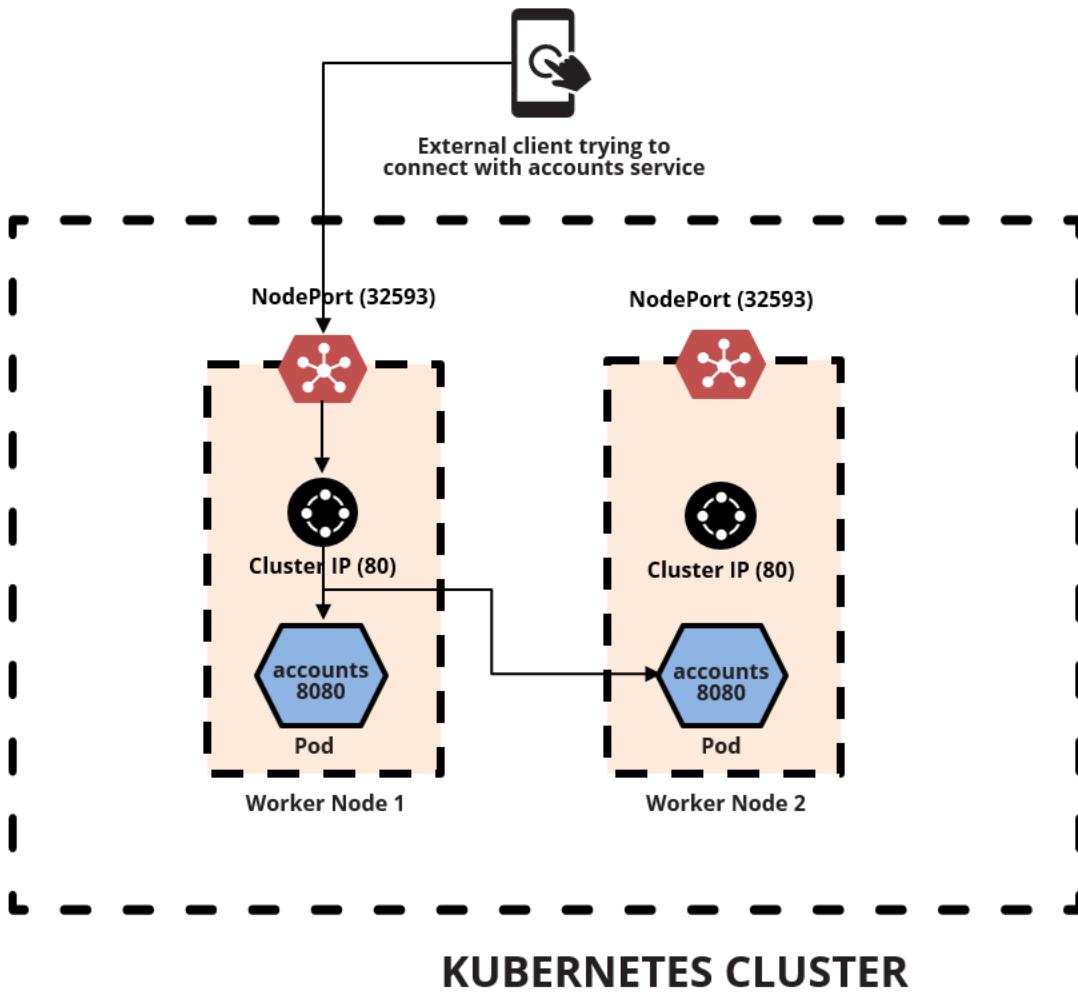


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
        - name: accounts
          image: eazybytes/accounts:latest
          ports:
            - containerPort: 8080
```

```
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

# K8s NODEPORT SERVICE

Services of type *NodePort* build on top of *ClusterIP* type services by exposing the *ClusterIP* service outside of the cluster on high ports (default 30000-32767). If no port number is specified then Kubernetes automatically selects a free port. The local *kube-proxy* is responsible for listening to the port on the node and forwarding client traffic on the *NodePort* to the *ClusterIP*.

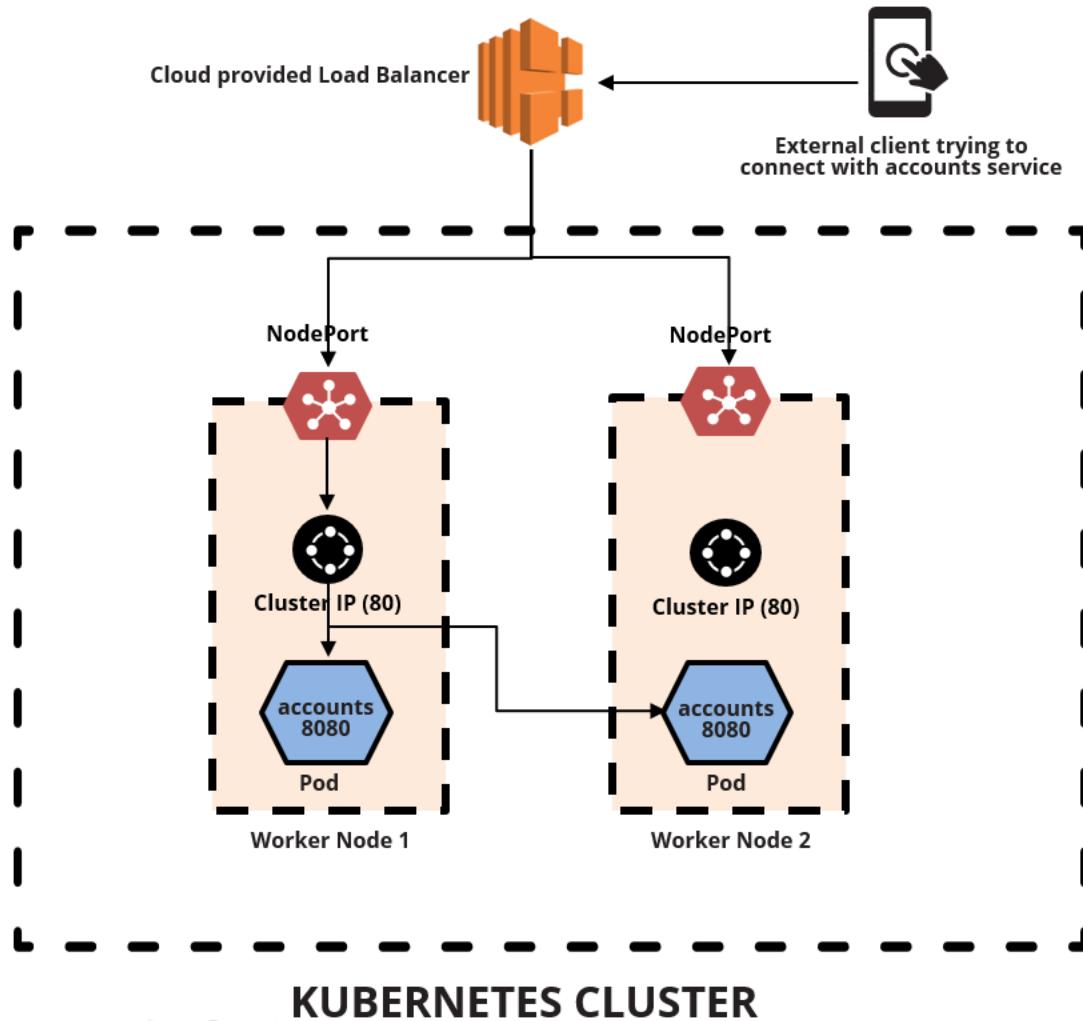


```
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 32593
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
        - name: accounts
          image: eazybytes/accounts:latest
          ports:
            - containerPort: 8080
```

# K8s LOADBALANCER SERVICE

The LoadBalancer service type is built on top of NodePort service types by provisioning and configuring external load balancers from public and private cloud providers. It exposes services that are running in the cluster by forwarding layer 4 traffic to worker nodes. This is a dynamic way of implementing a case that involves external load balancers and NodePort type services.

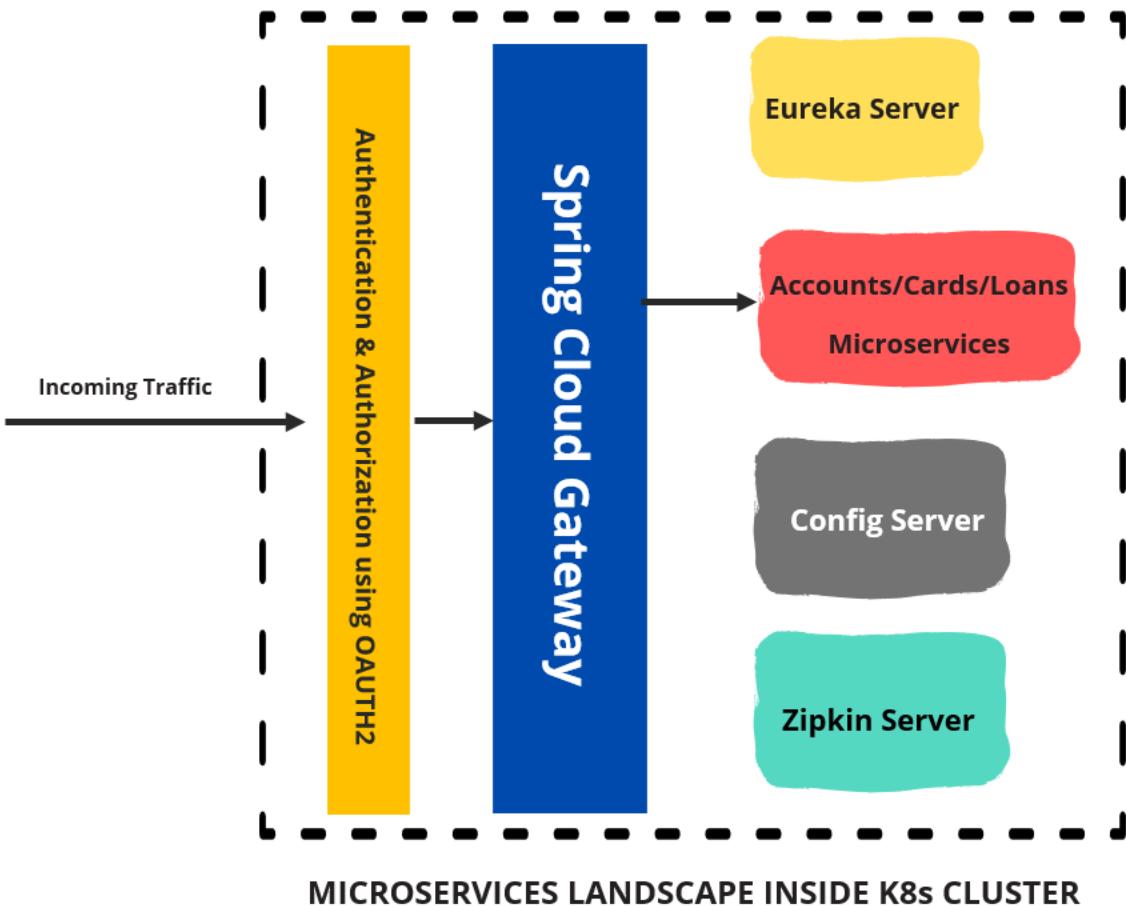


```
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
        - name: accounts
          image: eazybytes/accounts:latest
          ports:
            - containerPort: 8080
```

# SECURING SPRING CLOUD GATEWAY

eazy  
bytes



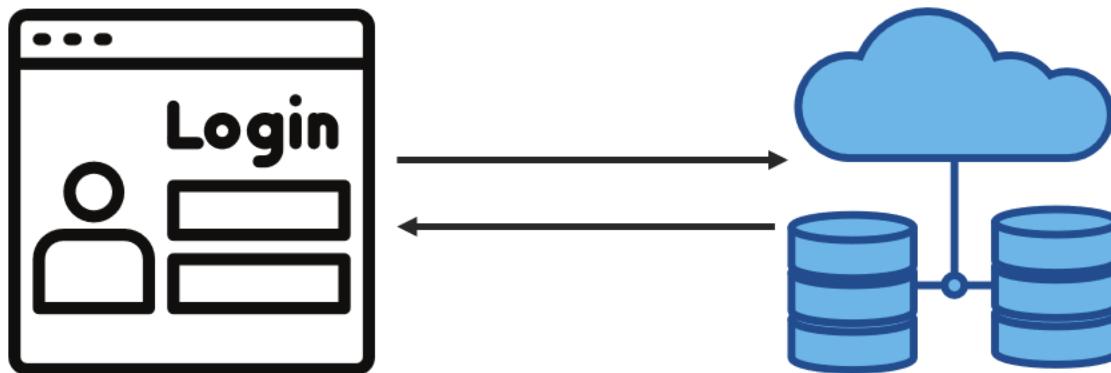
To completely secure our microservices, we need to implement authentication & authorization at Spring Cloud Gateway as it is the only entry point into your cluster

In order to secure gateway, we can implement **OAuth2** which is a industry standard framework in terms of Authentication & Authorization

To implement security with the help of OAuth2 framework, we can take help from **Keycloak** which is a famous Open Source Identity and Access Management

# WHY OAUTH2 ?

Why should we use OAuth2 framework for implementing security inside our microservices ? Why can't we use the basic authentication ?  
To answer this, first lets try to understand the basic authentication & its drawbacks.



 Early websites usually ask for credentials via an HTML form, which the browser will send to the server. The server authenticates the information and writes a session value in the cookie; as long as the session is still marked active, user can access protected features and resources.

## Drawbacks of Basic authentication



Backend server or business logic is tightly coupled with the Authentication/Authorization logic. Not mobile flow/ REST APIs friendly



Basic authentication flow does not accommodate well the use case where users of one product or service would like to grant third-party clients access to their information on the platform.

# Advantages of OAuth2

## Supports all kinds of Apps

✓ OAuth2 is a delegated authorization framework which supports multiple use cases addressing different device capabilities. It supports server-to-server apps, browser-based apps, mobile/native apps, IoT devices and consoles/TVs.

✓ OAuth2 has various authorization grant flows like Authorization Code grant, Client Credentials Grant Type etc. to support all kinds of apps communication,

## Separation of Auth logic

✓ Inside OAuth2, we have **Authorization Server** which receives requests from the Client for Access Tokens and issues them upon successful authentication. This enable us to maintain all the security logic in a single place. Regardless of how many applications an organization has, they all can connect to Auth server to perform login operation.

✓ All user credentials & client application credentials will be maintained in a single location which is inside Auth Server.

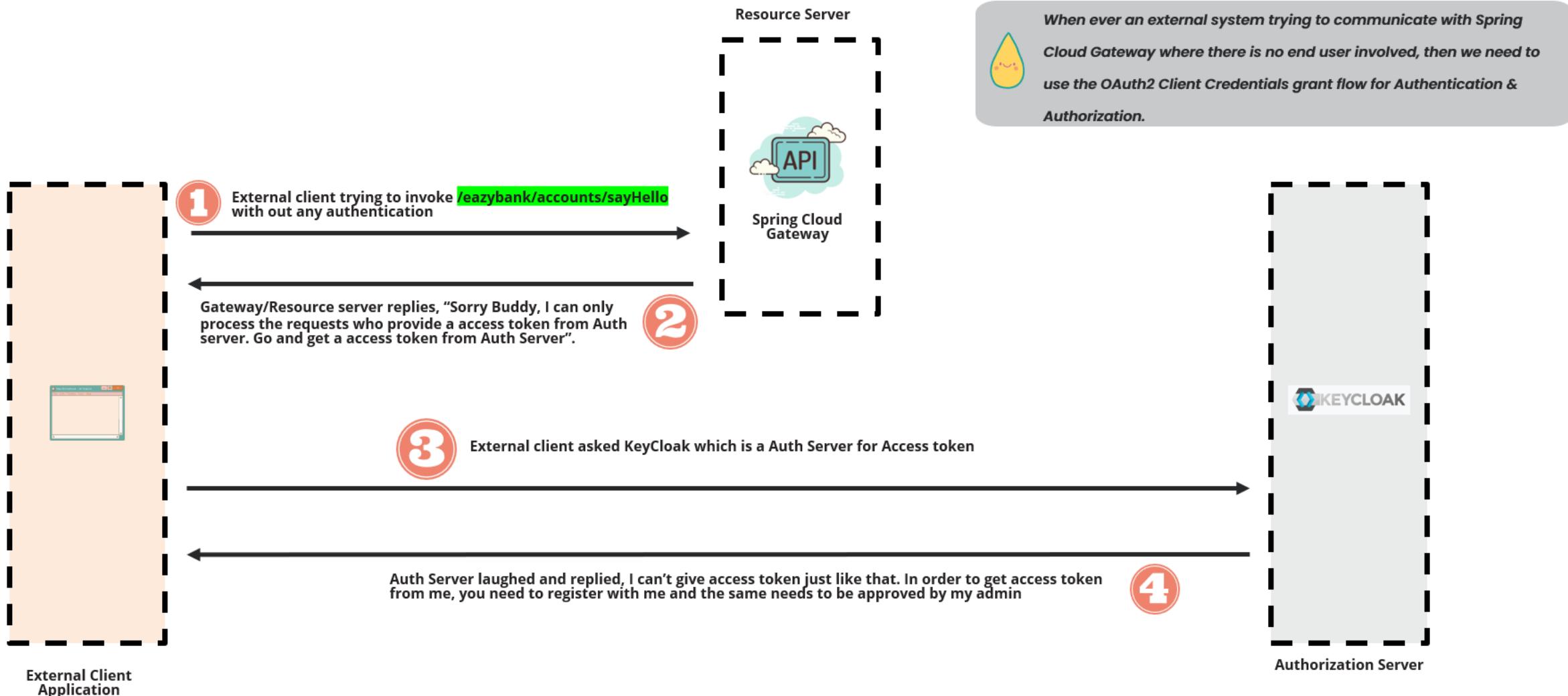
## No need to share Credentials

✓ If you plan to allow a third-party applications and services to access your resources, then there is no need to share your credentials.

✓ In many ways, you can think of the OAuth2 token as a “access card” at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

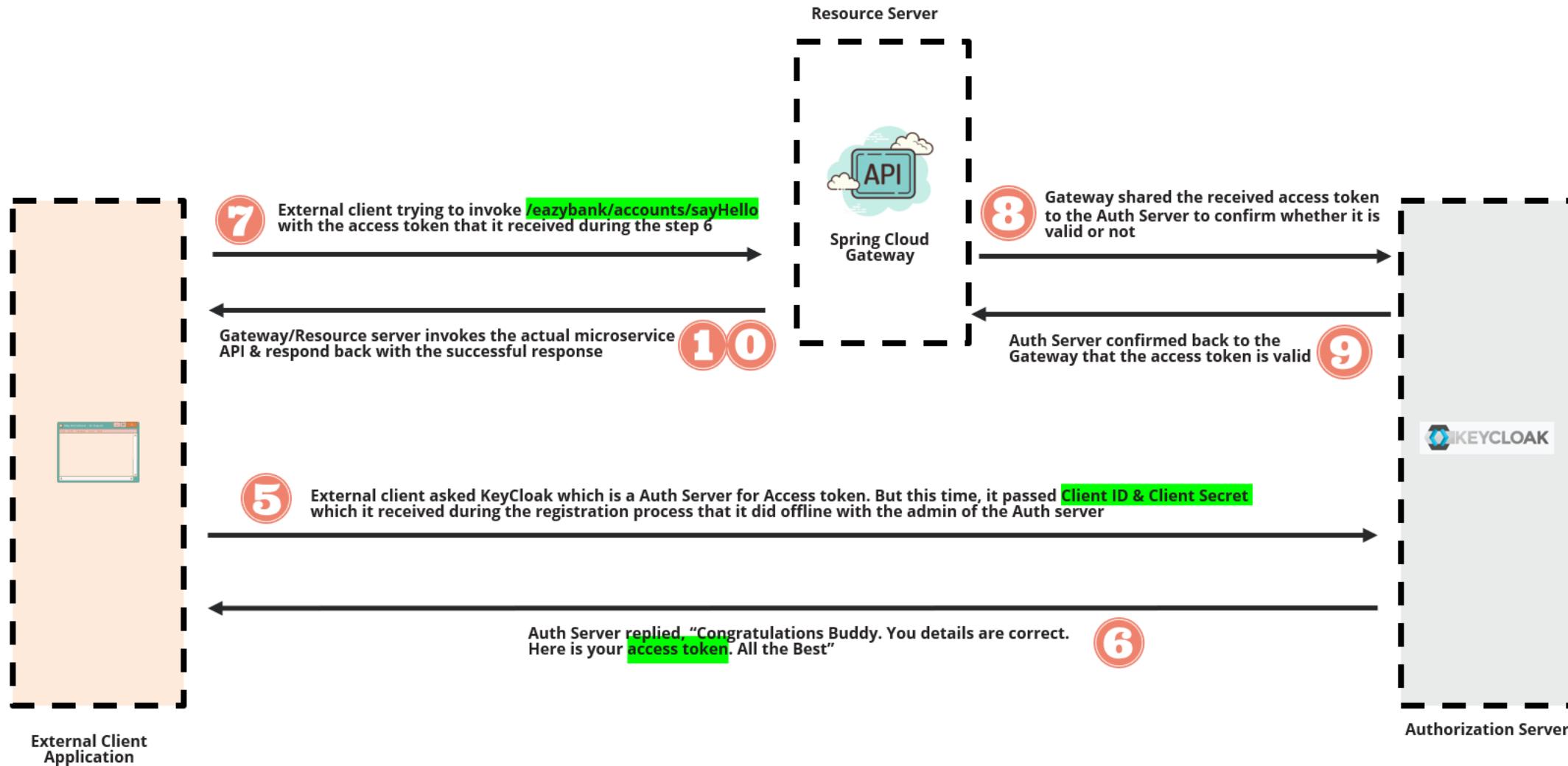
# SECURING SPRING CLOUD GATEWAY USING OAUTH2 CLIENT CREDENTIALS FLOW

eazy  
bytes



# SECURING SPRING CLOUD GATEWAY USING OAUTH2 CLIENT CREDENTIALS FLOW

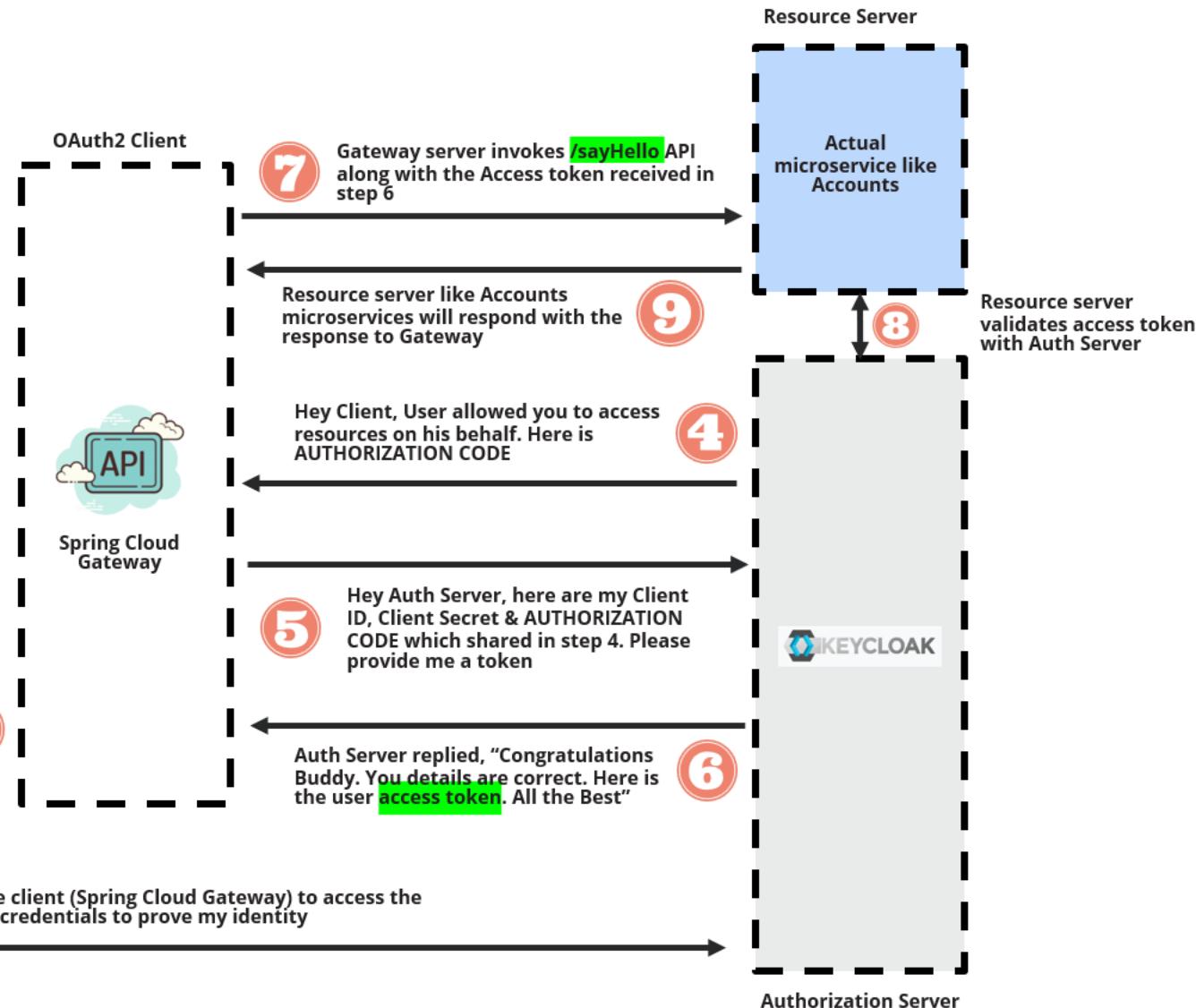
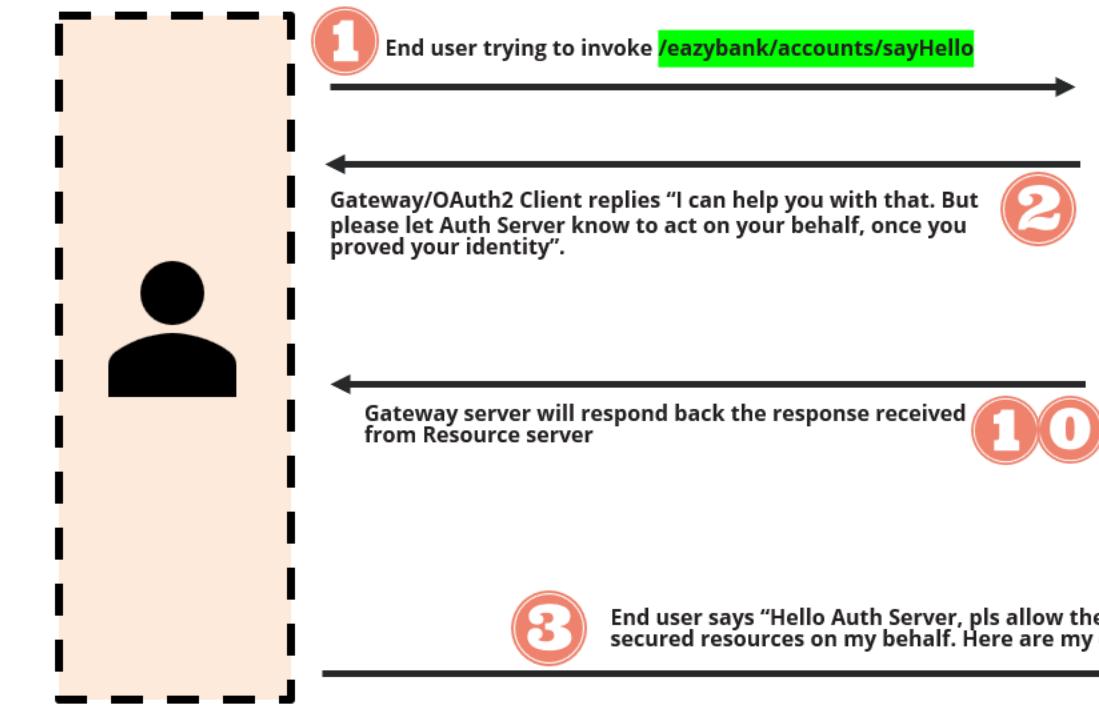
eazy  
bytes



# SECURING SPRING CLOUD GATEWAY USING OAUTH2 AUTHORIZATION CODE FLOW

eazy  
bytes

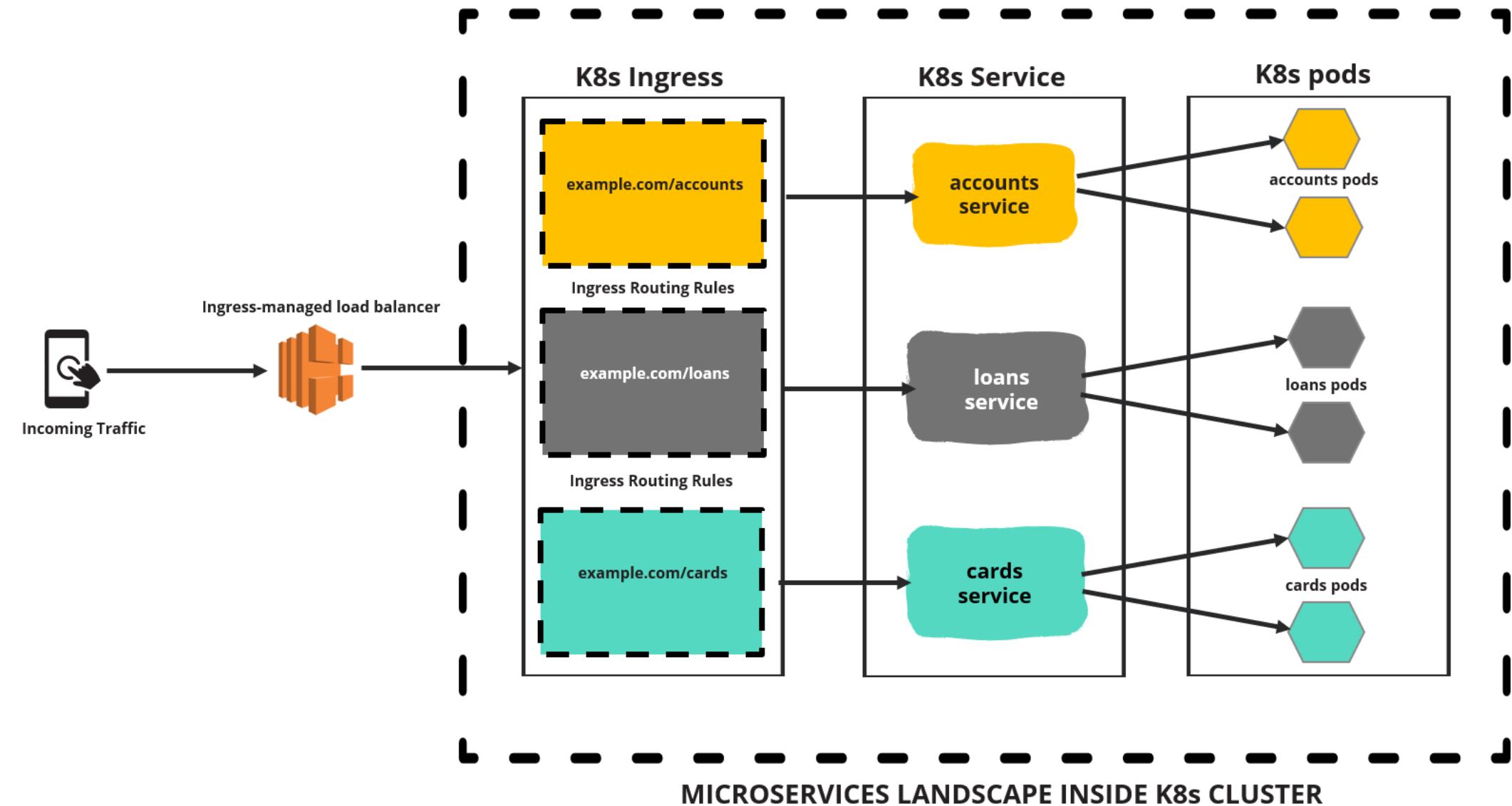
 When ever an end user involved & trying to communicate with Spring Cloud Gateway, then we need to use the OAuth2 Authorization Code grant flow for Authentication & Authorization.



# K8s INGRESS

## IN THE PLACE OF SPRING CLOUD GATEWAY

eazy  
bytes



# K8s INGRESS

IN THE PLACE OF SPRING CLOUD GATEWAY

eazy  
bytes

K8s Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers available in the market.

K8s Ingress evaluates all the routing rules and manage the redirections. But it can't handle all kinds of complex routings like Spring Cloud Gateway.

Just like Spring Cloud gateway K8s ingress can acts as an entry point/edge server of K8s cluster.

K8s Ingress can handle authentication, authorization, SSL/TLS termination.

Setting up of K8s Ingress is more of a DevOps team responsibility rather than Dev team.

# INTRO TO SERVICE MESH

eazy  
bytes

Service mesh helps organizations run distributed, microservices-based apps anywhere. Why use it ? It enables organizations to secure, connect, and monitor microservices.

Service mesh manages traffic flows between services, enforces access policies, and aggregates telemetry data, all without requiring changes to application code.

Service mesh can help on security at the application level with strong identity-based authentication, authorization, and encryption.

Service mesh can help with fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.

Service mesh can help with monitoring by automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress

Service mesh can secure internal service-to-service communication in a cluster with Mutual TLS (mTLS)

# INTRO TO SERVICE MESH

eazy  
bytes

*Problem that Service Mesh trying to solve*



Each microservice has lot of code/configurations related to Security, tracing etc. which is not related to business logic. All the extra code/configurations that is present inside each microservice will make them complex.

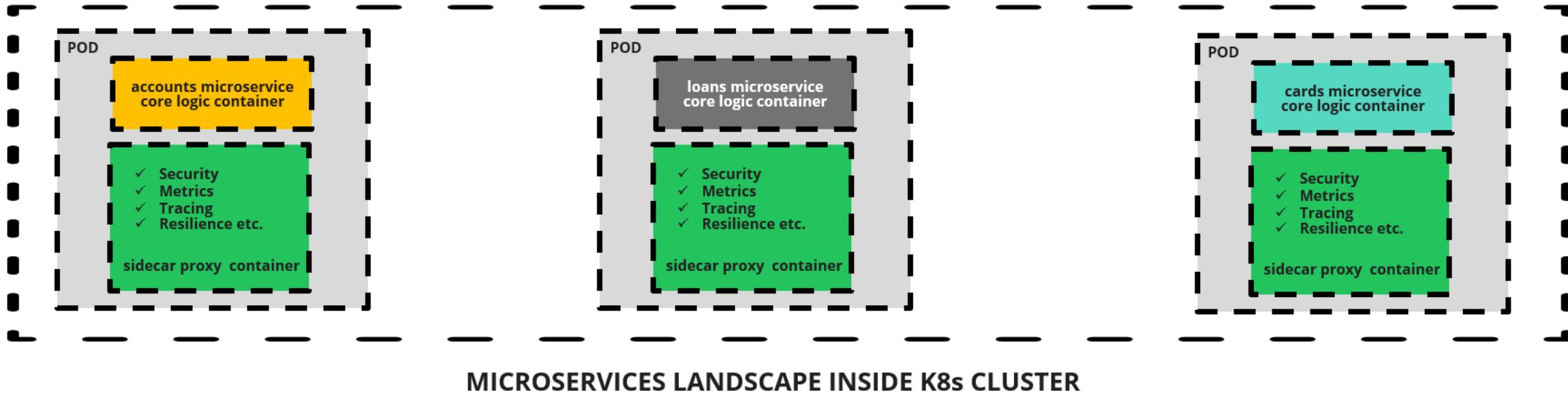


Developers needs to manage these changes consistently in all the microservices which deviates from their main focus of building the business logic.

# INTRO TO SERVICE MESH

eazy  
bytes

How Service Mesh makes our life easy while building microservices



A sidecar proxy container is deployed along with each service that you start in your cluster. This is also called as Sidecar pattern. This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, being created and retired alongside the parent.

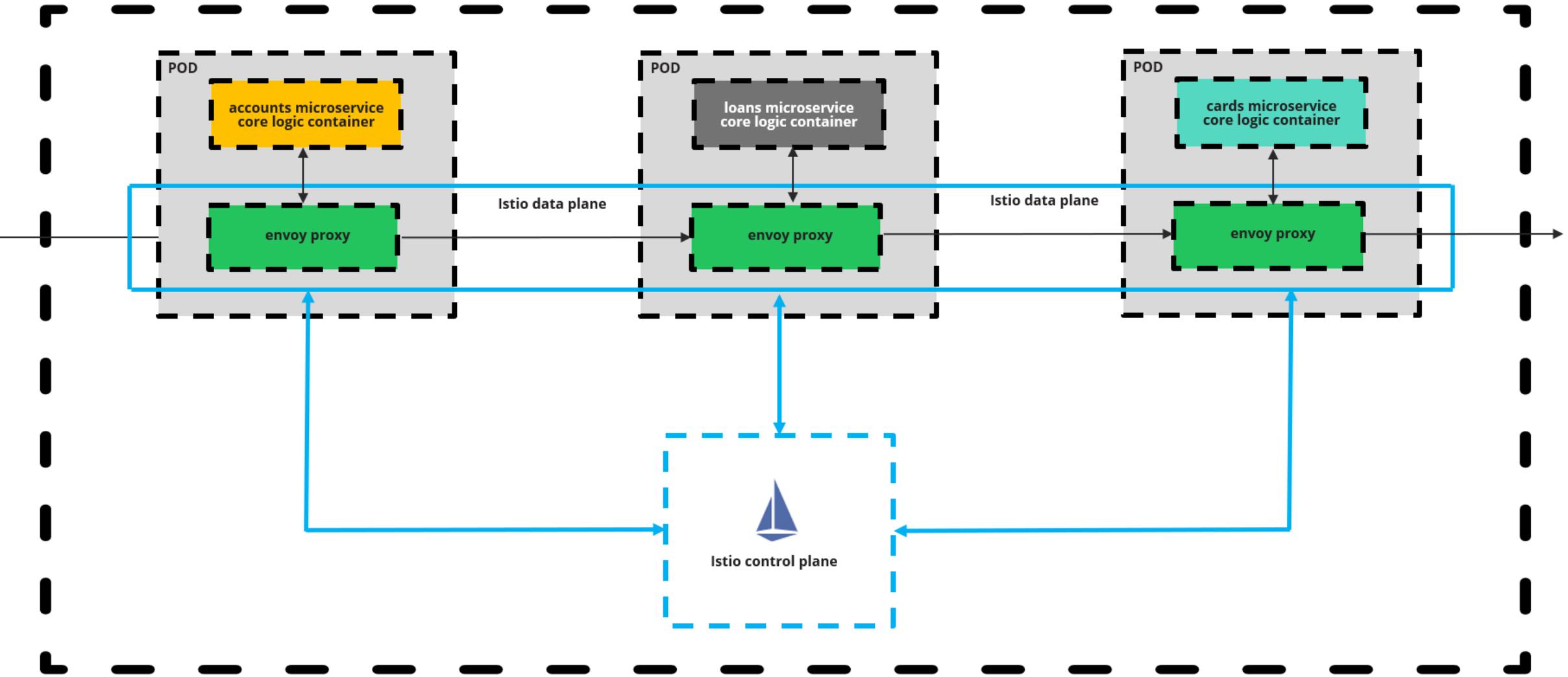


A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language. Developers will also be relieved from developing all these non business logic related components.

# THE ISTIO SERVICE MESH

eazy  
bytes

*How Service Mesh makes our life easy while building microservices*



# THE ISTIO SERVICE MESH

eazy  
bytes

## How Istio service mesh works

- *Istio has two components: the data plane and the control plane.*
- *The data plane is the communication between services. Without a service mesh, the network doesn't understand the traffic being sent over, and can't make any decisions based on what type of traffic it is, or who it is from or to.*
- *Service mesh uses a proxy to intercept all your network traffic, allowing a broad set of application-aware features based on configuration you set.*
- *An Envoy proxy is deployed along with each service that you start in your cluster, or runs alongside services running on VMs.*
- *The control plane takes your desired configuration, and its view of the services, and dynamically programs the proxy servers, updating them as the rules or the environment changes.*

# THANK YOU & CONGRATULATIONS

YOU ARE NOW A MASTER OF MICROSERVICES USING SPRING, DOCKER AND KUBERNETES

eazy  
bytes

