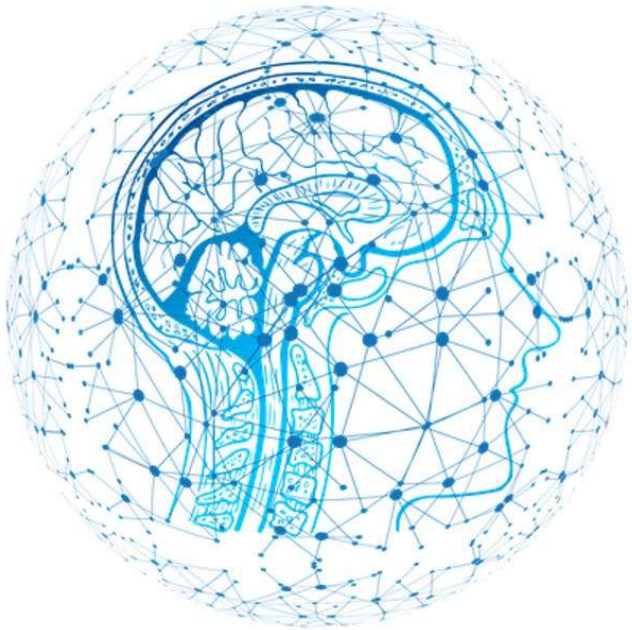
eazy bytes



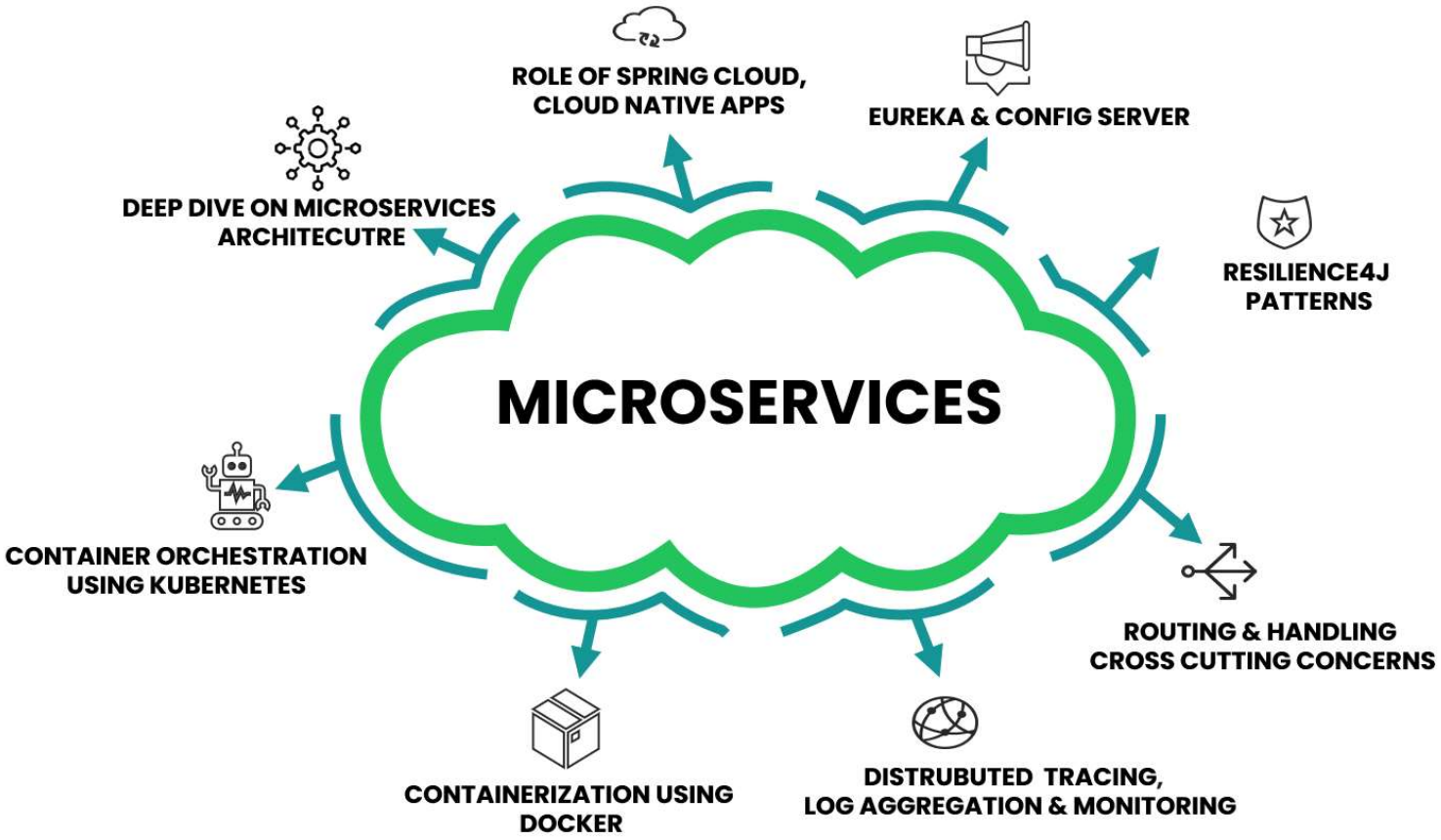
MICROSERVICES

USING SPRING, DOCKER, KUBERNETES

MICROSERVICES WITH SPRING, DOCKER, KUBERNETES

eazy bytes

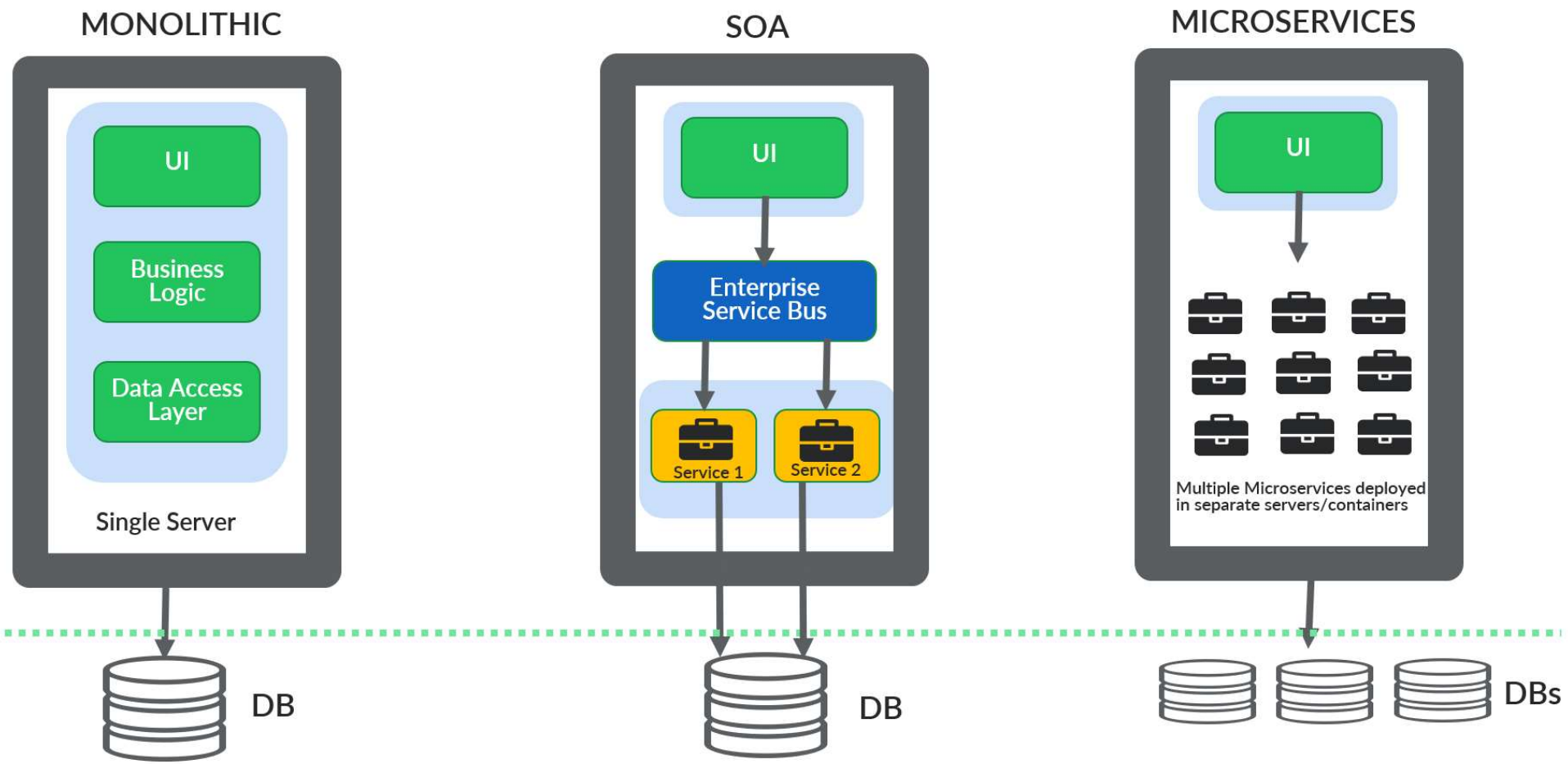
WHAT WE COVER IN THIS COURSE



MONOLITHIC VS SOA VS MICROSERVICES

eazy bytes

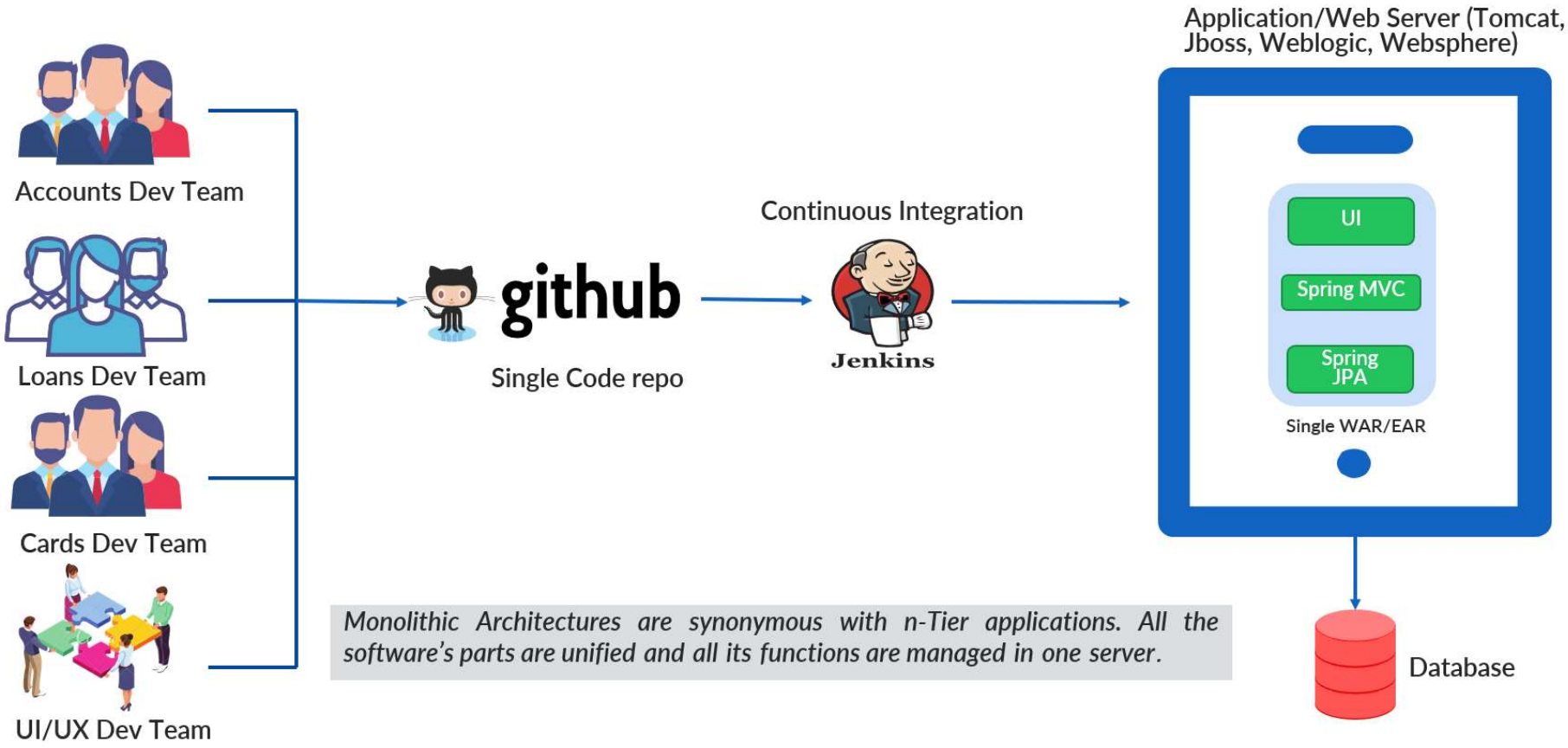
EVOLUTION OF MICROSERVICES



MONOLITHIC ARCHITECTURE

SAMPLE BANK APPLICATION

eazy bytes

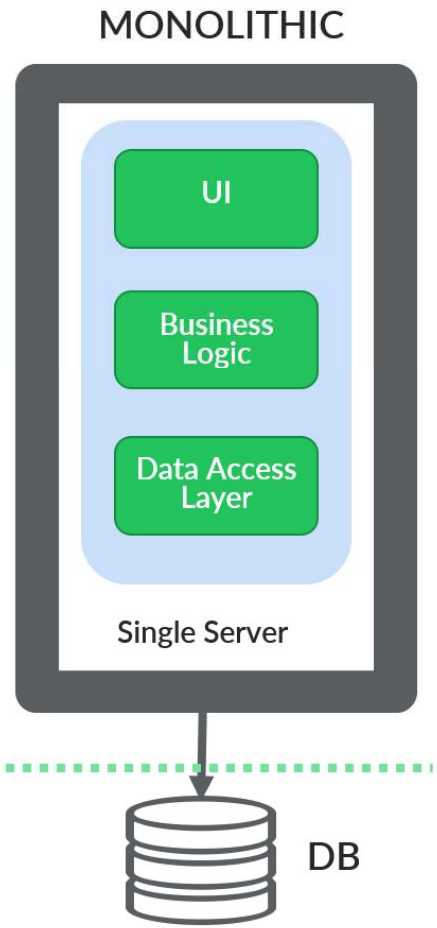


Monolithic Architectures are synonymous with n-Tier applications. All the software's parts are unified and all its functions are managed in one server.

MONOLITHIC ARCHITECTURE

PROS & CONS

eazy bytes



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

UI/UX Dev Team

github

UI Code repo

Jenkins

UI Code

Enterprise Service Bus

Accounts Dev Team

Loans Dev Team

Cards Dev Team

github

Backend Services repo

Jenkins

REST/SOAP Services

Spring JPA

App/Web Server

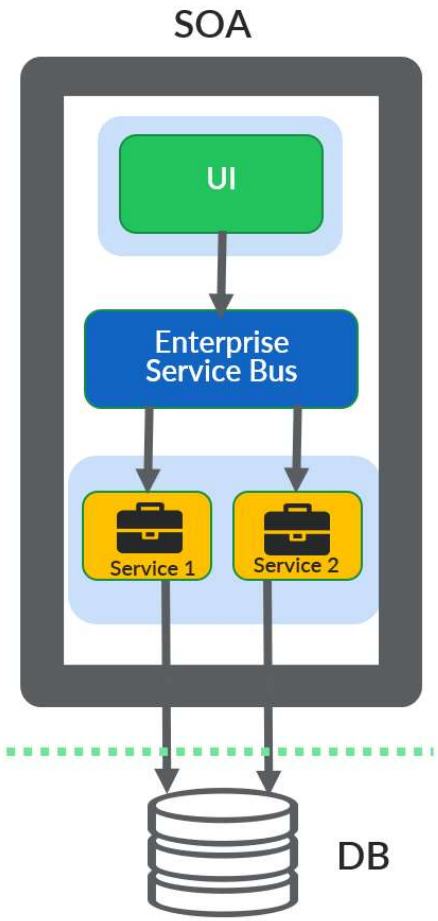
Database

SOA Services exposed with a standard protocol, such as SOAP and consumed/reused by other services – leverage messaging middleware.

SOA ARCHITECTURE

PROS & CONS

eazy bytes



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

UI/UX Dev Team

github

UI Code repo

Jenkins

UI Web App

Invokes all the backend logic through REST API calls

Accounts Dev Team

github

Accounts Code repo

Jenkins

Accounts Microservice

Accounts DB

Loans Dev Team

github

Loans Code repo

Jenkins

Loans Microservice

Loans DB

Cards Dev Team

github

Cards Code repo

Jenkins

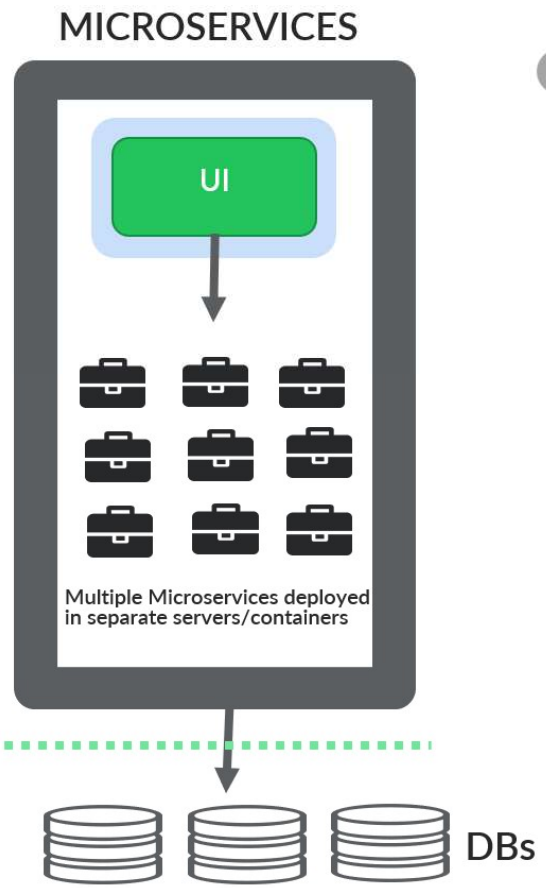
Cards Microservice

Cards DB

MICROSERVICES ARCHITECTURE

PROS & CONS

eazy bytes



Pros

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

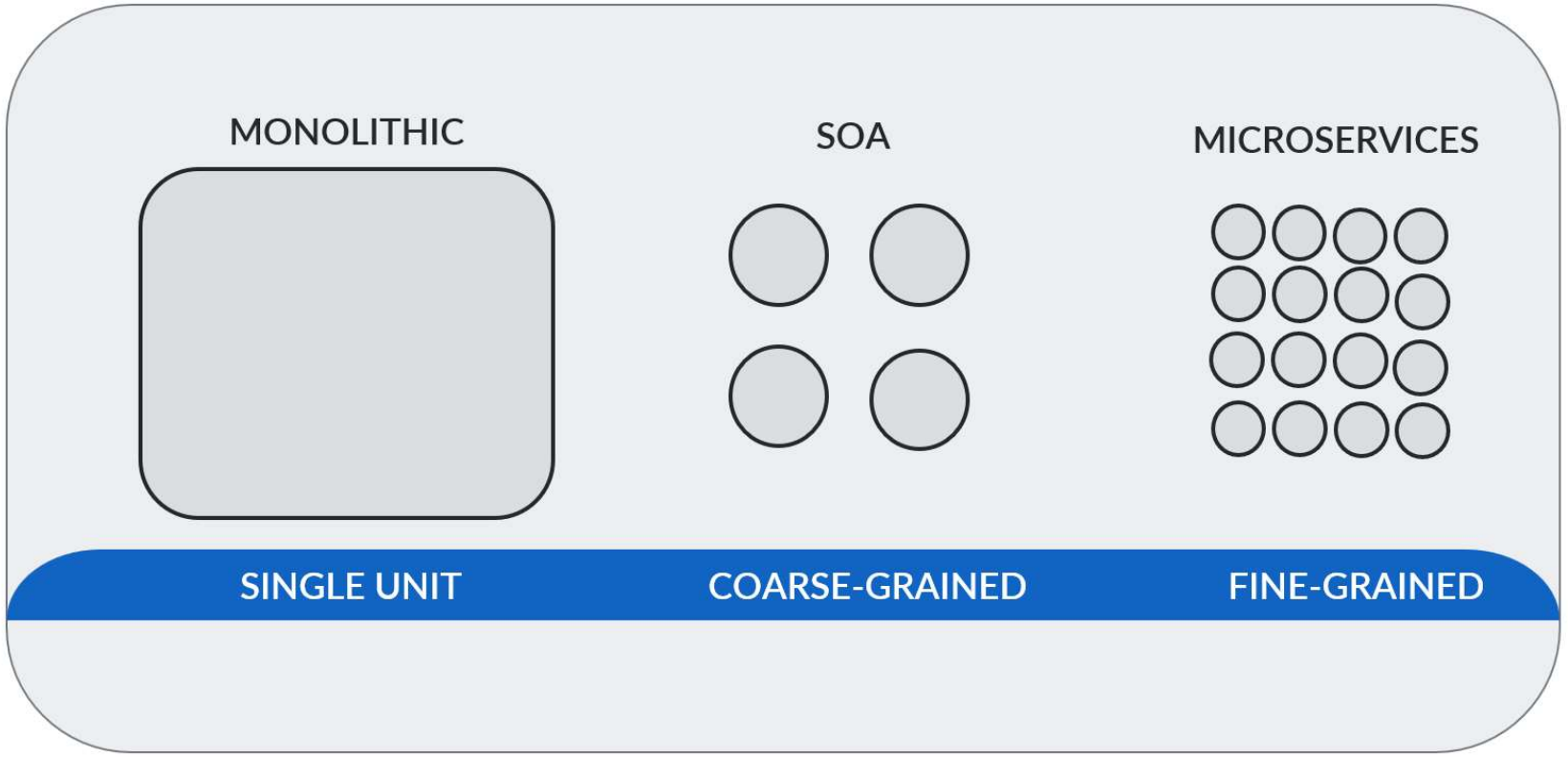
Cons

- Complexity
- Infrastructure overhead
- Security concerns

MONOLITHIC VS SOA VS MICROSERVICES

COMPARISION

eazy bytes



MONOLITHIC VS SOA VS MICROSERVICES

eazy bytes

COMPARISION

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

WHAT ARE MICROSERVICES?

eazy bytes

DEFINITION OF MICROSERVICES

"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?

eazy bytes

WHY SPRING IS THE BEST FRAMEWORK TO BUILD MICROSERVICE

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

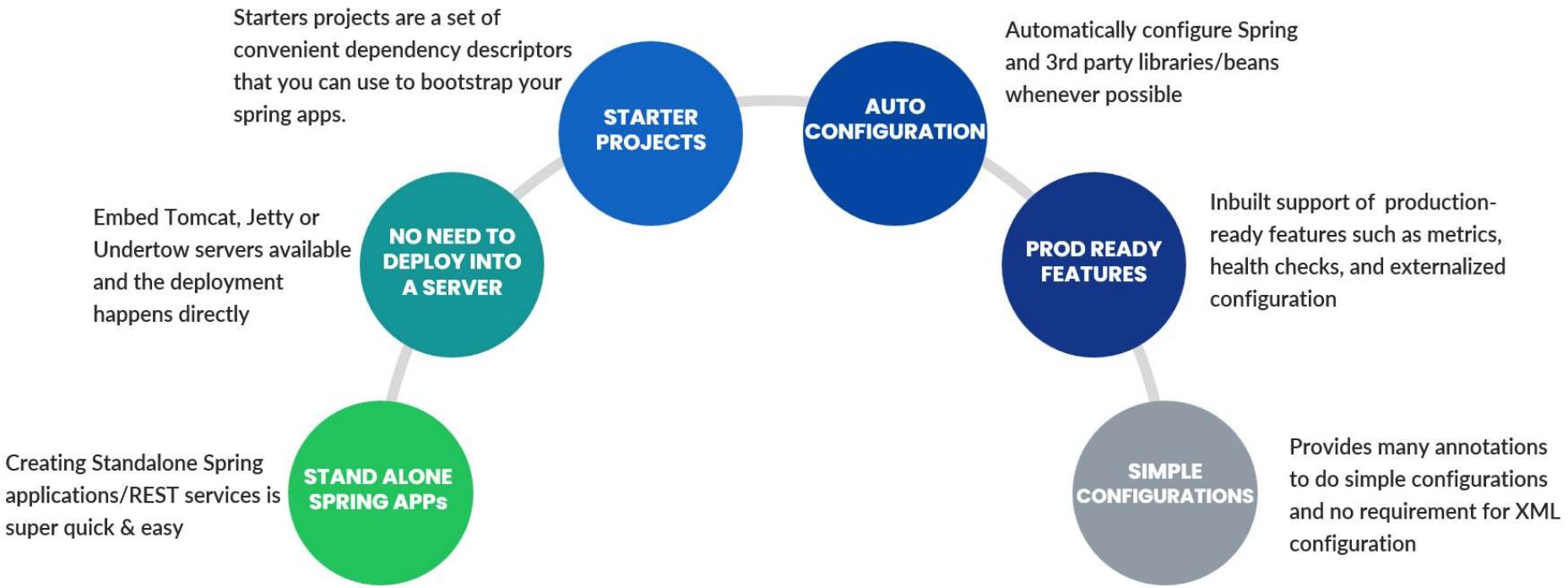
There is a large community of Spring developers who can help & adapt easily

WHAT IS SPRING BOOT?

eazy bytes

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".

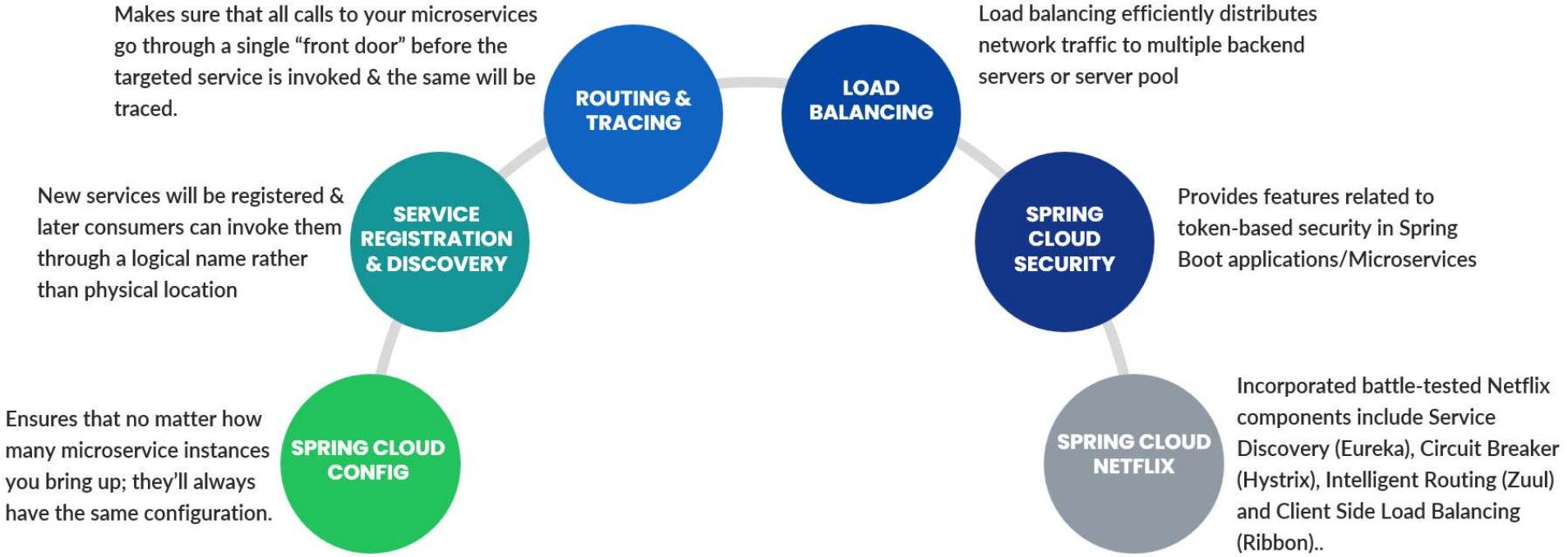


WHAT IS SPRING CLOUD?

eazy bytes

USING SPRING CLOUD FOR MICROSERVICES DEVELOPMENT

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

Saving Account

Trading Account

Cards

Loans

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

Trading Account

Debit Card

Credit Card

Home Loan

Vehicle Loan

Personal Loan

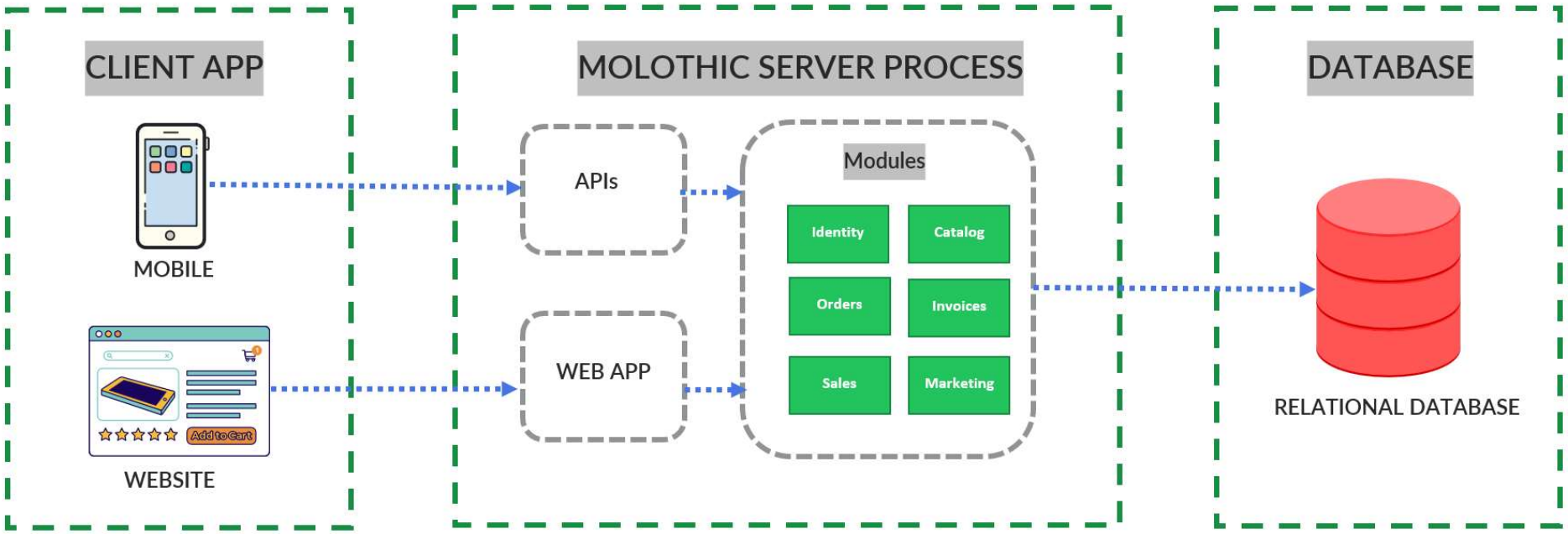
NOT CORRECT SIZING AS WE CAN SEE TOO MANY SERVICES UNDER LOANS & CARDS

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,

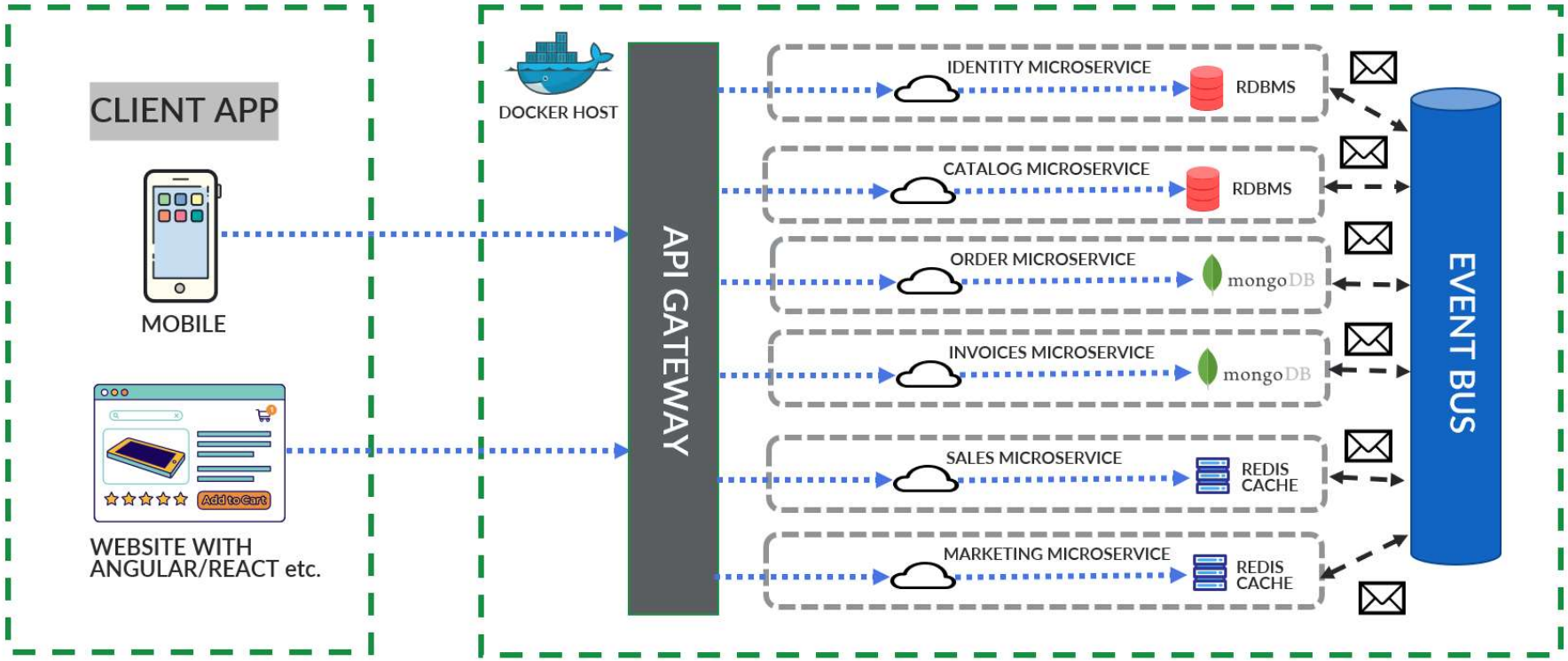
- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as possible and requires a full 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

eazy bytes

DEPLOYMENT, PORTABILITY & SCALABILITY

DEPLOYMENT

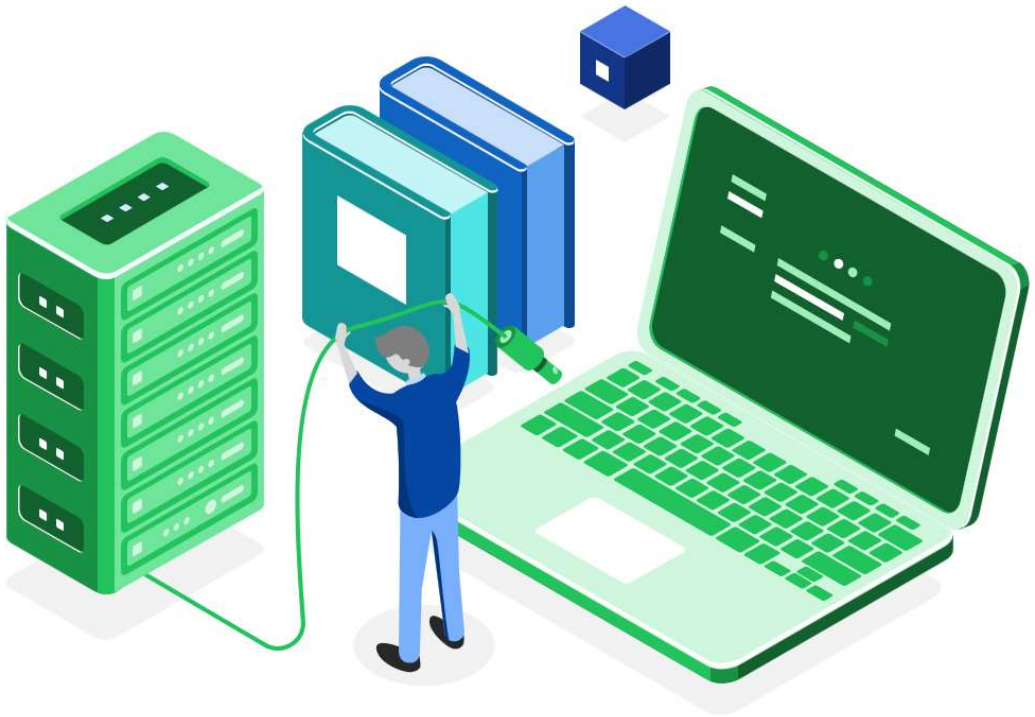
How do we deploy all the tiny 100s of microservices with less effort & cost?

PORTABILITY

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

SCALABILITY

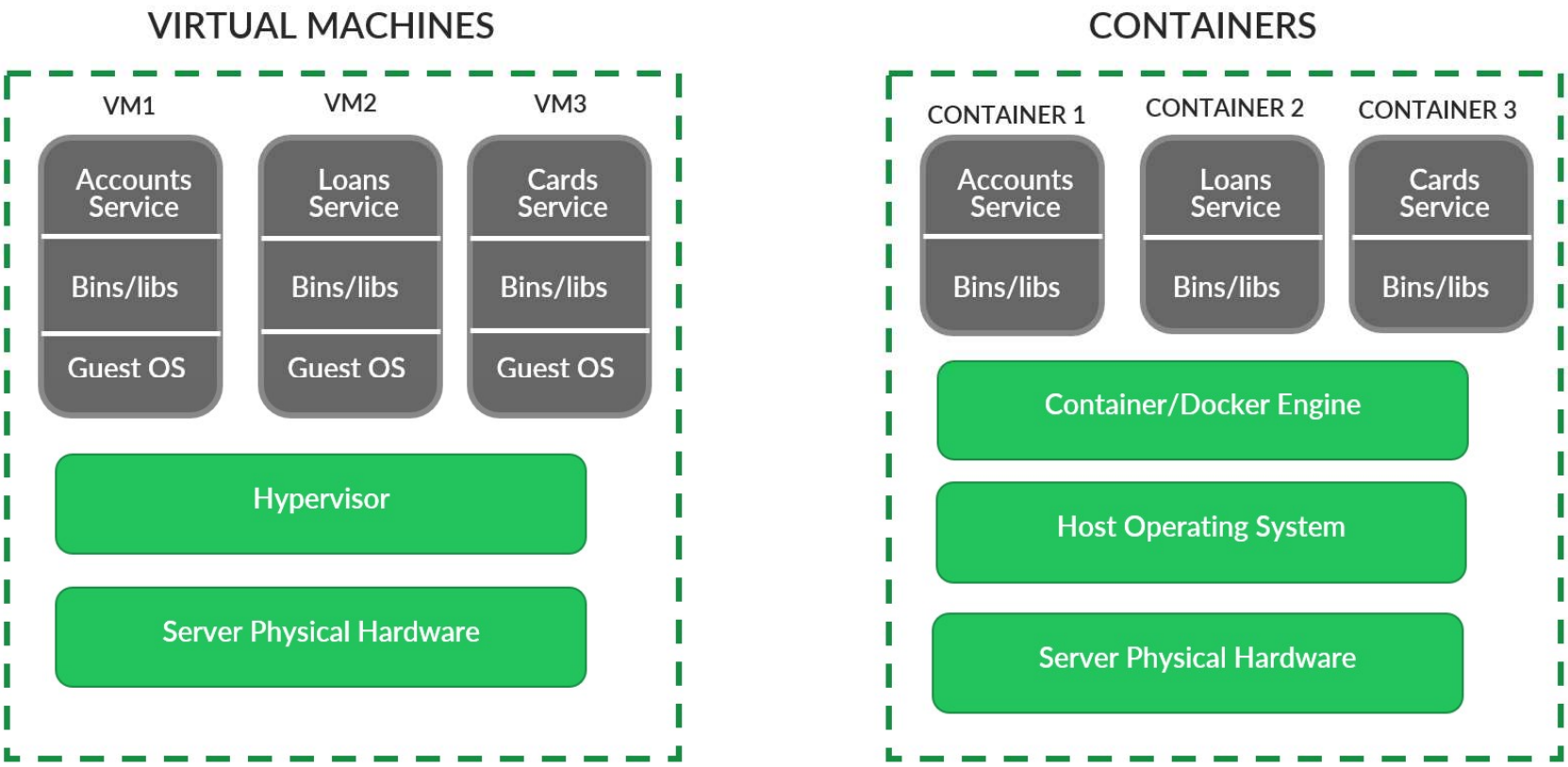
How do we scale our applications based on the demand on the fly with minimum effort & cost?



CONTAINERIZATION TECHNOLOGY

eazy bytes

USING DOCKER



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.

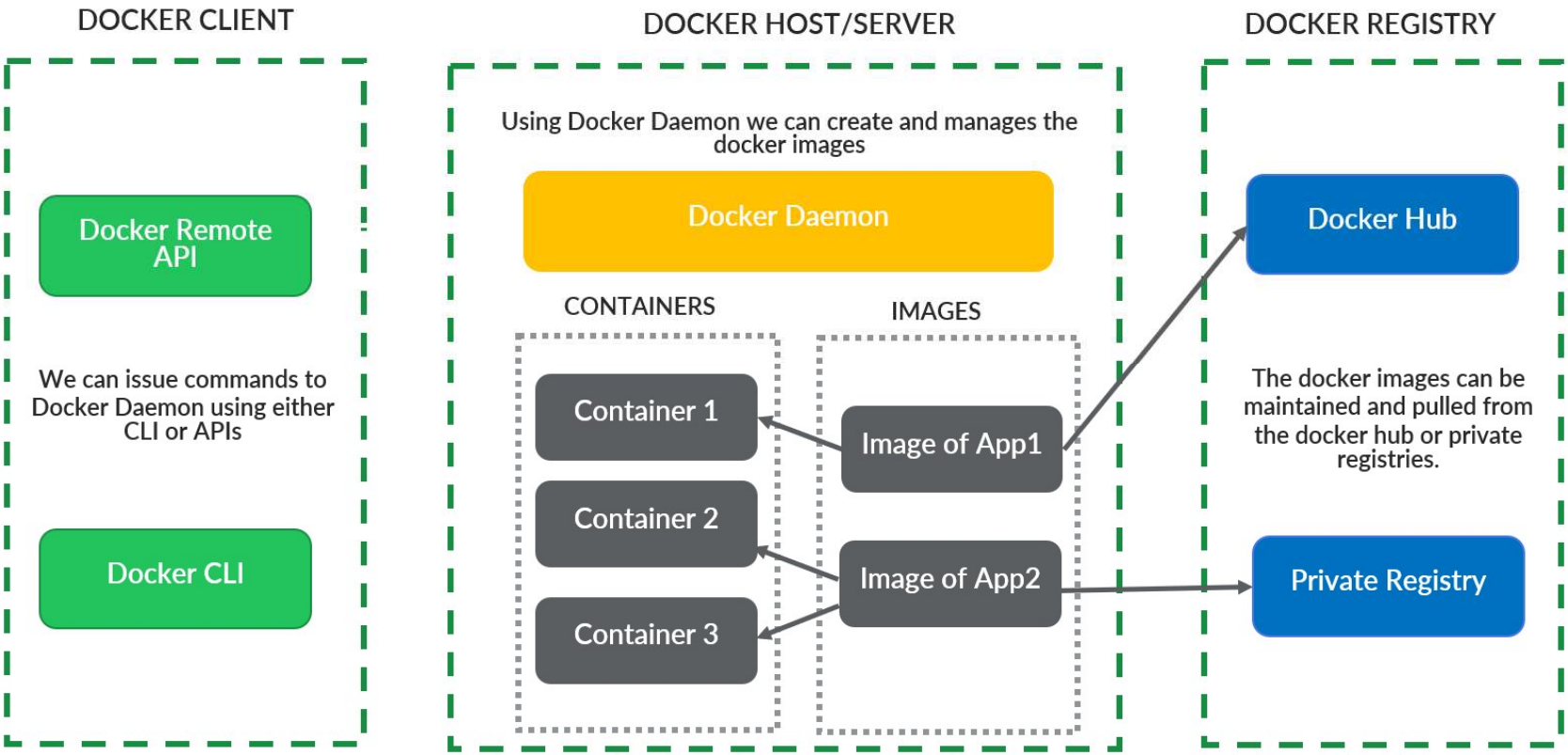
What is Docker?

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

DOCKER ARCHITECTURE

eazy bytes



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.

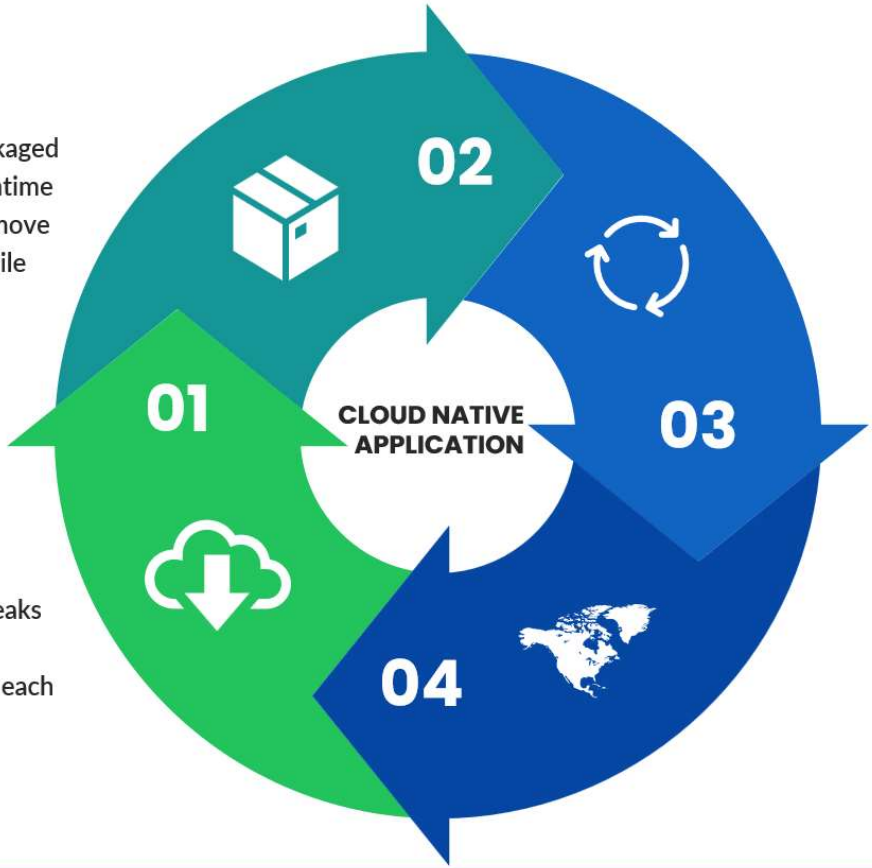
CLOUD-NATIVE APPLICATIONS

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.



DEVOPS

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

MICROSERVICE

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

CONTINOUS DELIVERY

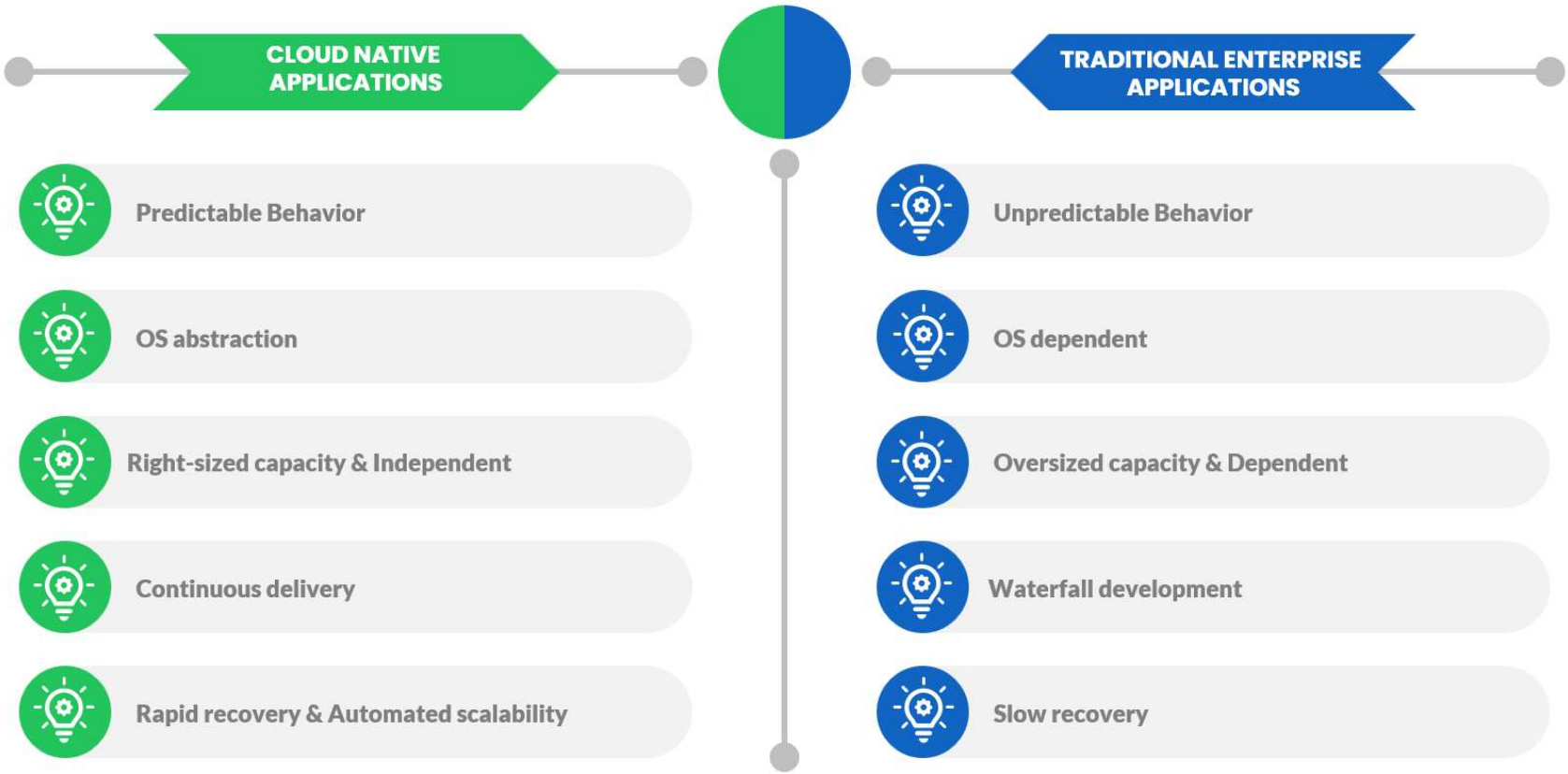
It's a software development practice in which the process of delivering software is automated to allow short-term deliveries into a production environment.

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

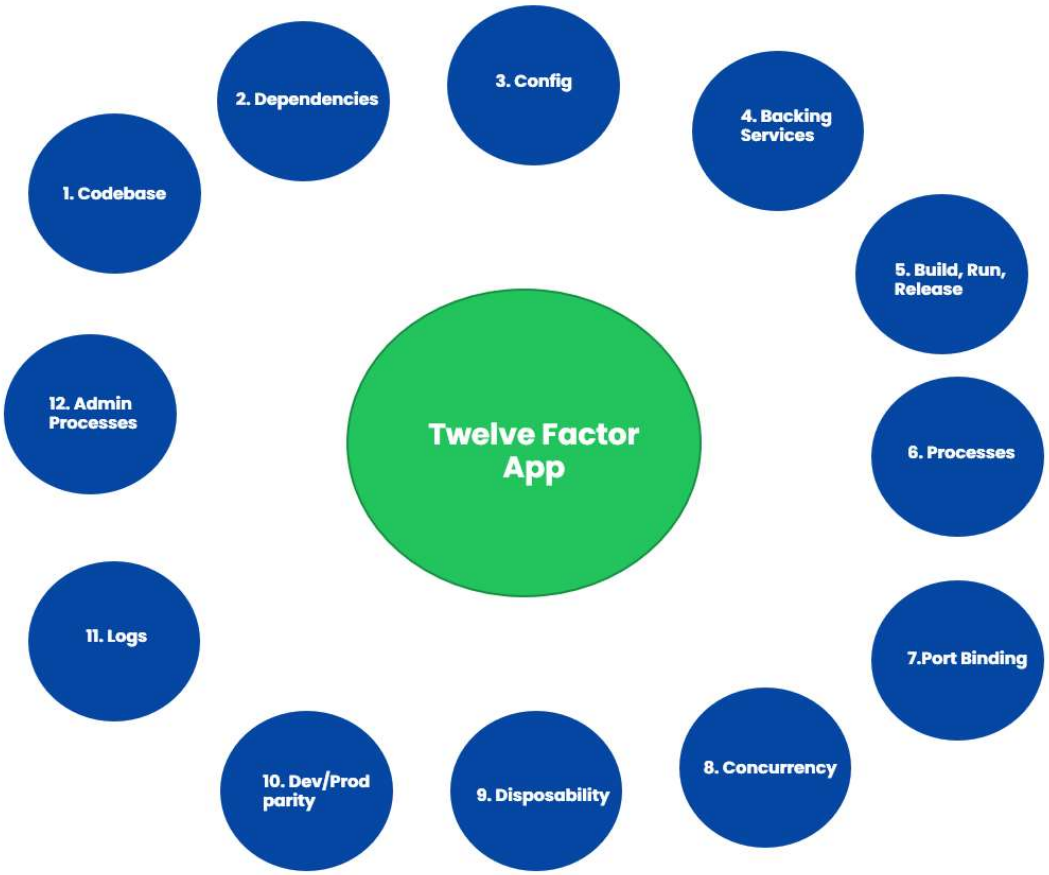
eazy bytes

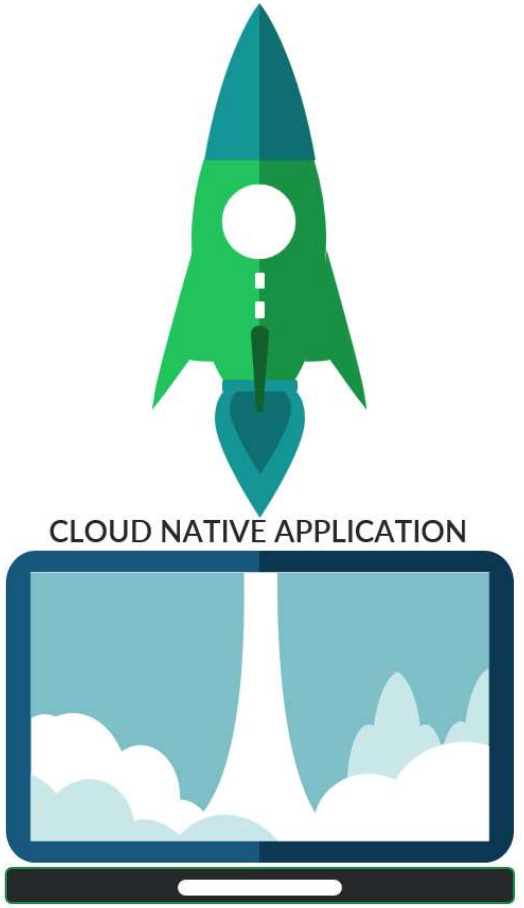


TWELVE FACTOR APP

BEST PRACTICES

eazy bytes





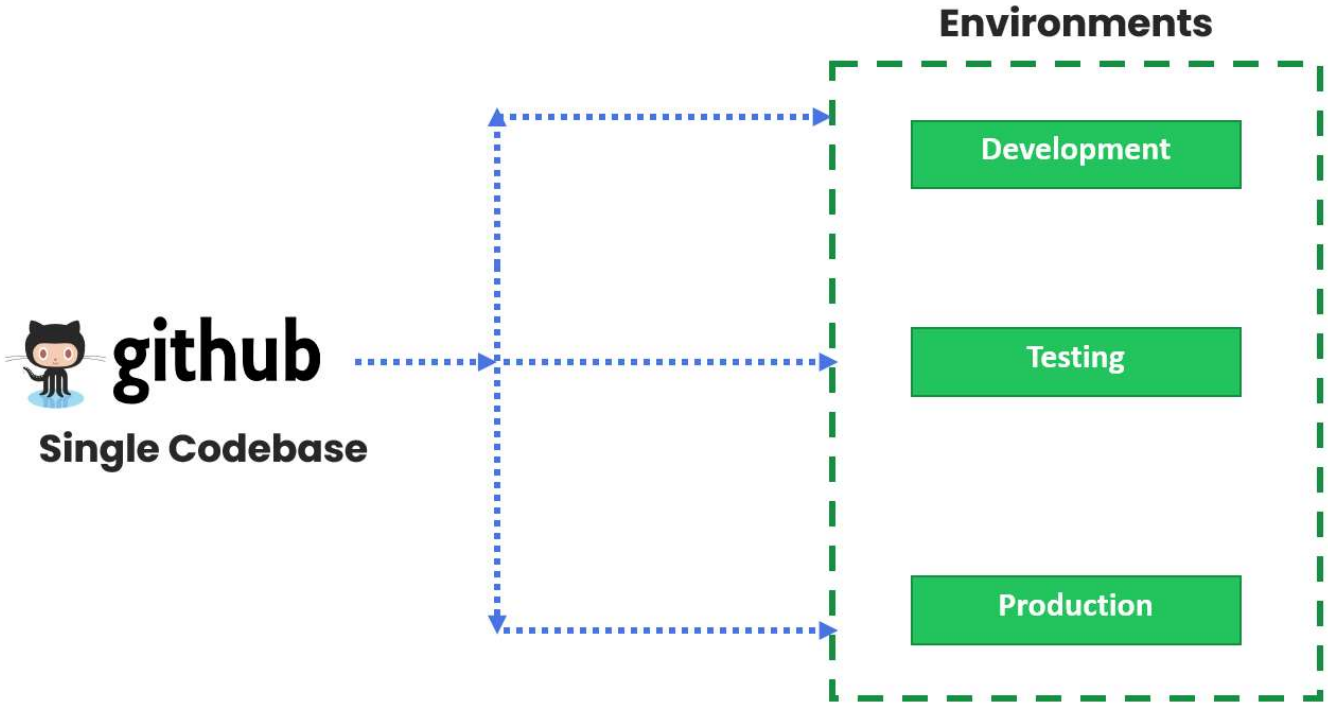
CLOUD NATIVE APPLICATION

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.



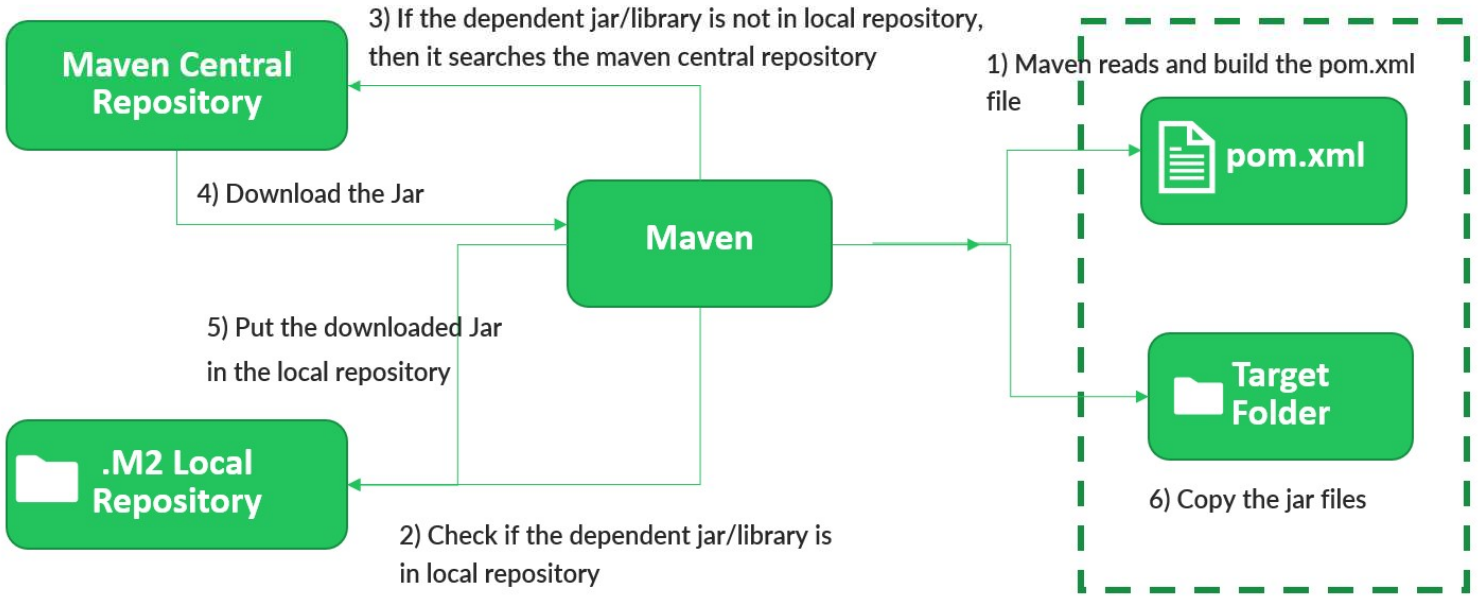
TWELVE FACTOR APP

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.

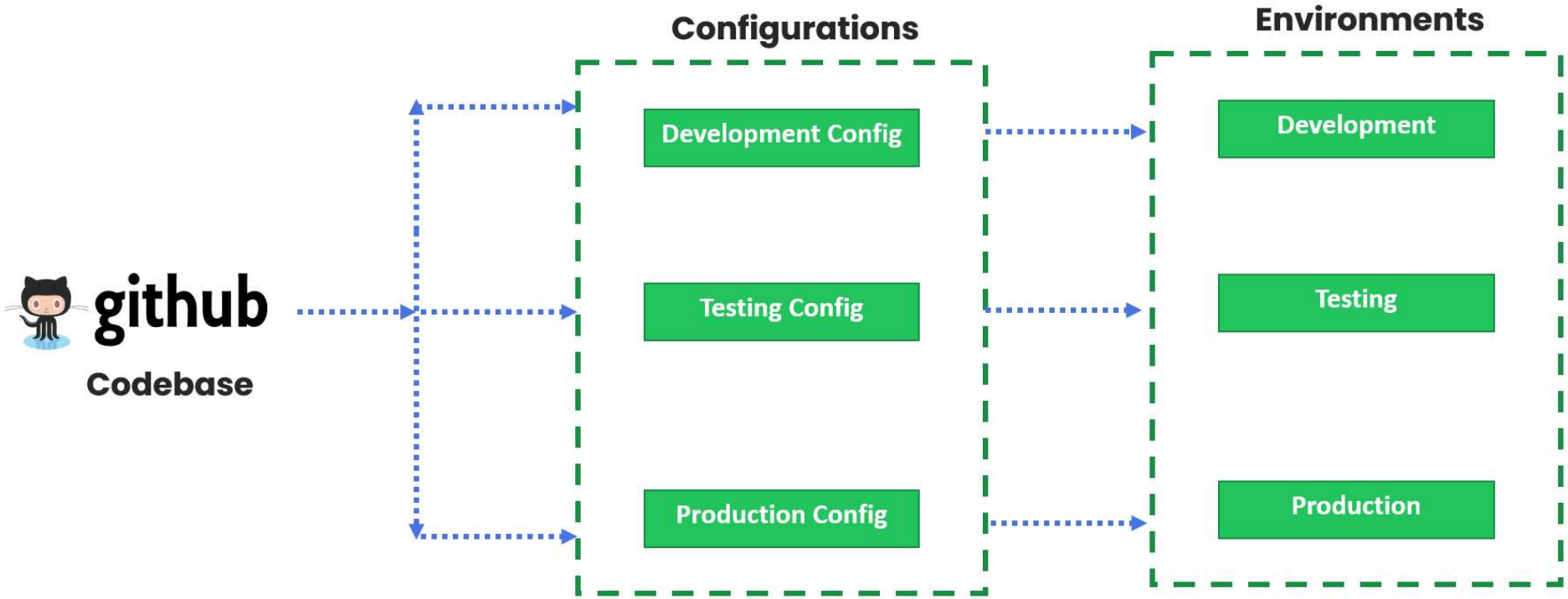


TWELVE FACTOR APP

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.



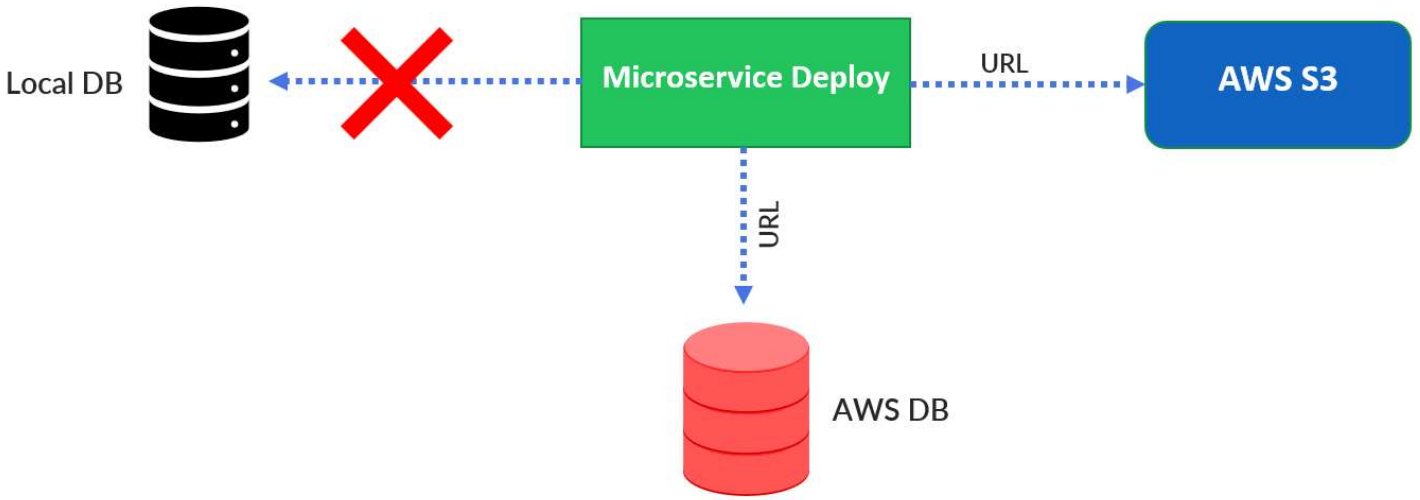
TWELVE FACTOR APP

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.

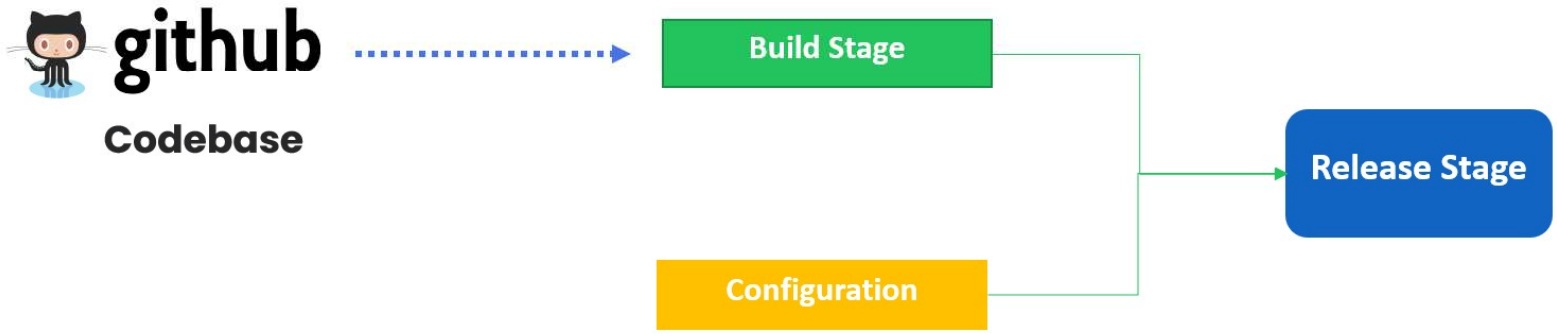


TWELVE FACTOR APP

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.

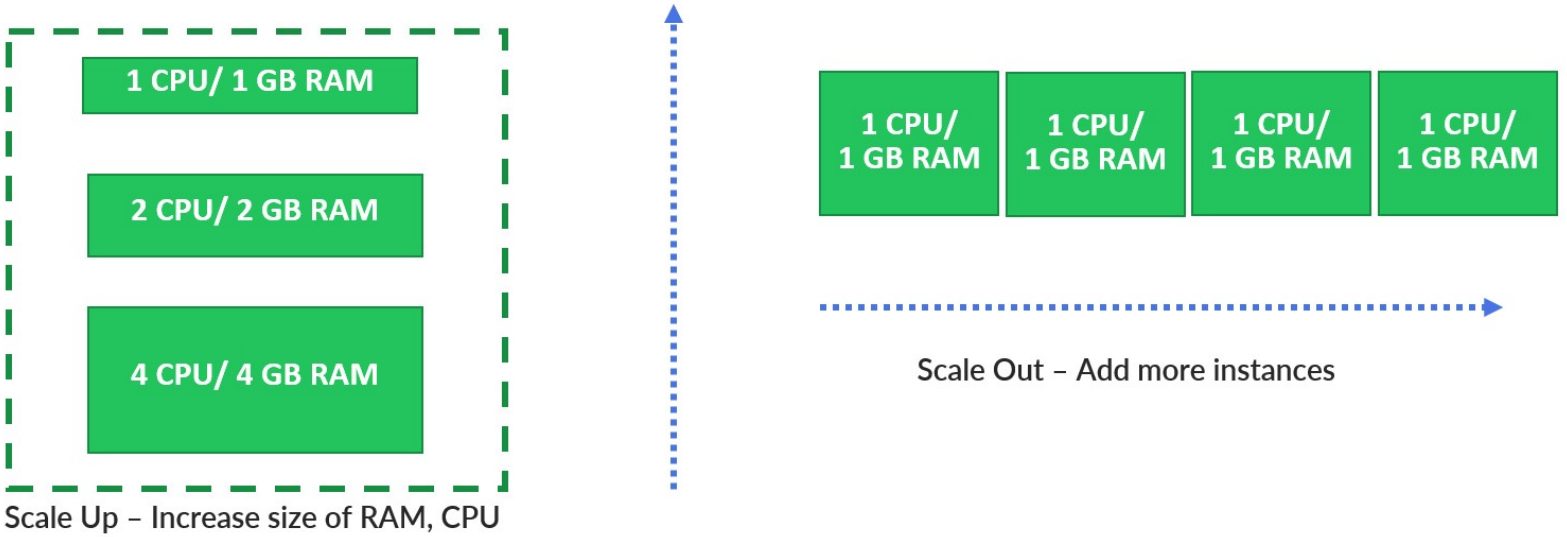
TWELVE FACTOR APP

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.

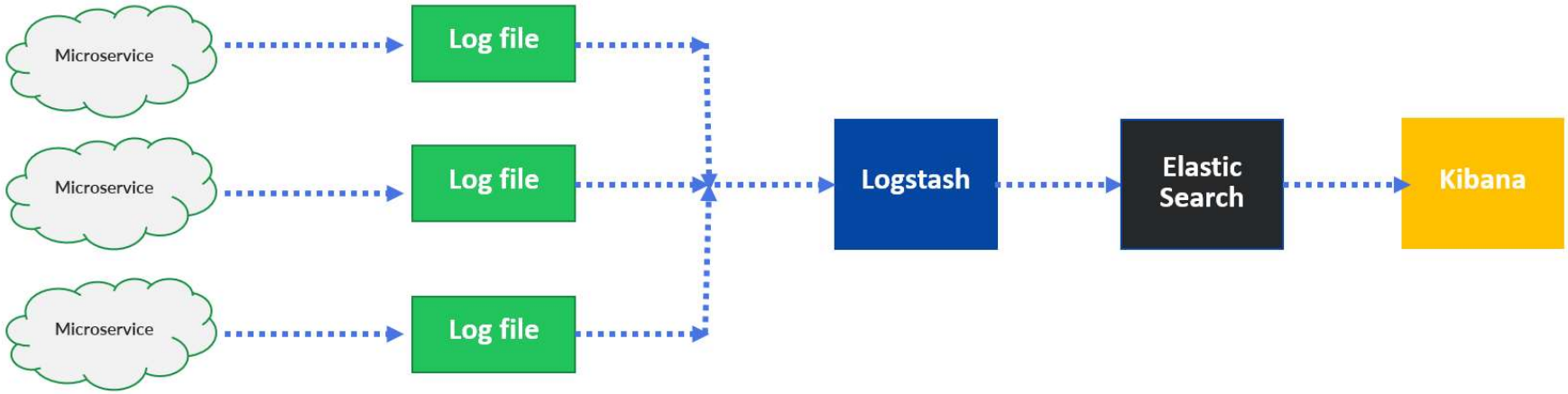
TWELVE FACTOR APP

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

eazy bytes

CONFIGURATION MANAGEMENT

SEPARATION OF CONFIGs/PROPERTIES

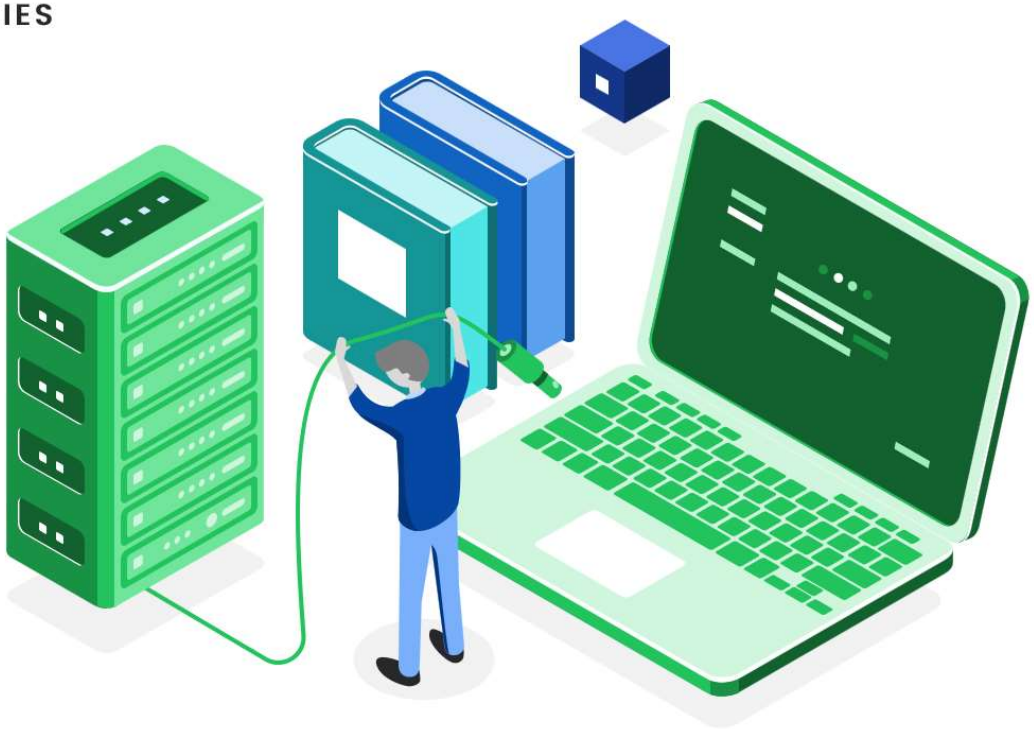
How do we separate the configurations/properties from the microservices so that same Docker image can be deployed in multiple envs.

INJECT CONFIGs/PROPERTIES

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

MAINTAIN CONFIGs/PROPERTIES

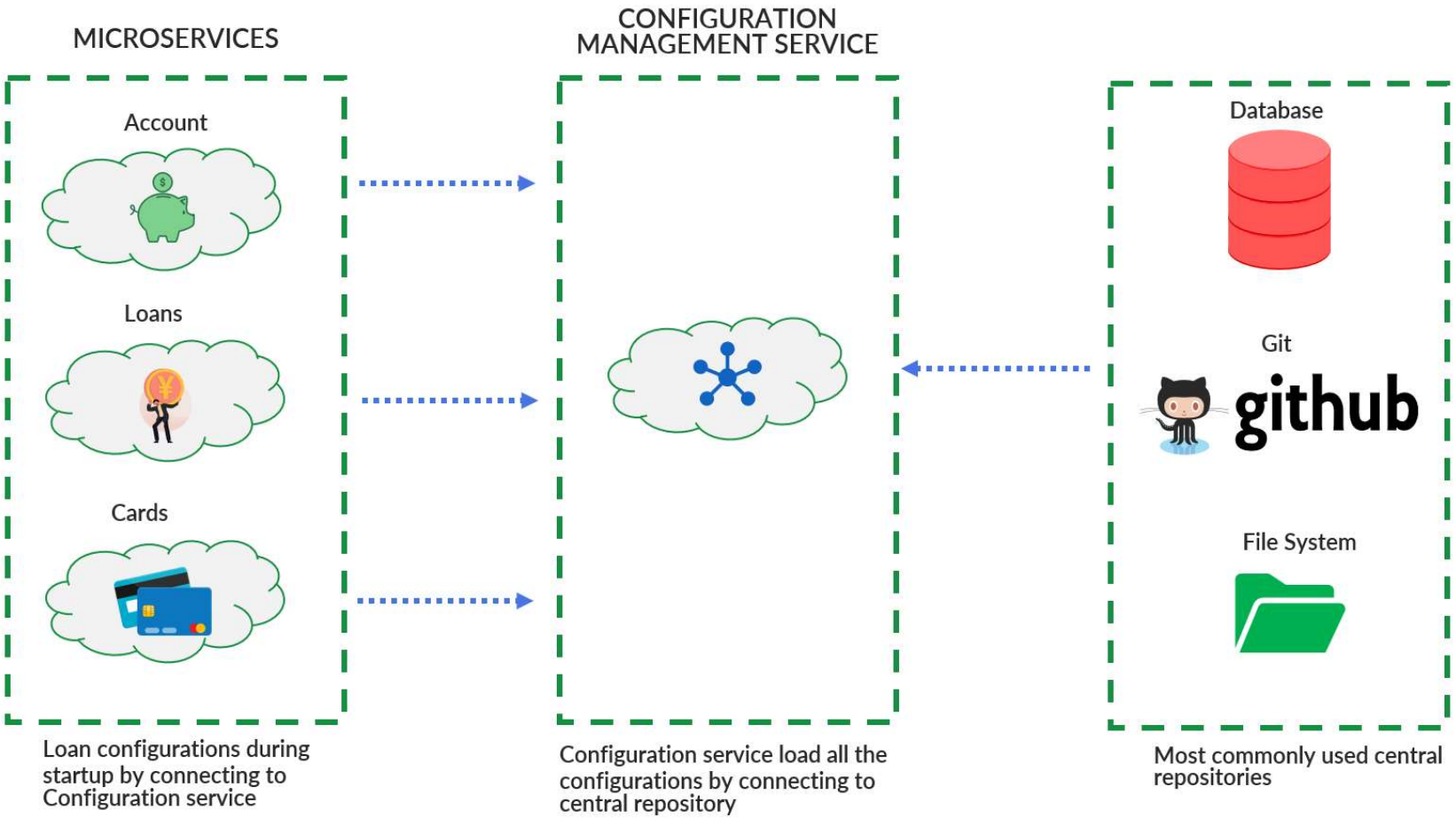
How do we maintain configurations/properties in a centralized repository along with versioning of them



CONFIGURATION MANAGEMENT

eazy bytes

ARCHITECTURE INSIDE MICROSERVICES

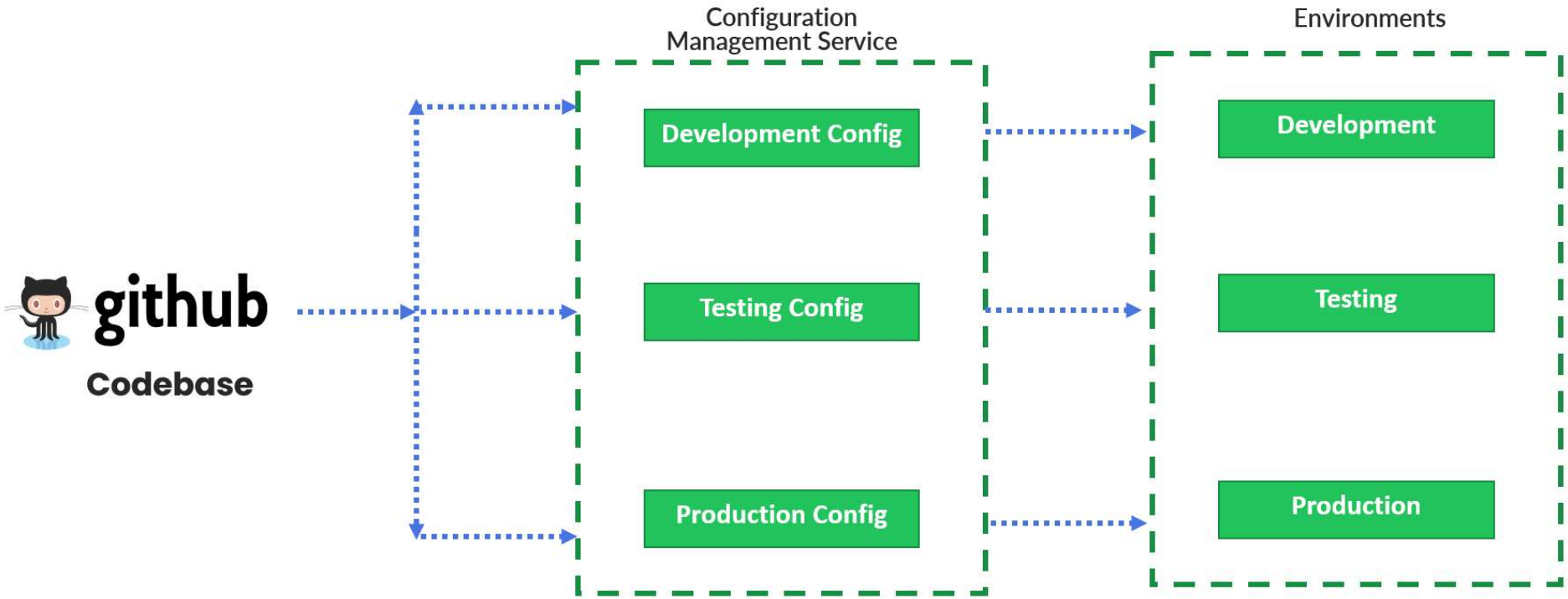


SPRING CLOUD CONFIG

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

eazy bytes

SERVICE DISCOVERY & REGISTRATION

HOW DO SERVICES LOCATE EACH OTHER INSIDE A NETWORK?

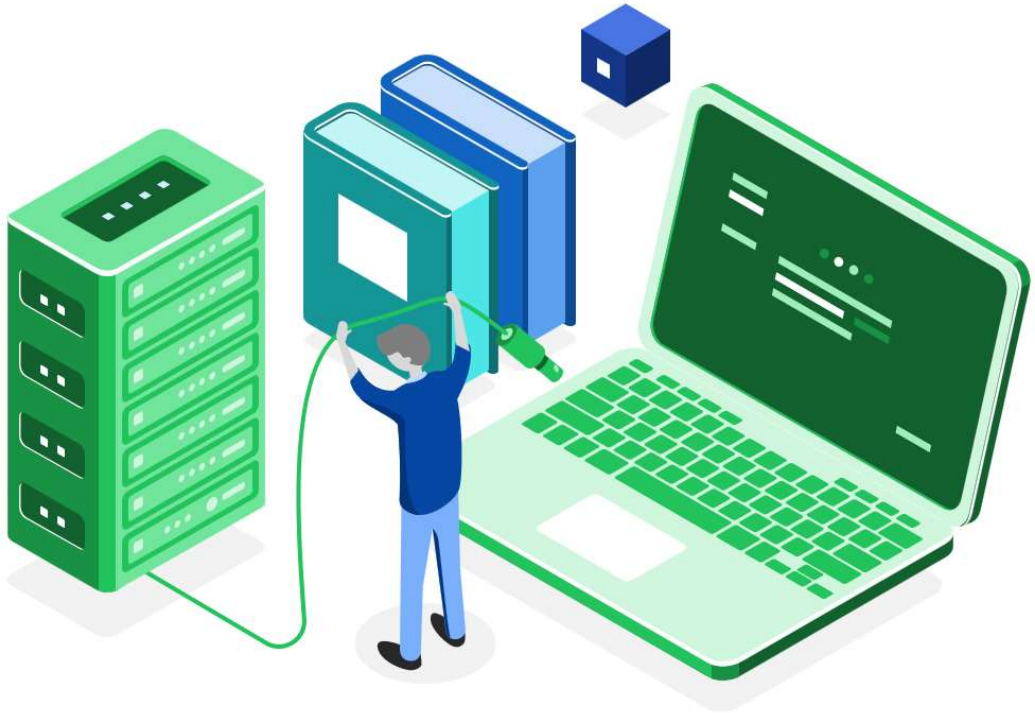
Each instance of a microservice exposes a remote API with it's own host and port. how do other microservices & clients know about these dynamic endpoint URLs to invoke them. So where is my service?

HOW DO NEW SERVICE INSTANCES ENTER INTO THE NETWORK?

If an 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 can 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 a specific service information shared across the network?



SERVICE DISCOVERY & REGISTRATION

INSIDE MICROSERVICES NETWORK

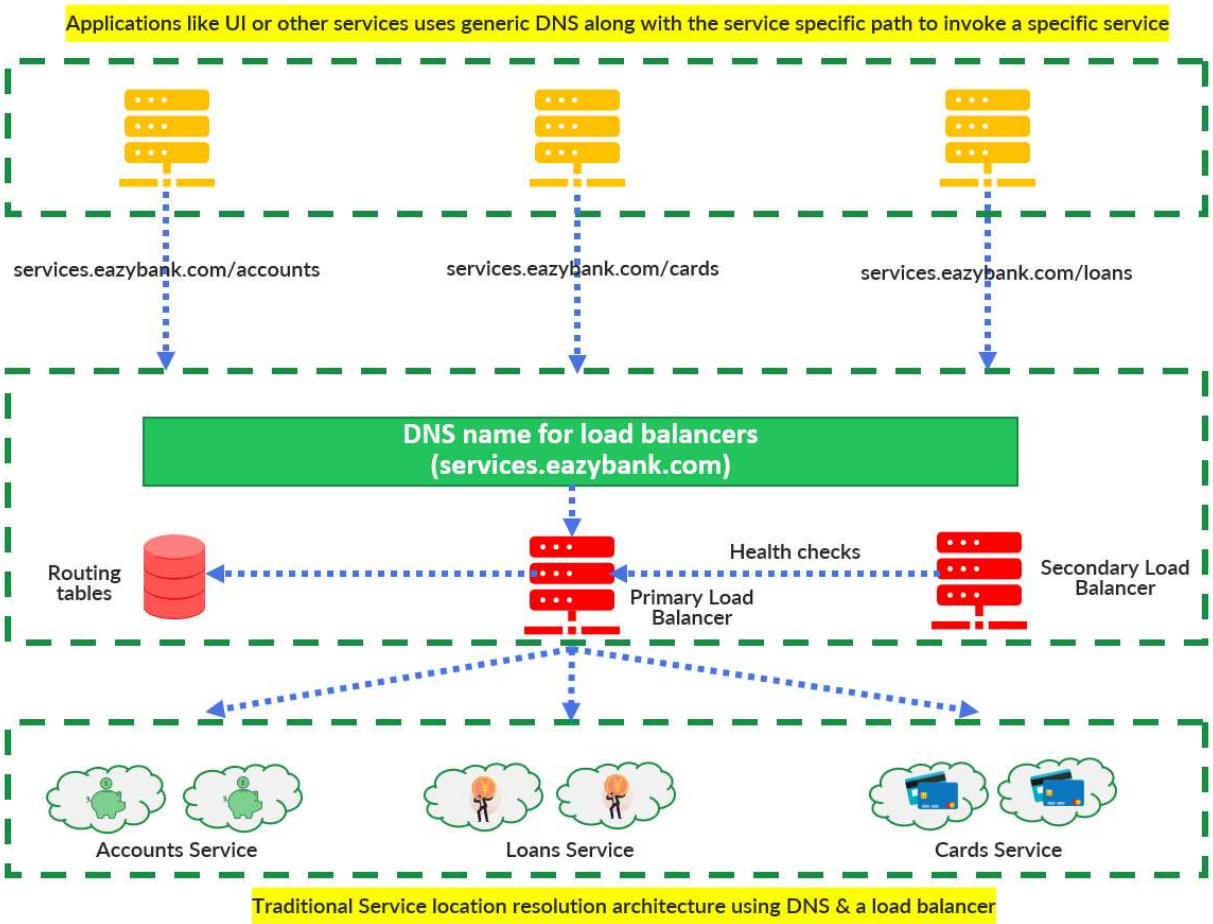
eazy bytes

- 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

eazy bytes

FOR SERVICE DISCOVERY & REGISTRATION



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.

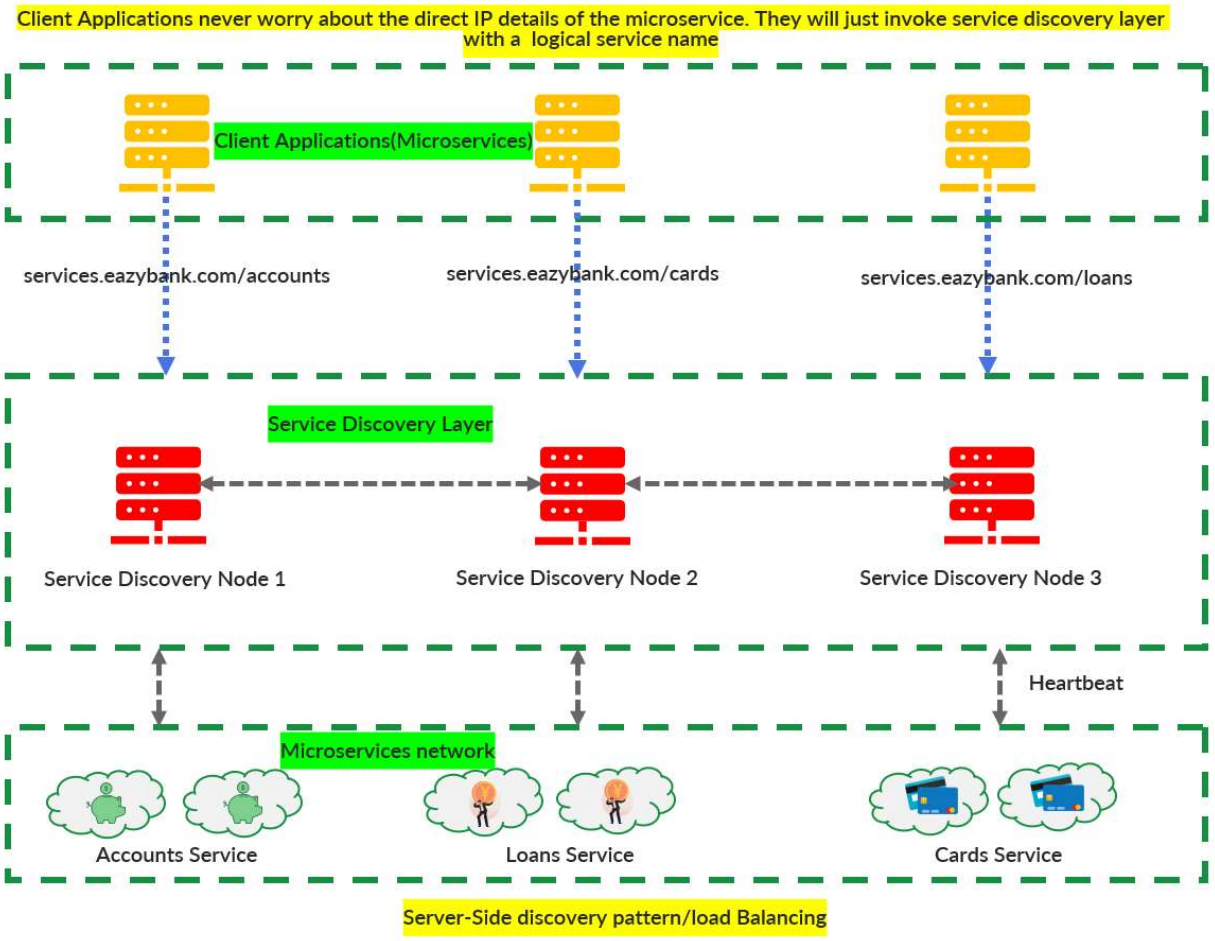
- While this type of model works well with monolithic and SOA based applications with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications for the following reasons,

    - 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



1. A service actual location can be looked up based on the given logical name

2. When a service comes online it register its IP address with a service discovery agent and let it know that it is ready to take requests

3. Service discovery nodes communicate with each other about new services, health of the services etc.

4. Service instances send a heartbeat to the service discovery agent. If a service didn't send a heartbeat, service discovery will remove the IP of the dead instance from the list

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