Challenge 2 With Microservices

Deployment

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

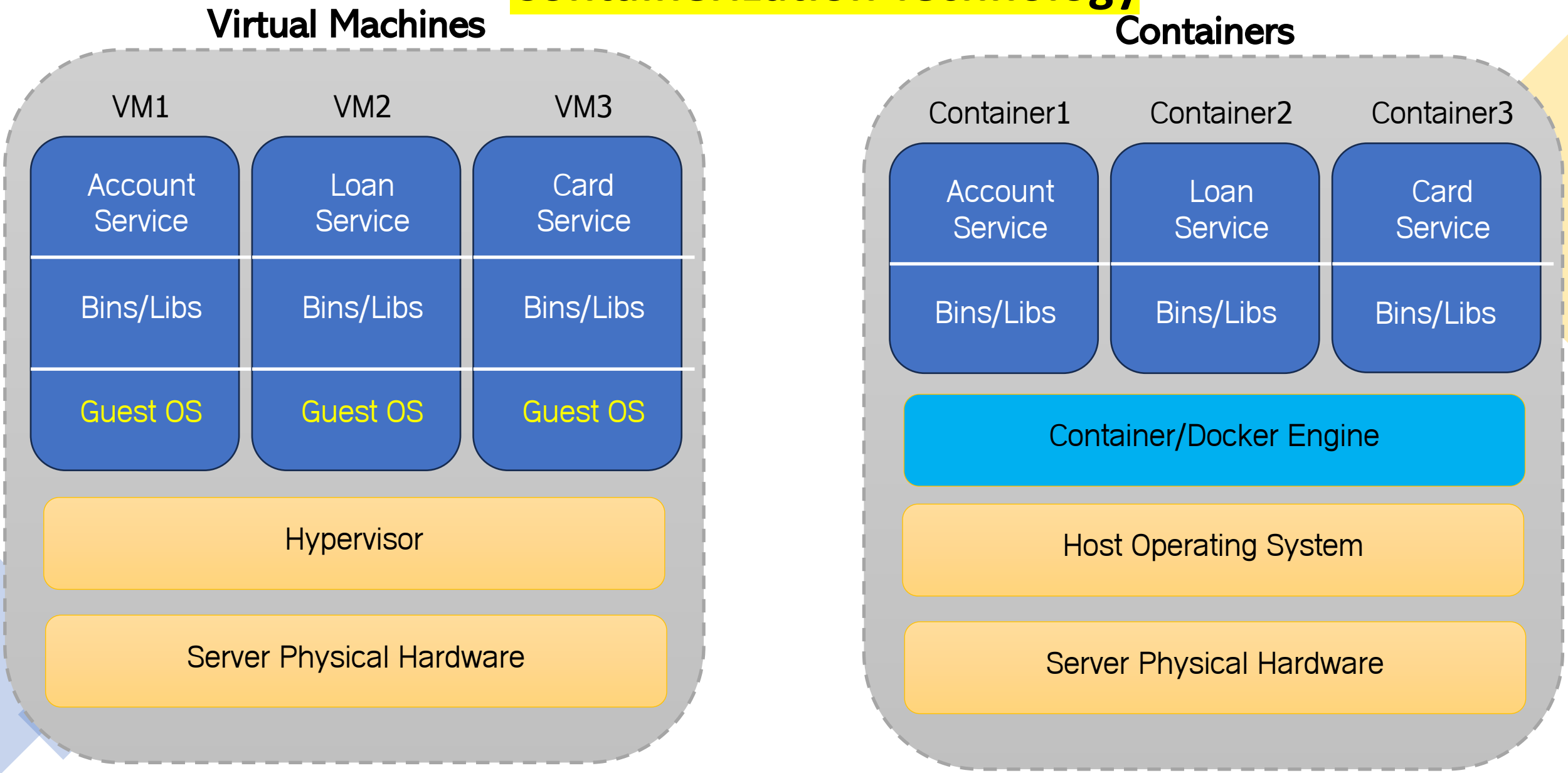
Portability

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



Containers don't need the Guest OS nor the hypervisor to assign resources; instead, they use the container engine.

What are Container and Docker?

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.

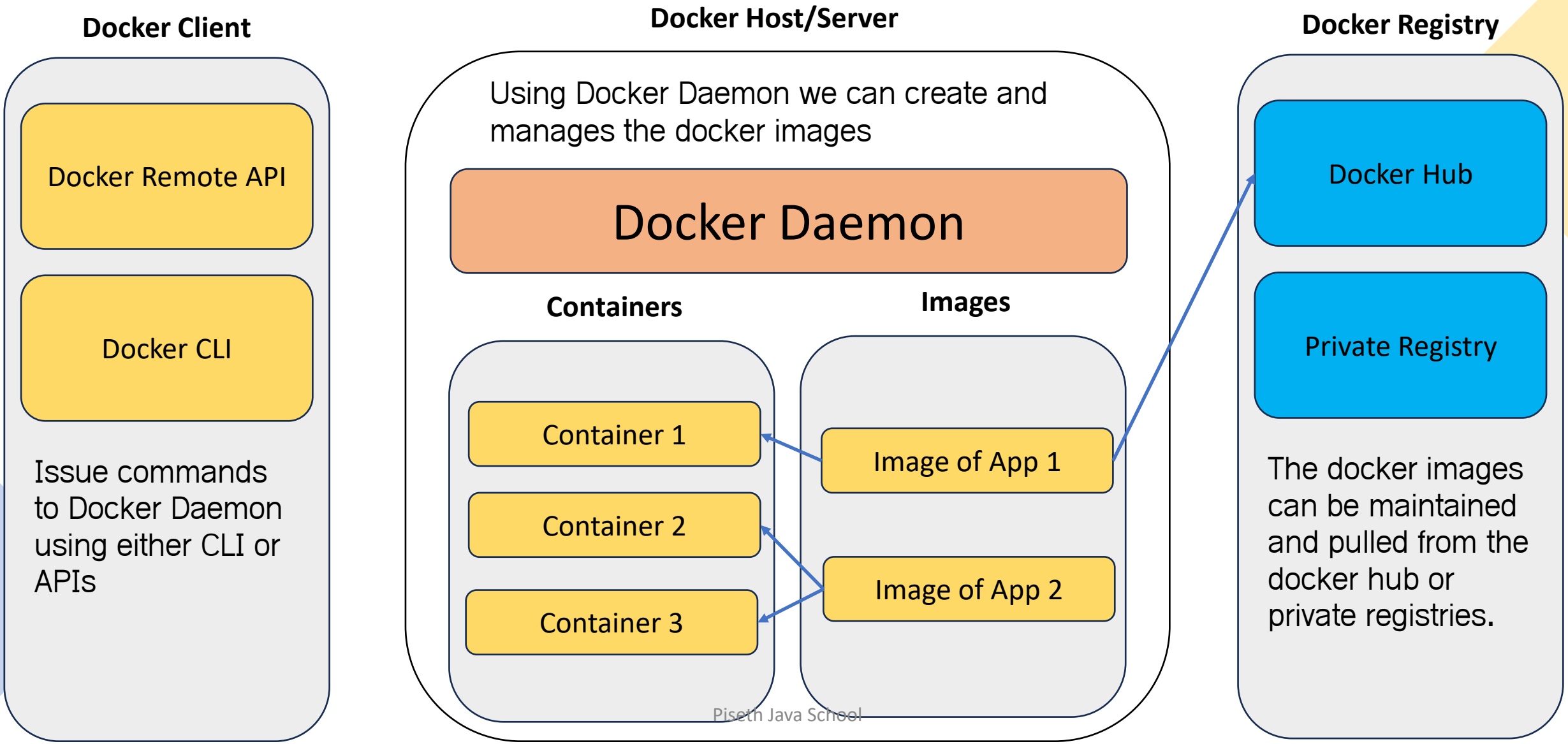
What is software containerization?

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.

Challenge 2 With Microservices



Cloud-Native Application Introduction

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.

Principles of Cloud-Native Application

1. MICROSERVICE

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

2. 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.

3. DEVOPS

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

4. 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 practices mentioned in the twelve factor app methodology (https://12factor.net/)

Difference between Cloud-Native and Traditional App

Cloud Native Application

- Predictable Behavior
- OS abstraction
- Right-sized capacity & Independent
- Continuous delivery
- Rapid recovery & Automated scalability

Traditional Enterprise Application

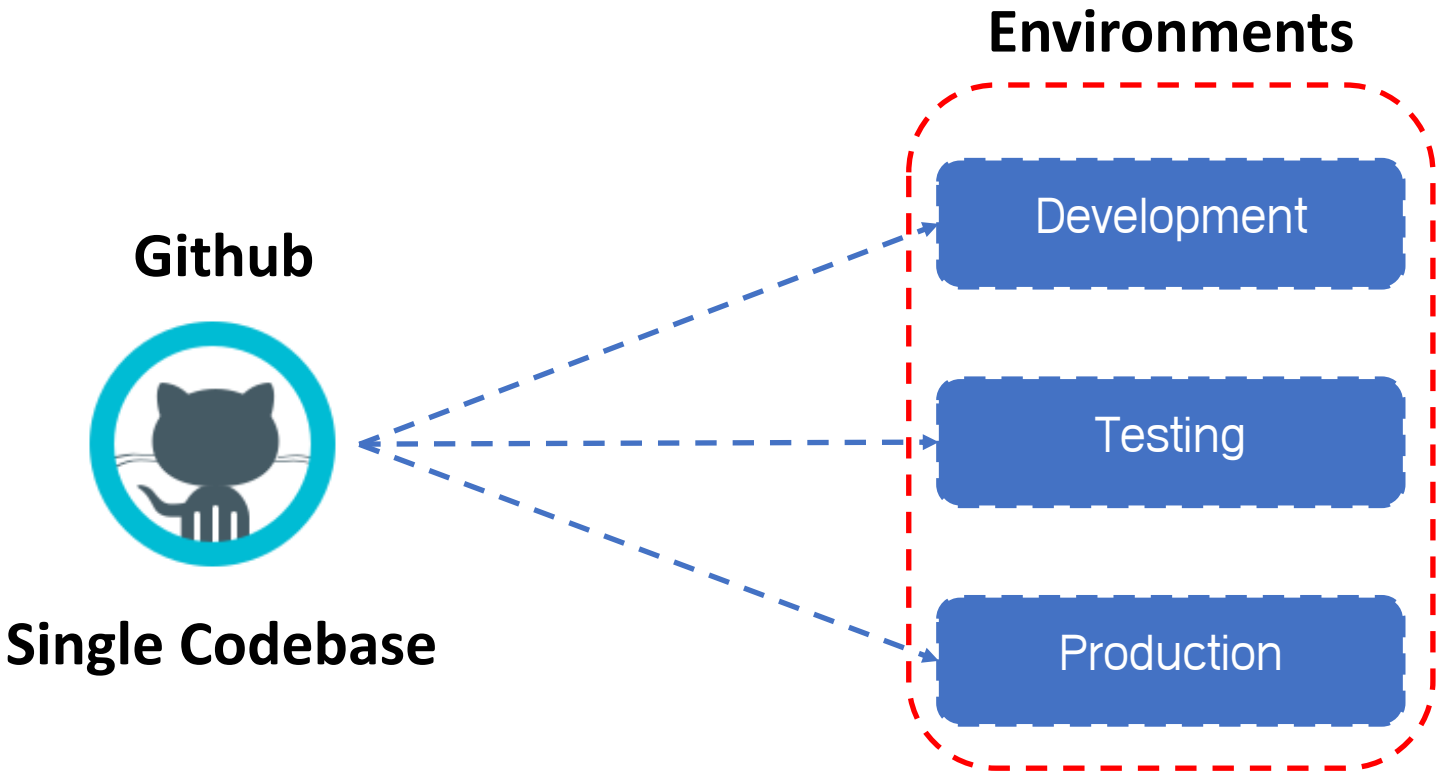
- Unpredictable Behavior
- OS dependent
- Oversized capacity & Dependent
- Waterfall development
- Slow recovery

Twelve Factors App Best Practices

1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, Run, Release
6. Processes
7. Port Binding
8. Concurrency
9. Disposability
10. Dev/Prod parity
11. Logs
12. Admin Processes

1. Codebase

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.



2. Dependencies

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.

2. Check if the dependent jar/library is in local repository

.m2 Local Repository

5. Put the downloaded Jar in the local repository

pom.xml

1. Maven reads and build the pom.xml file

Target Folder

6. Copy the jar files

Maven

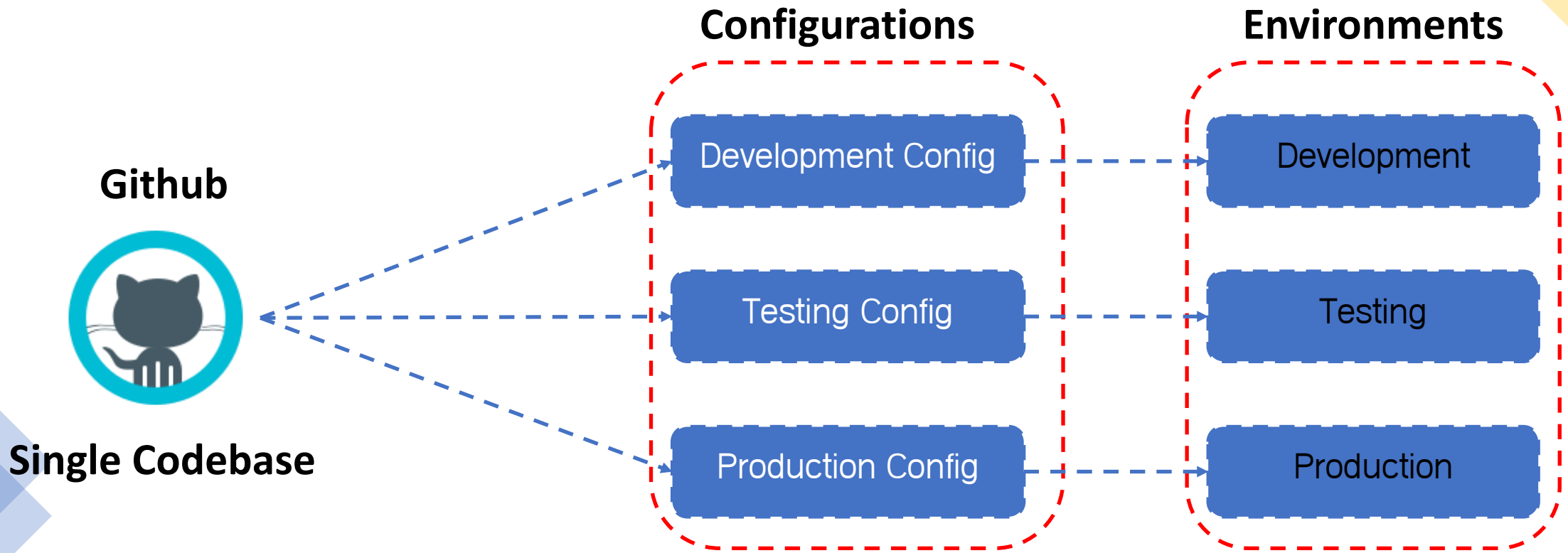
3. If the dependent jar/library is not in local repository, then it searches the maven central repository

Maven Central Repository

4. Download the Jar

3. Config

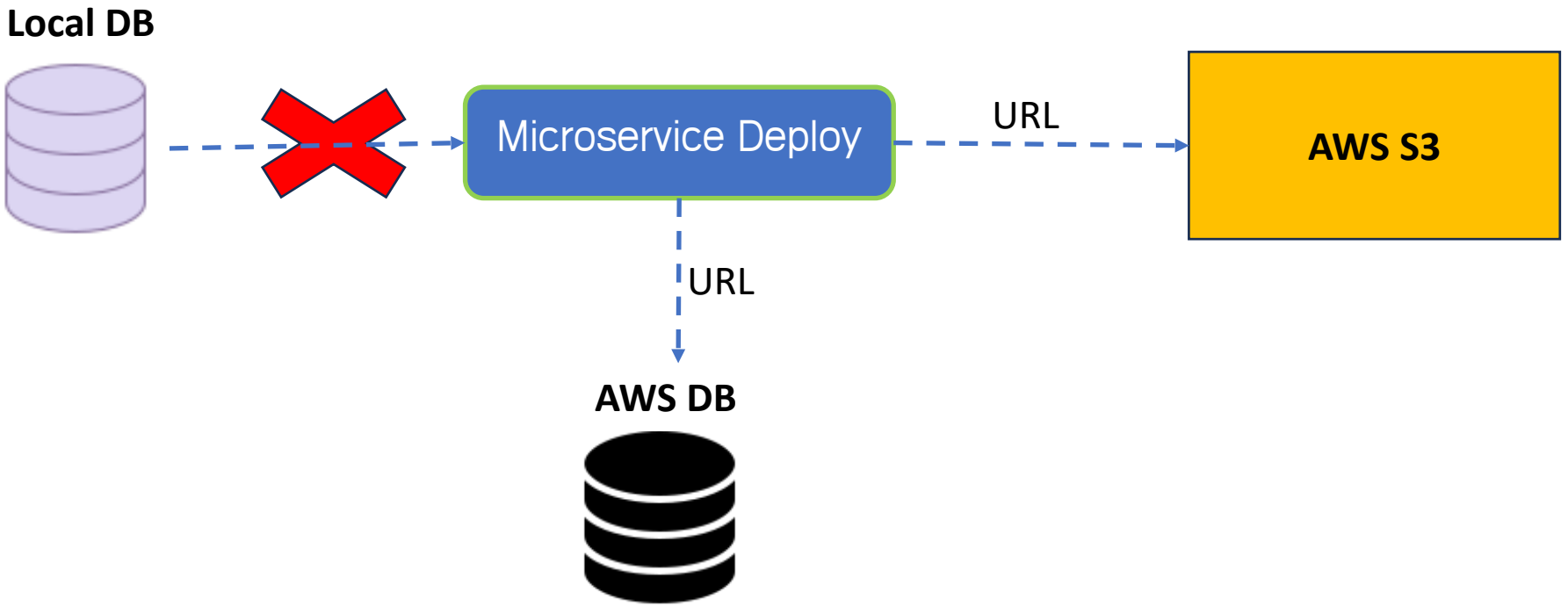
Store environment-specific configuration independently from your code. Never add embedded configurations to your source code; instead, maintain your configuration completely separated from your deployable microservice. If we keep the configuration packaged within the microservice, we'll need to redeploy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



4. Backing Service

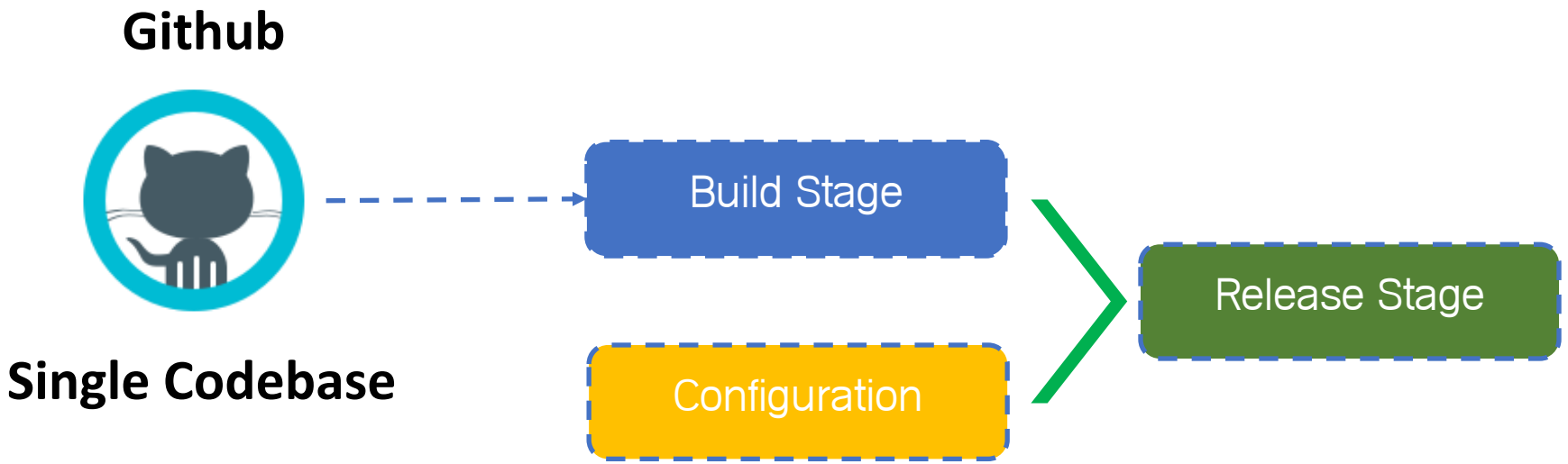
Backing Services best practice indicates that a microservices deploy should be able to swap between local connections to third party without any changes to the application code.

• 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.



5. Build , Release, Run

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.



6. Processes

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.

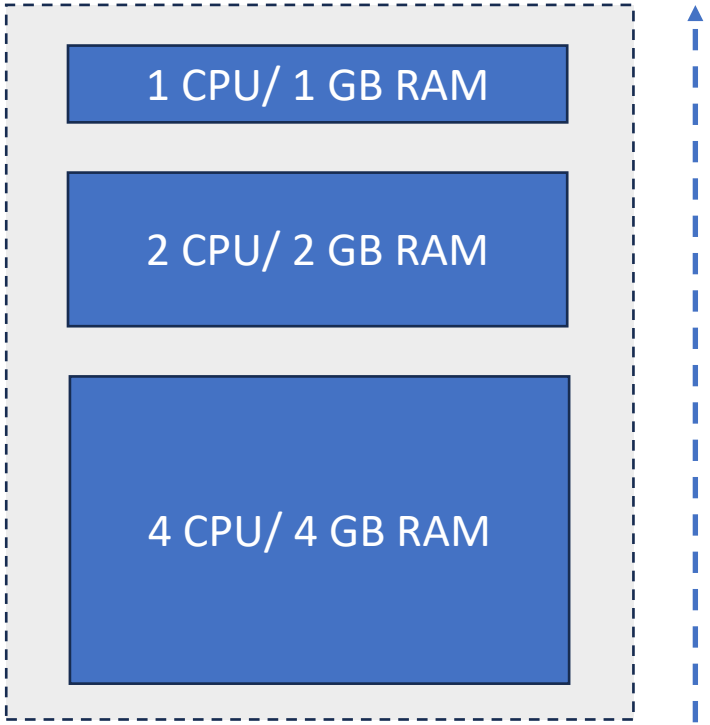
We can store the data of the loan microservice inside a SQL or NoSQL DB

7. Port Binding

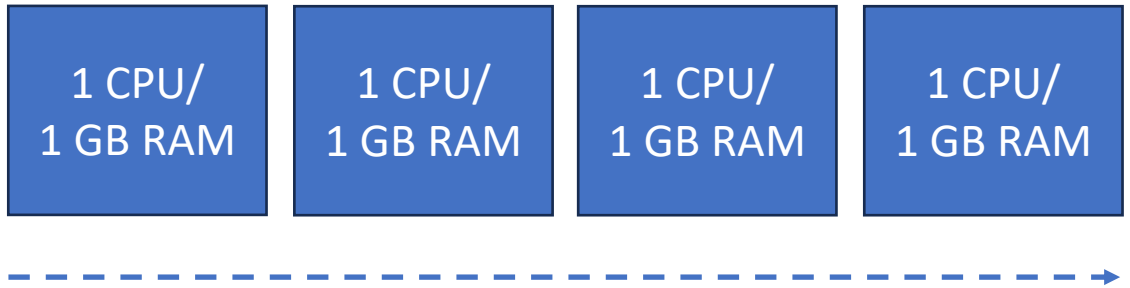
Web apps are sometimes executed inside a webserver container. For example, PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat. But each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation 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.

8. Concurrency

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.



Scale Up - Increase size of RAM, CPU



Scale Out - Add more instances

9. Disposability

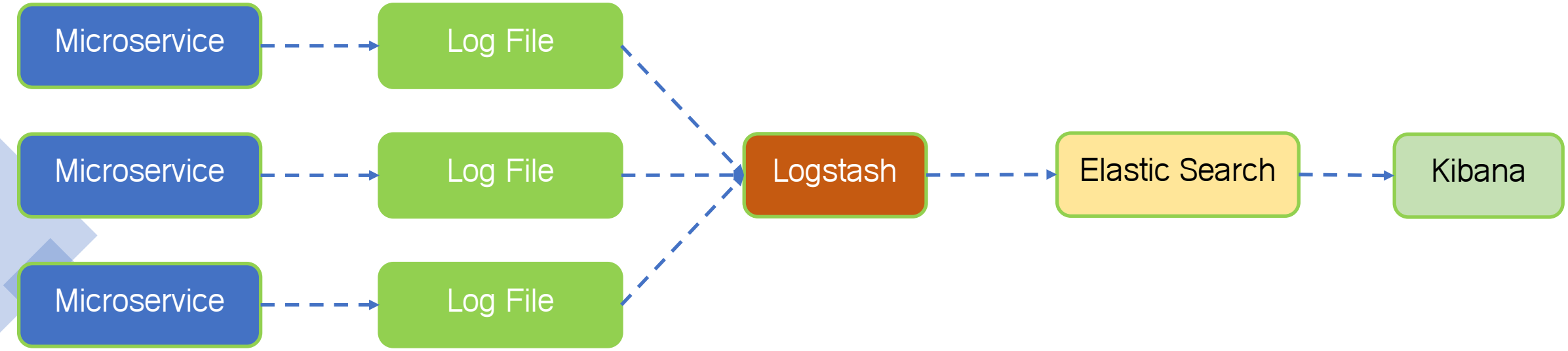
Service instances should be disposable, favoring fast startups to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement.
For example, if one of the instances of the microservice 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.

10. Dev/Prod parity

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.

11. Logs

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 standard output.



12. Admin Process

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.