Introduction

Chapter Content

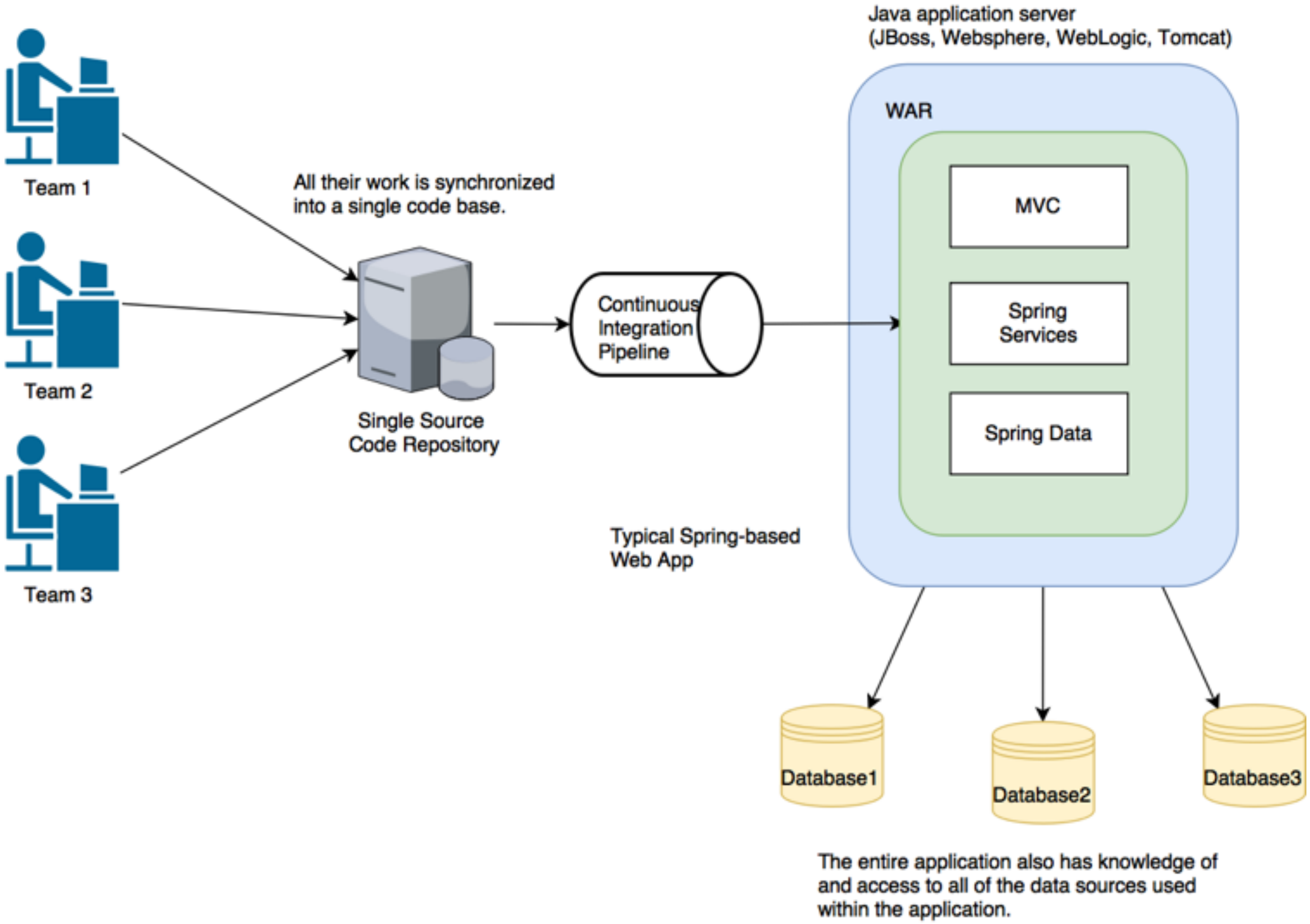
1. What is a microservice?
2. Spring Boot and Spring Cloud
3. Build a Microservice with Spring Boot
4. Benefits of Microservices
5. Cloud and Microservices
6. Various patterns for Microservices
7. Using Spring Cloud in Building your Microservices
8. Provisioning

1.What is Microservice?

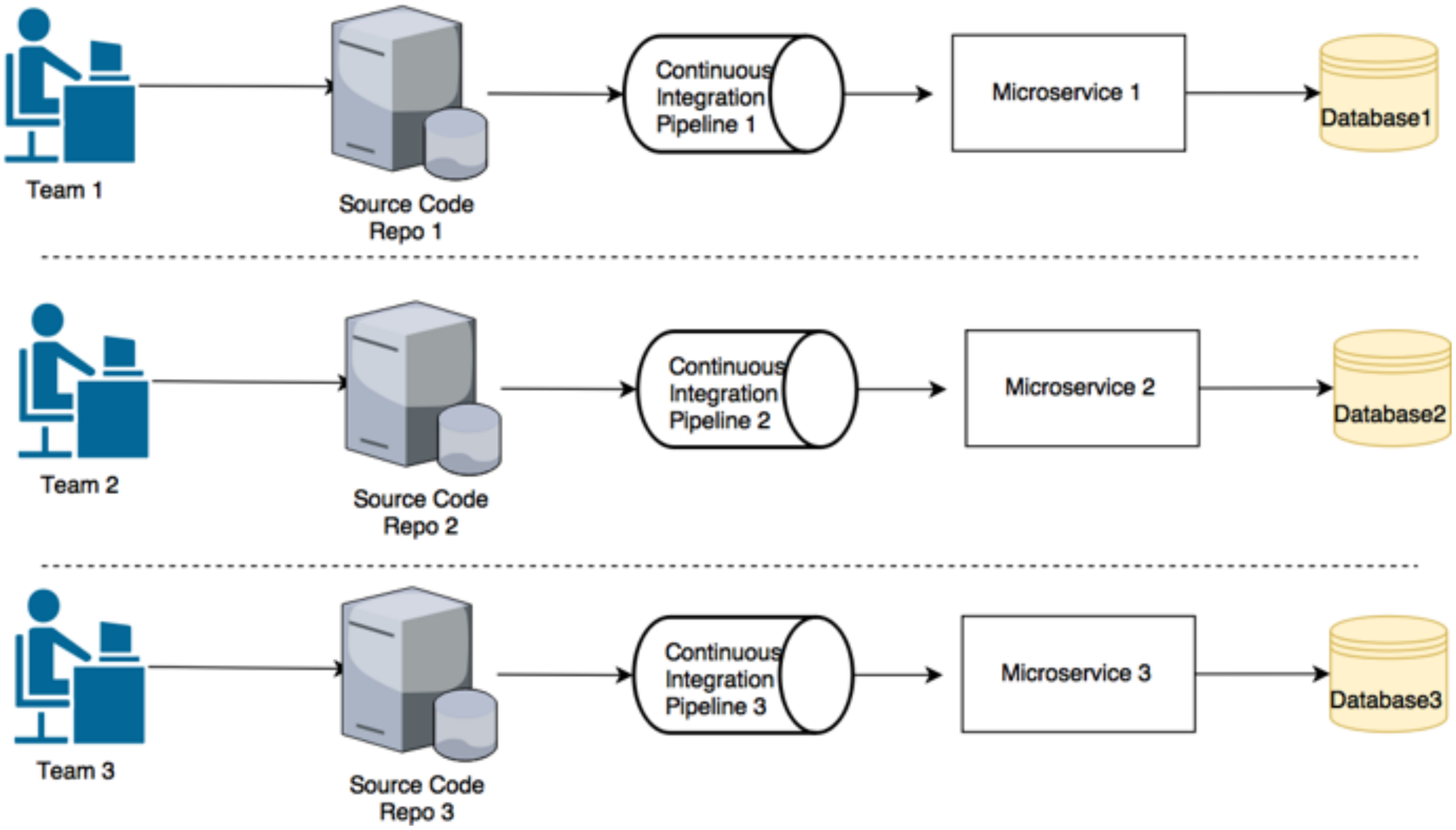
- Traditional **monolithic** architectural style
    - **Single** deployable software artifact
    - All UI, business, database access logic are **packaged** together
    - **Multiple** teams work on the application
    - Team Coordination costs didn’t **scale**
    - Every time an individual team needed to make a change, the entire app had to be **rebuilt**, **retested** and **redeployed**

- A Microservice is a **small**, **loosely** coupled, **distributed** service.

- Microservice allows you to take a large application and **decompose** it into easy-to-manage components with narrowly defined responsibilities.


Java application server
(JBoss, Websphere, WebLogic, Tomcat)
WAR
MVC
Spring Services
Spring Data
Team 1
Team 2
Team 3
All their work is synchronized into a single code base.
Single Source Code Repository
Continuous Integration Pipeline
Typical Spring-based Web App
Database1
Database2
Database3
The entire application also has knowledge of and access to all of the data sources used within the application.

Monolithic Application



Microservices

- A Microservice has the following **characteristics**:
  - Application logic is broken down into **small-grained** components with well-defined boundaries of responsibility that coordinate to deliver a solution.
  - Each component has a small domain of responsibility and is deployed completely **independently** of one another. Microservices should have responsibility for a **single part** of a business domain. Also, a microservice should be **reusable** across multiple applications.
  - Microservices communicate based on a few basic principles and employ lightweight communication protocols such as **HTTP** and **JSON** for exchanging data between the service consumer and service provider.
  - The underlying technical implementation of the service is irrelevant because the applications always communicate with a **technology-neutral** protocol (JSON is the most common).
  - Microservices allow organizations to have small development teams with well-defined areas of responsibility. These teams might work toward a single goal such as delivering an application, but each team is responsible only for the services on which they’re working.

2. Spring Boot and Spring Cloud

- Spring has become the **de facto** development framework for building **Java-based** applications.

- Spring development community adapts to the movement from monolithic to distributed models and they launch two projects: **Spring Boot** and **Spring Cloud**

- Spring Boot strips away many of the “enterprise” features found in Spring and instead delivers a framework geared toward Java-based, **REST-based** Microservices.

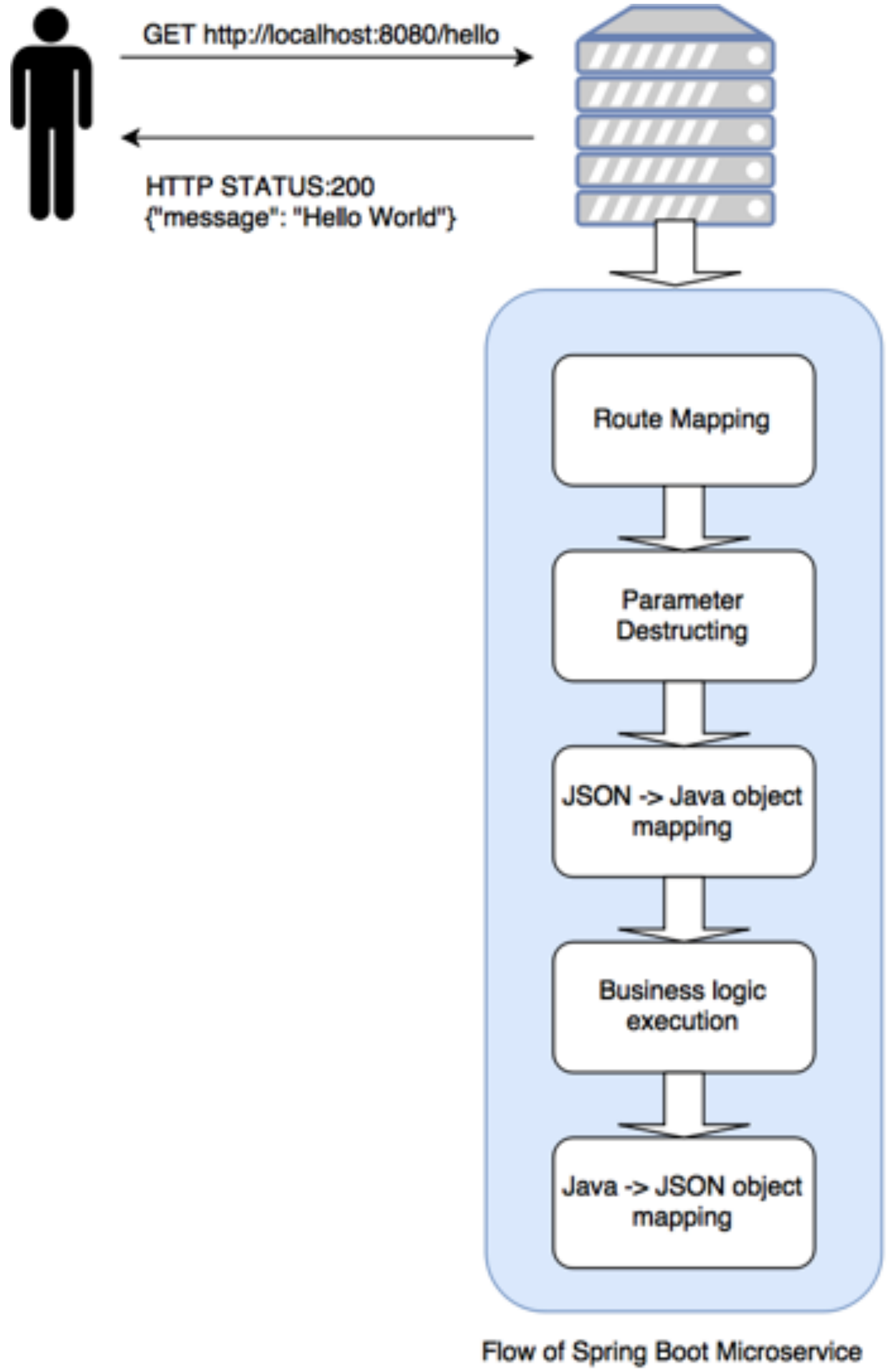
- With a few simple annotations, a Java developer can quickly build a REST microservice that can be packaged and deployed without the need for an external application container.

- As Microservices have become one of the more common architectural patterns for building cloud-based applications, the Spring Framework community provides us Spring Cloud

- Spring Cloud **wraps** several popular cloud-management microservice frameworks under a common framework and makes them easily available

3. Build A Microservice with Spring Boot

- A “Hello World” quick example to show the process of creating Spring Boot microservice.


GET http://localhost:8080/hello
HTTP STATUS:200
{"message": "Hello World"}
Route Mapping
Parameter Destructing
JSON -> Java object mapping
Business logic execution
Java -> JSON object mapping
Flow of Spring Boot Microservice

Spring Boot Flow

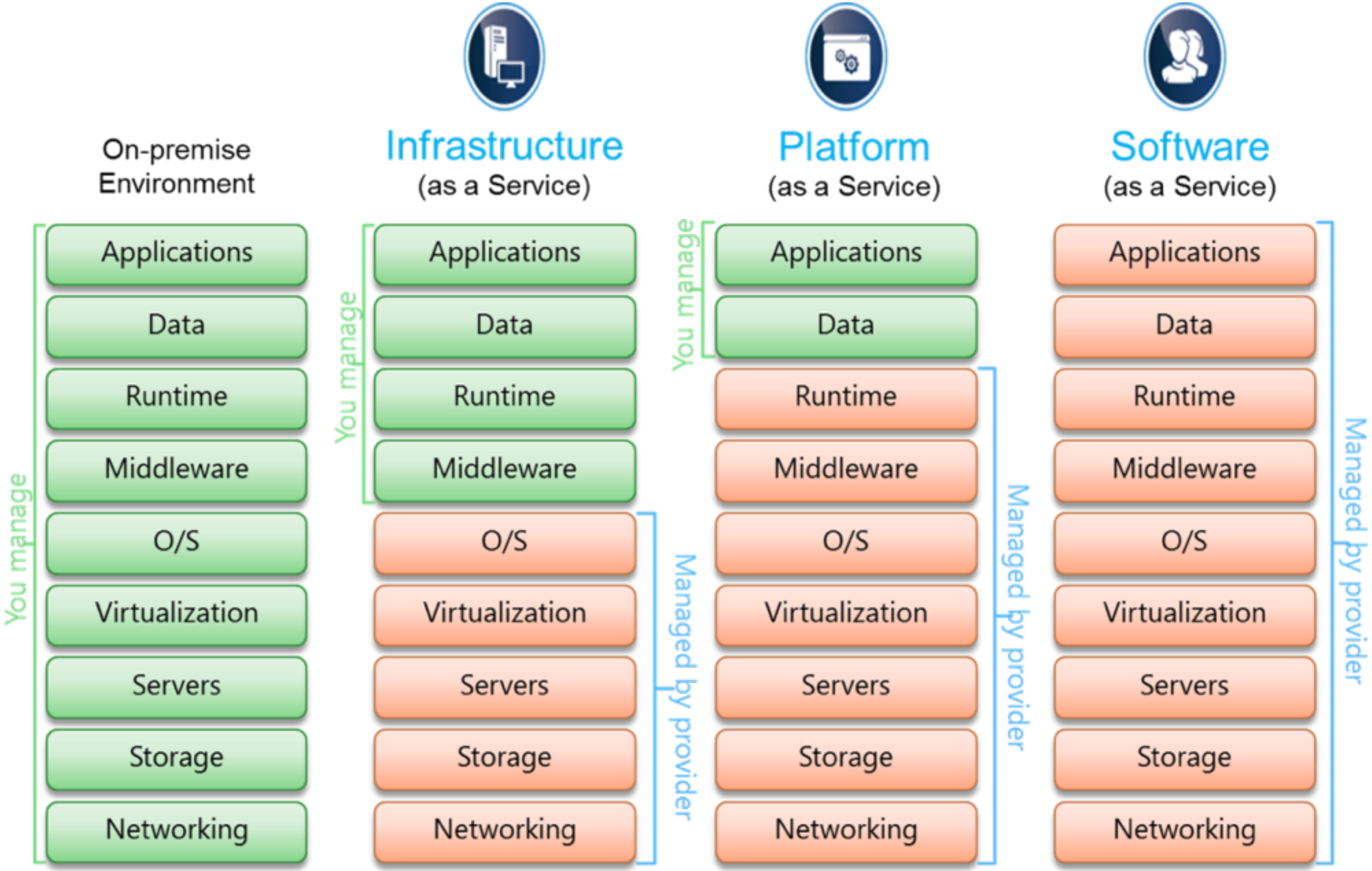
4. Benefits of Microservices

- The need for change from traditional monolithic approach:
  - **Complexity** increases. Today's applications need to talk to multiple services and databases.
  - Customer wants **faster** delivery.
  - **Performance** and **scalability**. Applications need to scale up and down quickly to accommodate the transaction volume
  - Customer expects their application to be available. Application **failure** is becoming less tolerable.

- For the concept of Microservice, we can unbundle the application into small services:
    - **Flexible**. Decoupled services can be composed and rearranged to quickly deliver new functionality.
    - **Resilient**. Failures can be localized to a small part of the application and contained before the entire application experiences an outage.
    - **Scalable**. Decoupled services can easily be distributed horizontally across multi- ple servers, making it possible to scale the features/services appropriately.

5. Cloud and Microservices

- Three basic models exist in cloud-based computing:
    - Infrastructure as a Service (IaaS)
    - Platform as a Service (Paas)
    - Software as a Service (Saas)

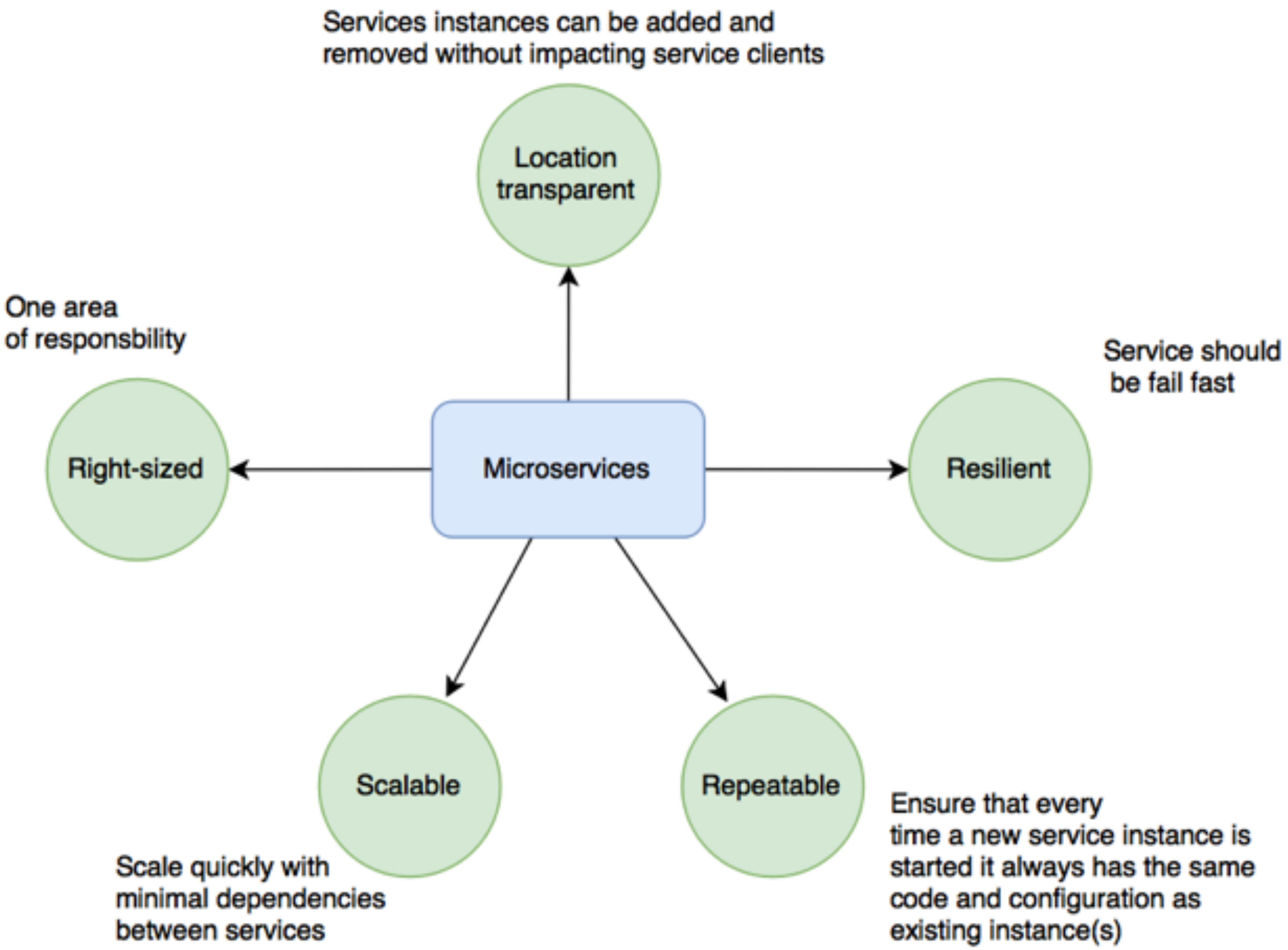


On-premise Environment	Infrastructure (as a Service)	Platform (as a Service)	Software (as a Service)
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking

- As developer is writing a microservice, it will sooner or later go to:
  - **Physical server** - Usually difficult to scale and less adopted
  - **Virtual machine image** - A microservice can be packaged up in a virtual machine image and multiple instances of the service can then be quickly deployed and started in either a IaaS private or public cloud.
  - **Virtual container** - Rather than deploying a service to a full virtual machine, many developers deploy their services as Docker containers (or equivalent container technology) to the cloud.
- The advantage of cloud-based microservices centers around the concept of **elasticity**

6. Various Patterns for Microservices

- The concept of Microservice is easy to understand, but **NOT** easy to implement



Services instances can be added and
removed without impacting service clients

Location transparent

One area
of responsbility

Right-sized

Microservices

Resilient

Service should
be fail fast

Scalable

Scale quickly with
minimal dependencies
between services

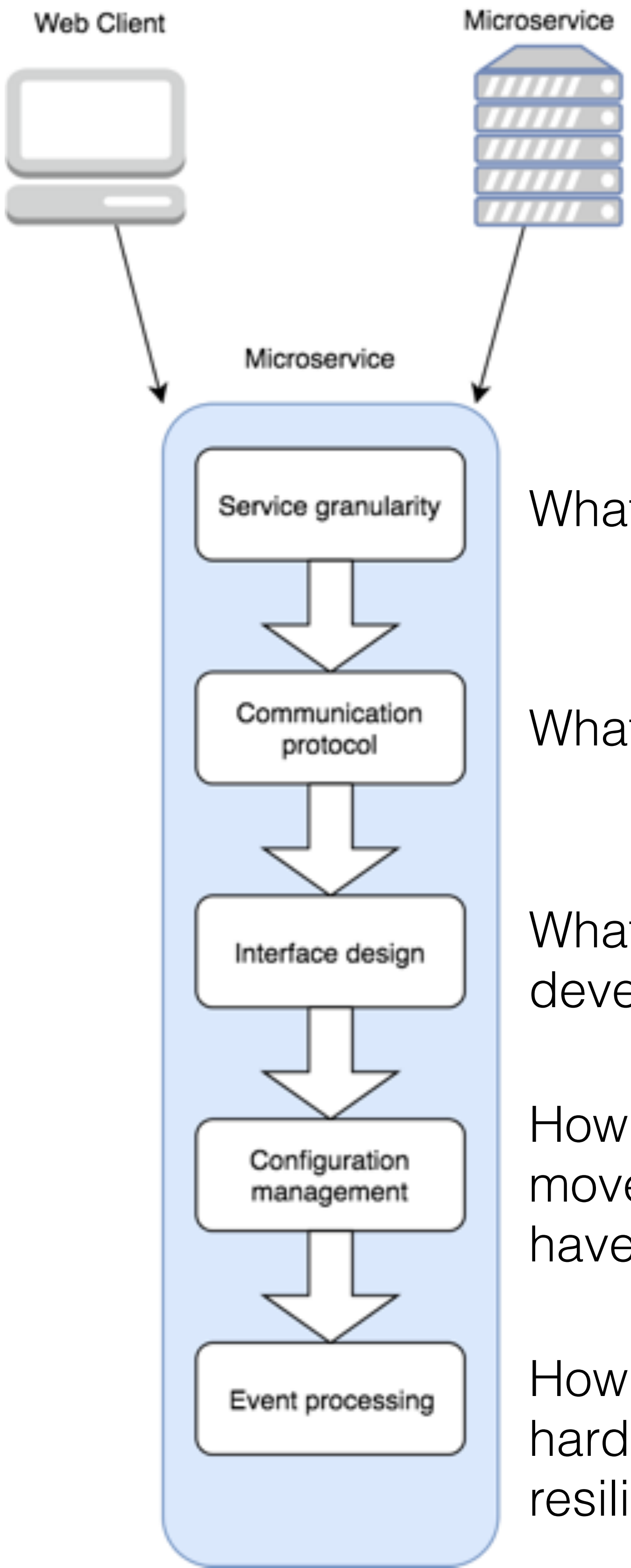
Repeatable

Ensure that every
time a new service instance is
started it always has the same
code and configuration as
existing instance(s)

Challenges in Microservice Architecture

- While we choose Spring Boot and Spring Cloud to implement the patterns. There are other availabilities as well.
- Six patterns:
  - Core development patterns
  - Routing patterns
  - Client resiliency patterns
  - Security patterns
  - Logging and tracing patterns
  - Build and deployment patterns

6.1 Core microservice development pattern



What is the right level of responsibility the service should have?

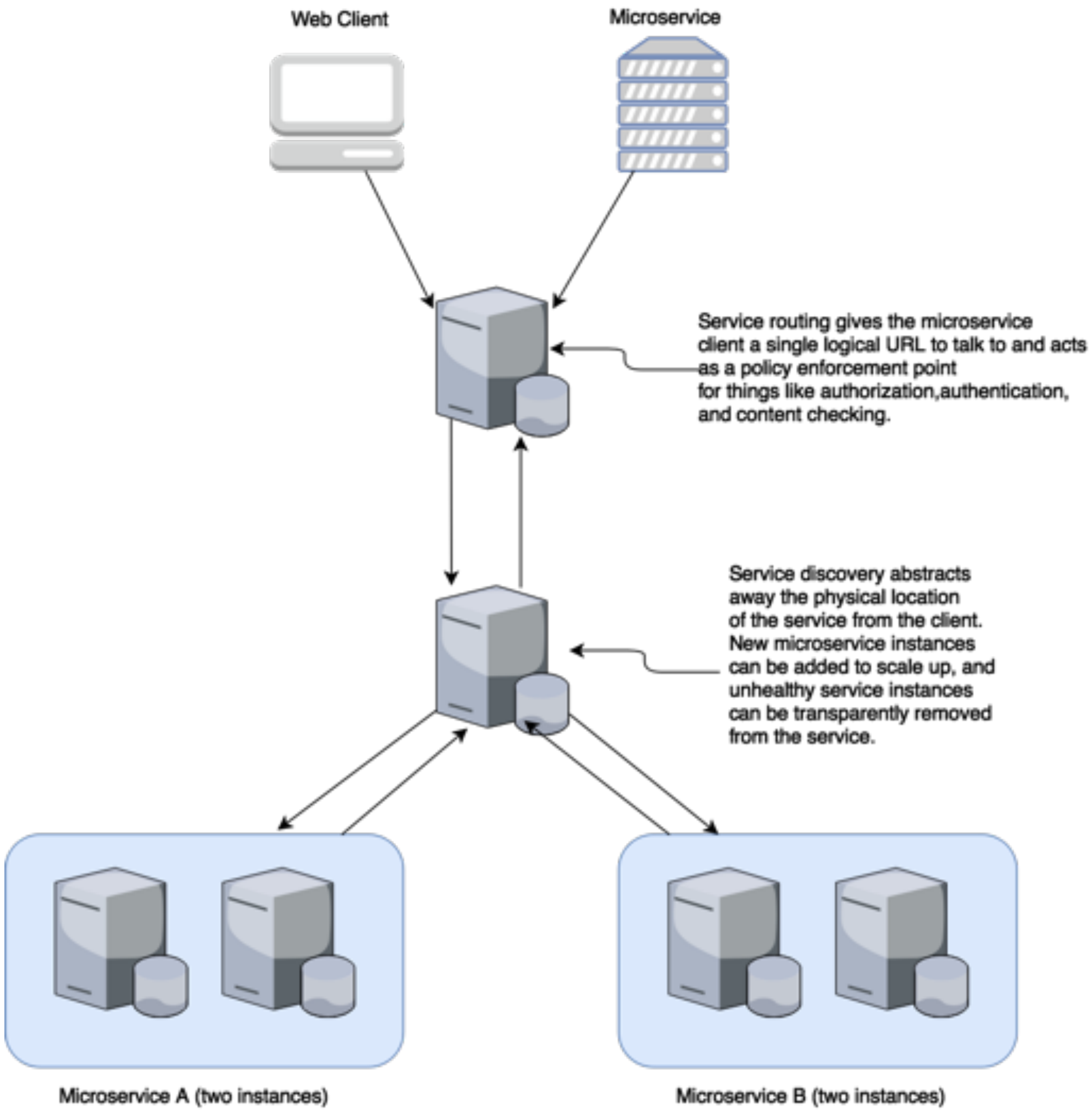
What is the communication protocol?

What is the best way to design the service interface that developers are going to use?

How to manage the configuration of your microservice so that as it moves between different environments in the cloud you never have to change the core application code or configuration?

How to decouple your microservice using events to minimize hardcoded dependencies between services and increase the resiliency of the application?

6.2 Routing Patterns


Web Client
Microservice
Microservice A (two instances)
Microservice B (two instances)

Service routing gives the microservice client a single logical URL to talk to and acts as a policy enforcement point for things like authorization,authentication, and content checking.

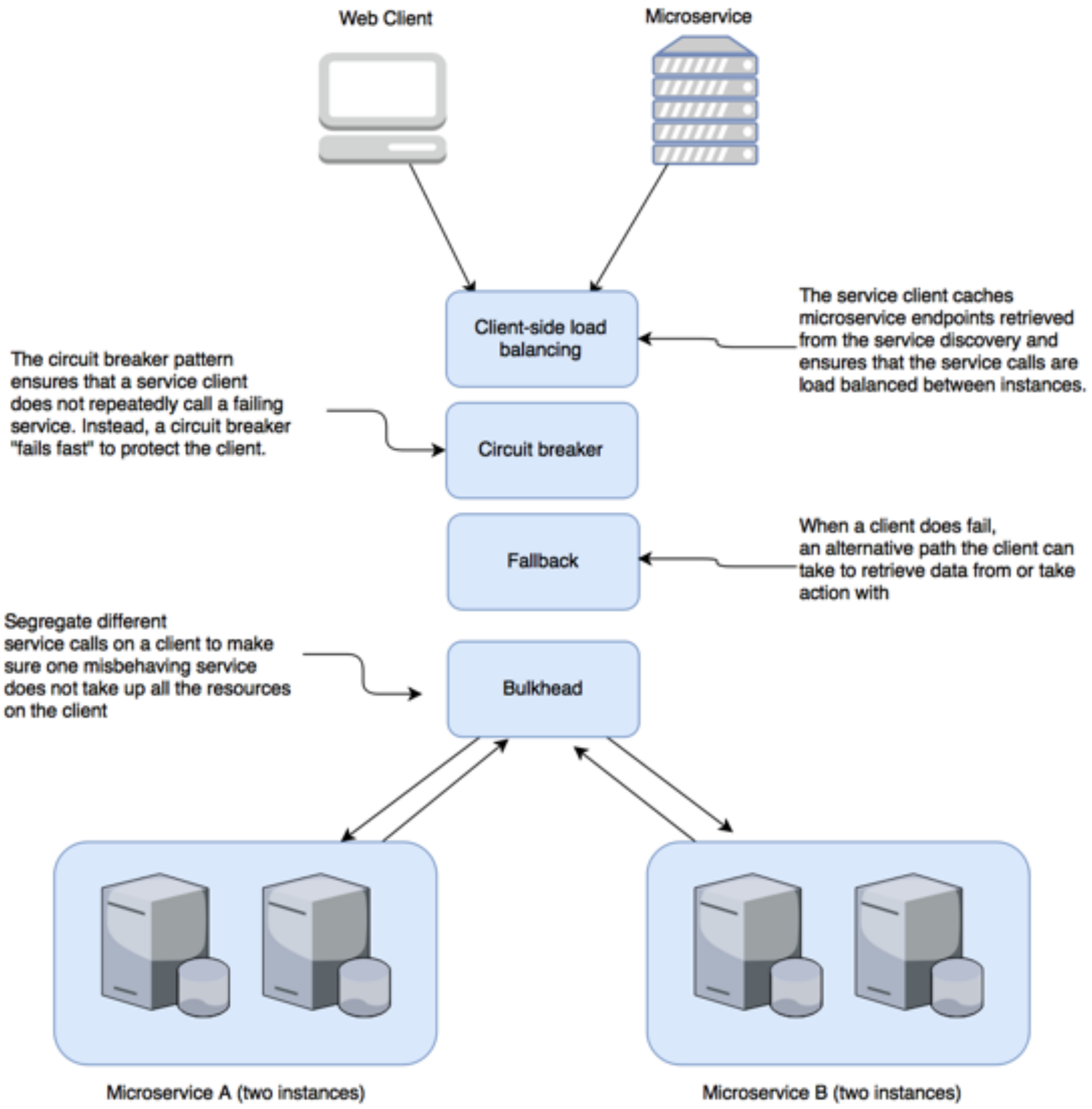
Service discovery abstracts away the physical location of the service from the client. New microservice instances can be added to scale up, and unhealthy service instances can be transparently removed from the service.

- In a cloud-based application, you might have hundreds of microservice instances running. You'll need to abstract away the physical IP address of these services and have a single point of entry for service calls so that you can consistently enforce security and content policies for all service calls.

- Service Routing - a single entry point for all of your services so that security policies and routing rules are applied uniformly to multiple services and service instances in your microservice applications

- Service Discovery - make your microservice discoverable so client applications can find them without having the location of the service hard- coded into the application

6.3 Resiliency Patterns


Web Client
Microservice
Client-side load balancing
Circuit breaker
Fallback
Bulkhead
Microservice A (two instances)
Microservice B (two instances)

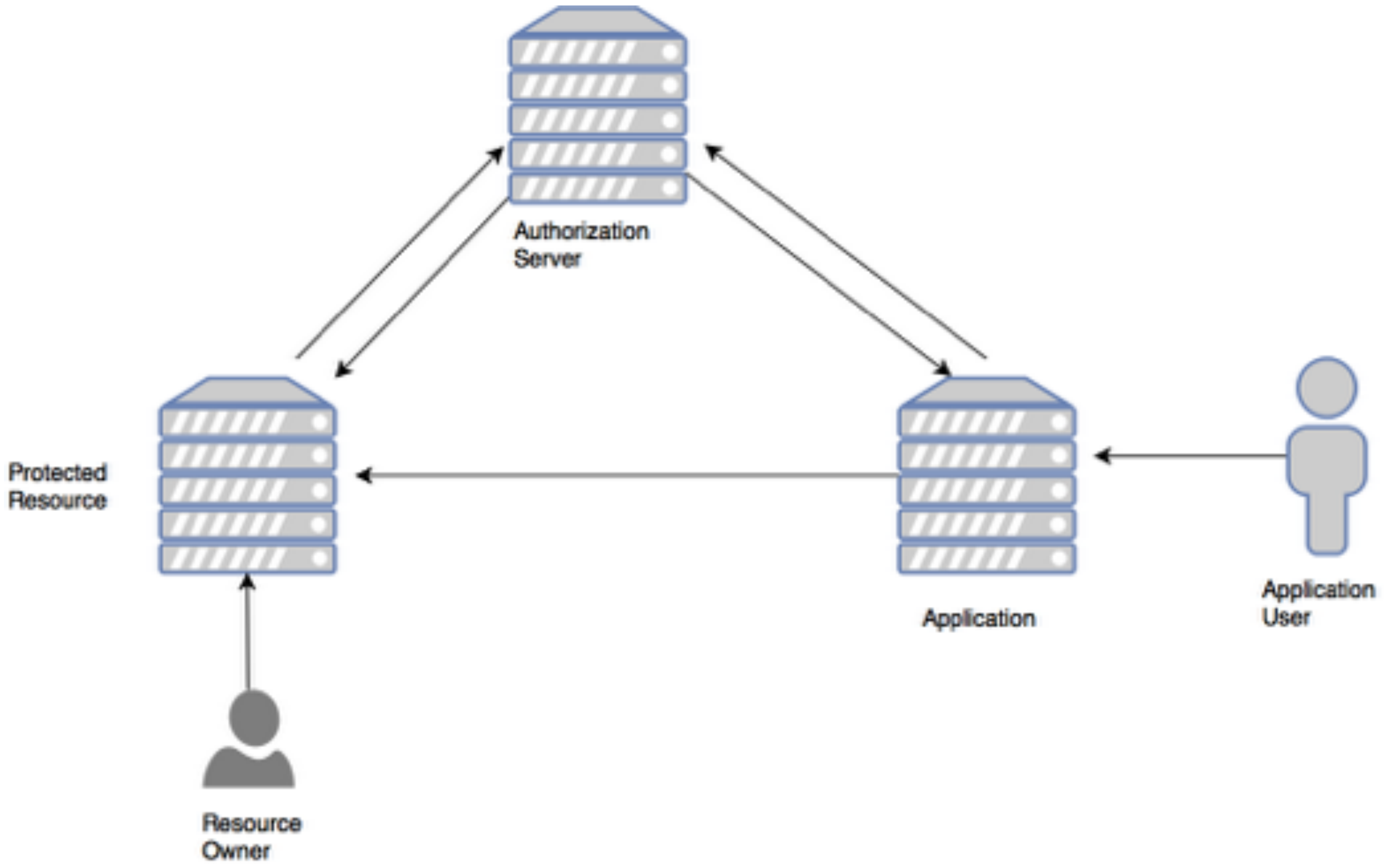
The service client caches microservice endpoints retrieved from the service discovery and ensures that the service calls are load balanced between instances.

The circuit breaker pattern ensures that a service client does not repeatedly call a failing service. Instead, a circuit breaker "fails fast" to protect the client.

When a client does fail, an alternative path the client can take to retrieve data from or take action with

Segregate different service calls on a client to make sure one misbehaving service does not take up all the resources on the client

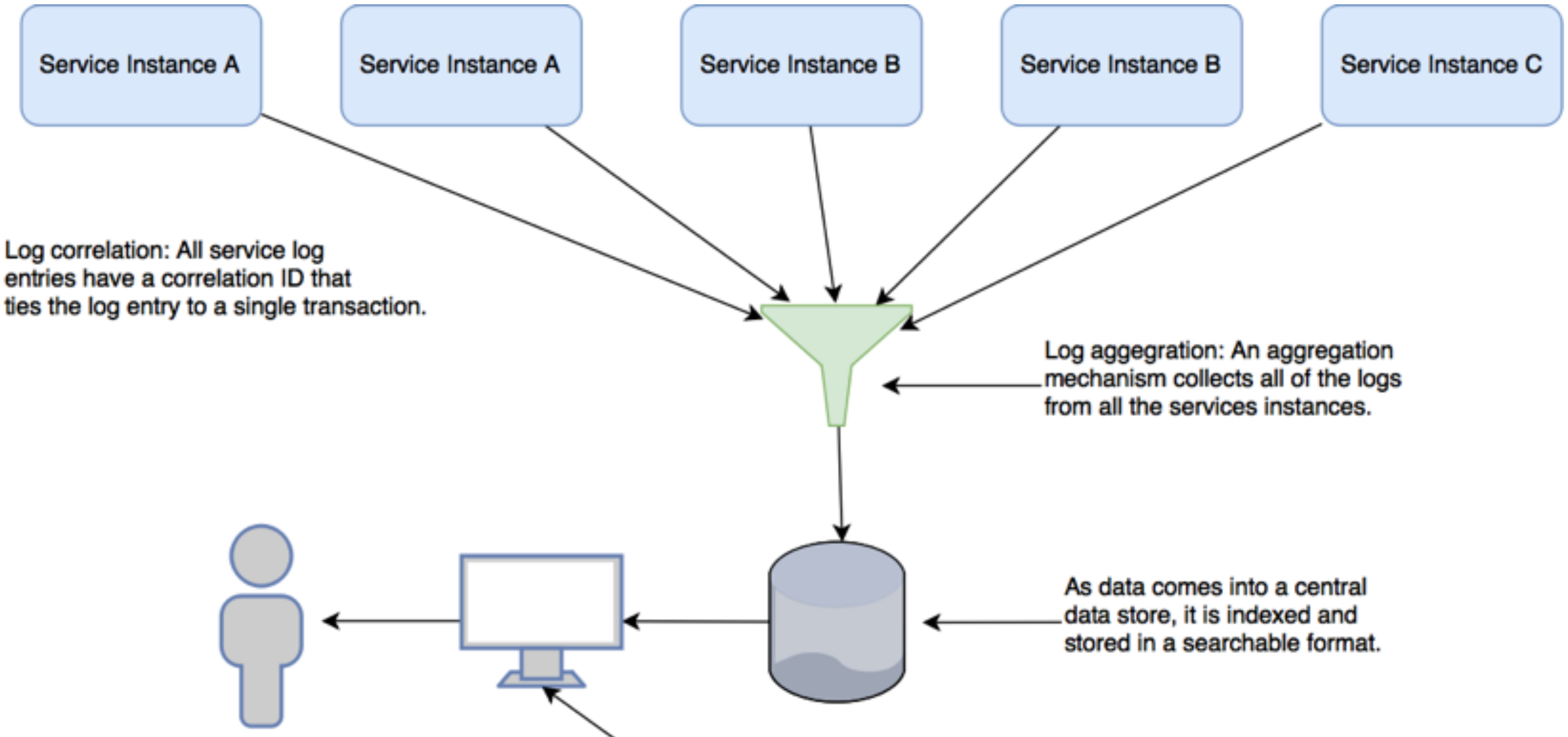
6.4 Security Patterns



- Authentication - determine who the user is
- Authorization - determine what privileges the user has
- Credential management and propagation - avoid constantly presenting credentials for service calls.

We will look at OAuth2 and JWT

6.5 Logging and Tracing Patterns



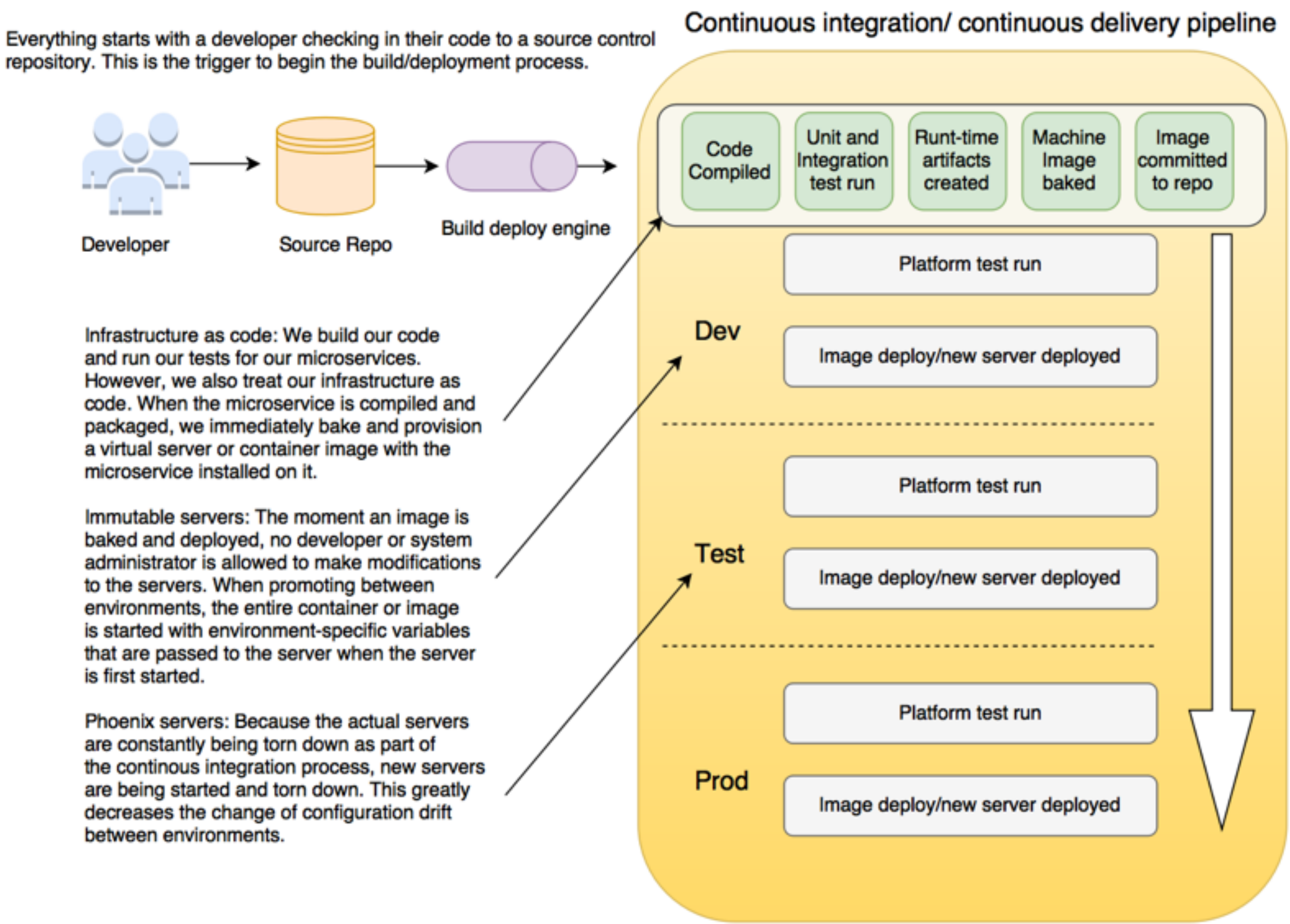
Log correlation: All service log entries have a correlation ID that ties the log entry to a single transaction.

Log aggegration: An aggregation mechanism collects all of the logs from all the services instances.

As data comes into a central data store, it is indexed and stored in a searchable format.

Microservice transaction tracing: The development and operations teams can query the log data to find individual transactions. They should also be able to visualize the flow of all the services involved in a transaction.

6.6 Build and Deploy Patterns



Everything starts with a developer checking in their code to a source control repository. This is the trigger to begin the build/deployment process.

Developer

Source Repo

Build deploy engine

Infrastructure as code: We build our code and run our tests for our microservices. However, we also treat our infrastructure as code. When the microservice is compiled and packaged, we immediately bake and provision a virtual server or container image with the microservice installed on it.

Immutable servers: The moment an image is baked and deployed, no developer or system administrator is allowed to make modifications to the servers. When promoting between environments, the entire container or image is started with environment-specific variables that are passed to the server when the server is first started.

Phoenix servers: Because the actual servers are constantly being torn down as part of the continous integration process, new servers are being started and torn down. This greatly decreases the change of configuration drift between environments.

Continuous integration/ continuous delivery pipeline

Code Compiled

Unit and Integration test run

Runt-time artifacts created

Machine Image baked

Image committed to repo

Dev

Platform test run

Image deploy/new server deployed

Test

Platform test run

Image deploy/new server deployed

Prod

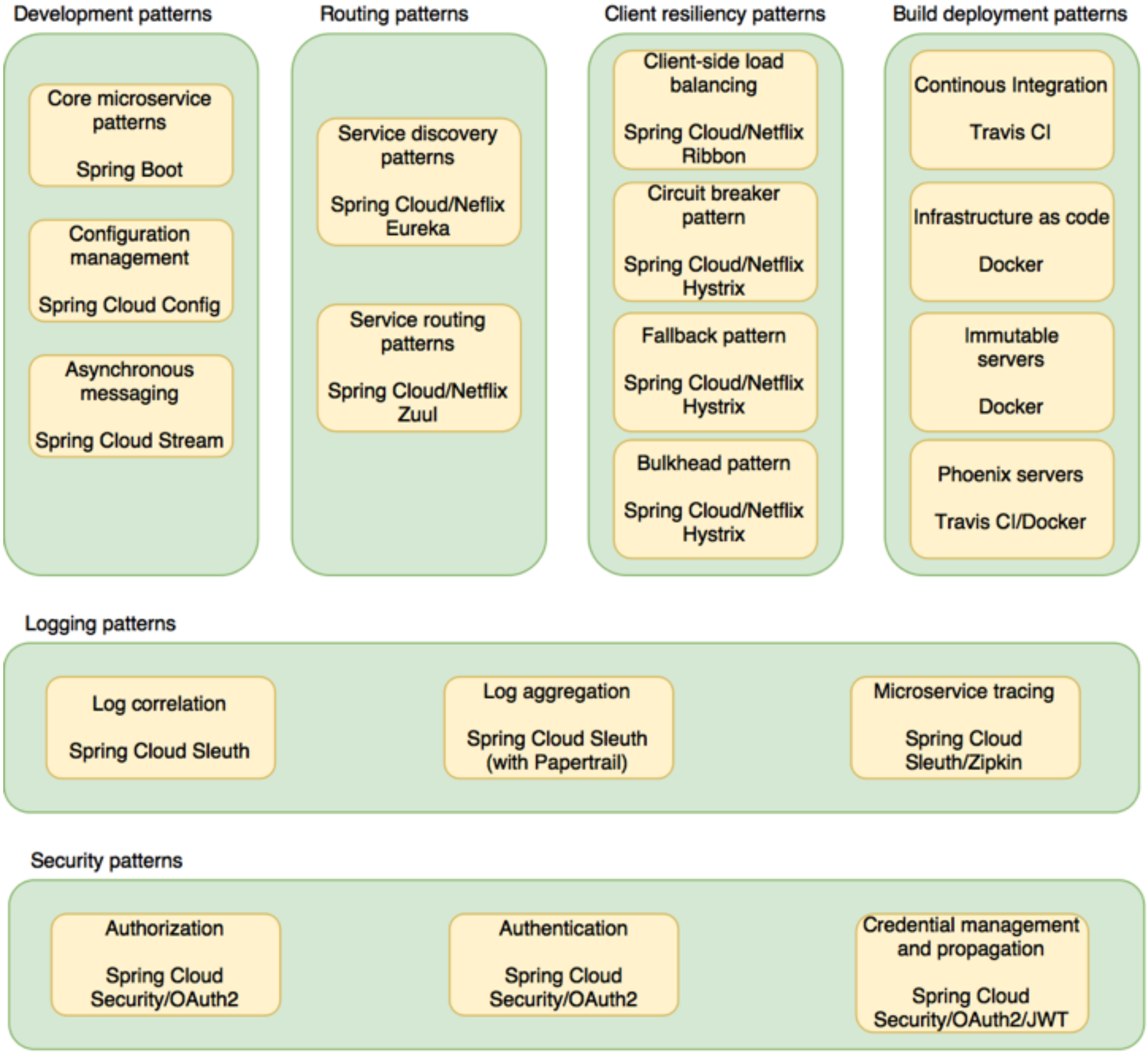
Platform test run

Image deploy/new server deployed

7. Using Spring Cloud in Building your Microservices

- Implementing all patterns from scratch would be a tremendous amount of work.

- Spring team has integrated a wide number of **battle-tested** open source projects into a Spring subproject collectively known as **Spring Cloud**



Development patterns

Core microservice patterns

Spring Boot

Configuration management

Spring Cloud Config

Asynchronous messaging

Spring Cloud Stream

Routing patterns

Service discovery patterns

Spring Cloud/Neflix Eureka

Service routing patterns

Spring Cloud/Netflix Zuul

Client resiliency patterns

Client-side load balancing

Spring Cloud/Netflix Ribbon

Circuit breaker pattern

Spring Cloud/Netflix Hystrix

Fallback pattern

Spring Cloud/Netflix Hystrix

Bulkhead pattern

Spring Cloud/Netflix Hystrix

Build deployment patterns

Continous Integration

Travis CI

Infrastructure as code

Docker

Immutable servers

Docker

Phoenix servers

Travis CI/Docker

Logging patterns

Log correlation

Spring Cloud Sleuth

Log aggregation

Spring Cloud Sleuth (with Papertrail)

Microservice tracing

Spring Cloud Sleuth/Zipkin

Security patterns

Authorization

Spring Cloud Security/OAuth2

Authentication

Spring Cloud Security/OAuth2

Credential management and propagation

Spring Cloud Security/OAuth2/JWT

- **Spring Boot** - core technology used in microservice implementation

- **Spring Cloud Config** - handles the management of application configurations data through a centralized service

- **Spring Cloud Service Discovery** - abstract away the physical location of where your servers are deployed from the clients consuming the service

- **Spring Cloud/Netflix Hystrix and Ribbon** - Hystrix for circuit breaker and bulkhead pattern; Ribbon for client-side load-balancing and integrating with Eureka

- **Spring Cloud/Netflix Zuul** - provide service routing capabilities

- **Spring Cloud Stream** - integrate lightweight message processing into microservice

- **Spring Cloud Sleuth** - integrate unique tracking identifiers into the HTTP calls and message channels (RabbitMQ, Apache Kafka) being used in the app

- **Spring Cloud Security** - authentication and authorization framework; token-based communication

- **Travis and Docker** - provisioning implementations