CitiusTech

Java Microservices with Netflix OSS and Spring

Version 1.0

December 2019

Agenda

- **Monolithic Architecture**
- What & Why of Microservices?
- Core characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

Monolithic Architecture

- To understand Microservices, let's first look at monolithic software. In monolithic software, we mainly use a three-tier architecture:

Monolithic approach Drawbacks:

- **Assembling and deploying challenges:** All code (presentation, business layer, and data access layer) is maintained within the same code base and deployed as a single unit . As there is one codebase, it grows gradually. Every programmer, whether it's a UI Developer or a business layer developer, commits in same code base, which becomes very inefficient to manage.

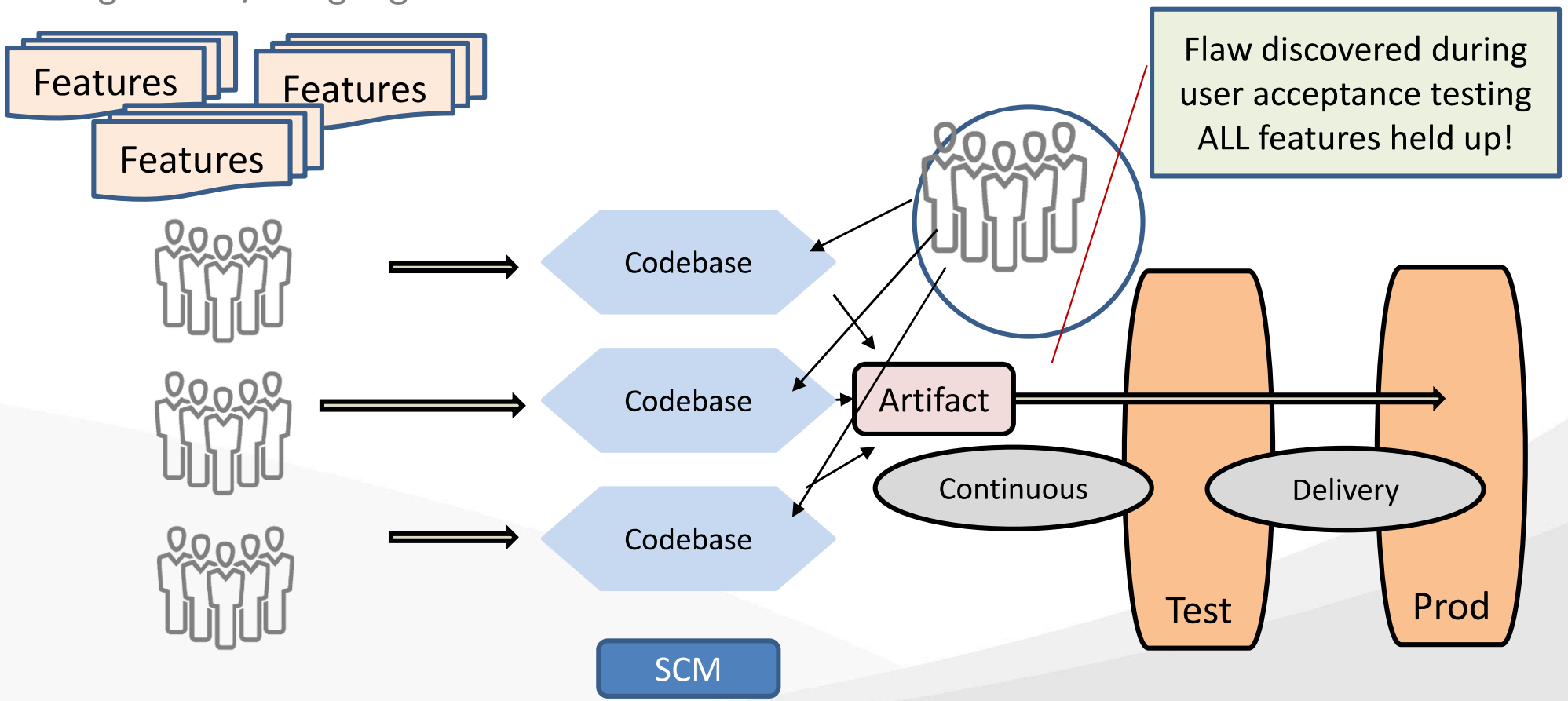
  Example : Suppose one developer only works in the JMS module, but he has to pull the whole codebase to his local and configure the whole module in order to run it on a local server. Why? He should only concentrate on the JMS module, but the current scenario doesn't allow for that.

Monolithic applications can evolve into a “big ball of mud”; a situation where no single developer (or group of developers) understands the entirety of the application.

Monolithic Challenges (1/2)

- Issue is not coding --Think about assembling and deploying challenges
- Cannot independently deploy single change to single component
- Changes are “held-hostage” by other changes

Using Teams / Language Constructs

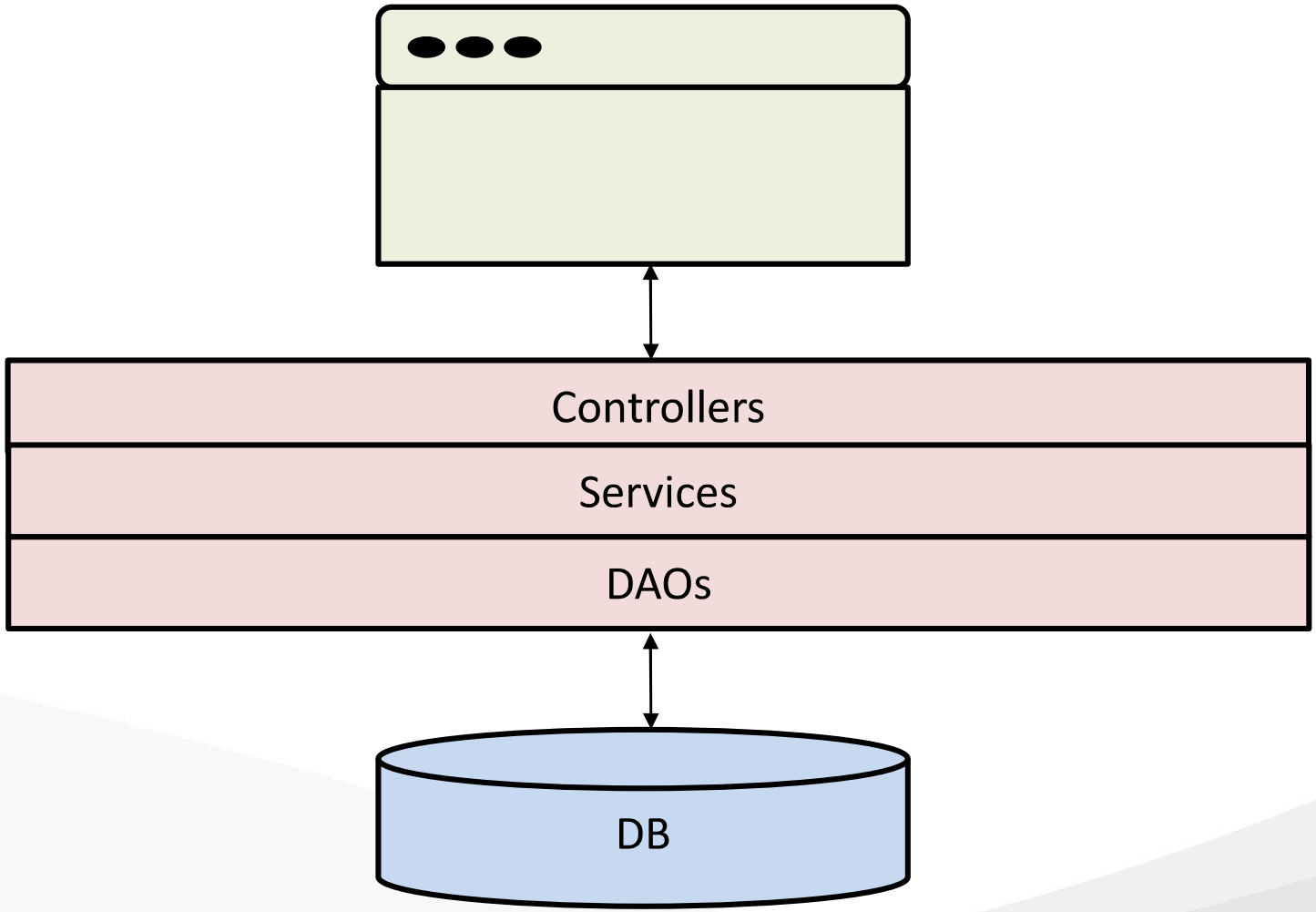


Monolithic Challenges (2/2)

- **Tight Coupling among Modules:** As there is one code base and modules are dependent on each other, minimal change in one module needs to generate all artifacts and needs to deploy in each server pool in a distributed environment.

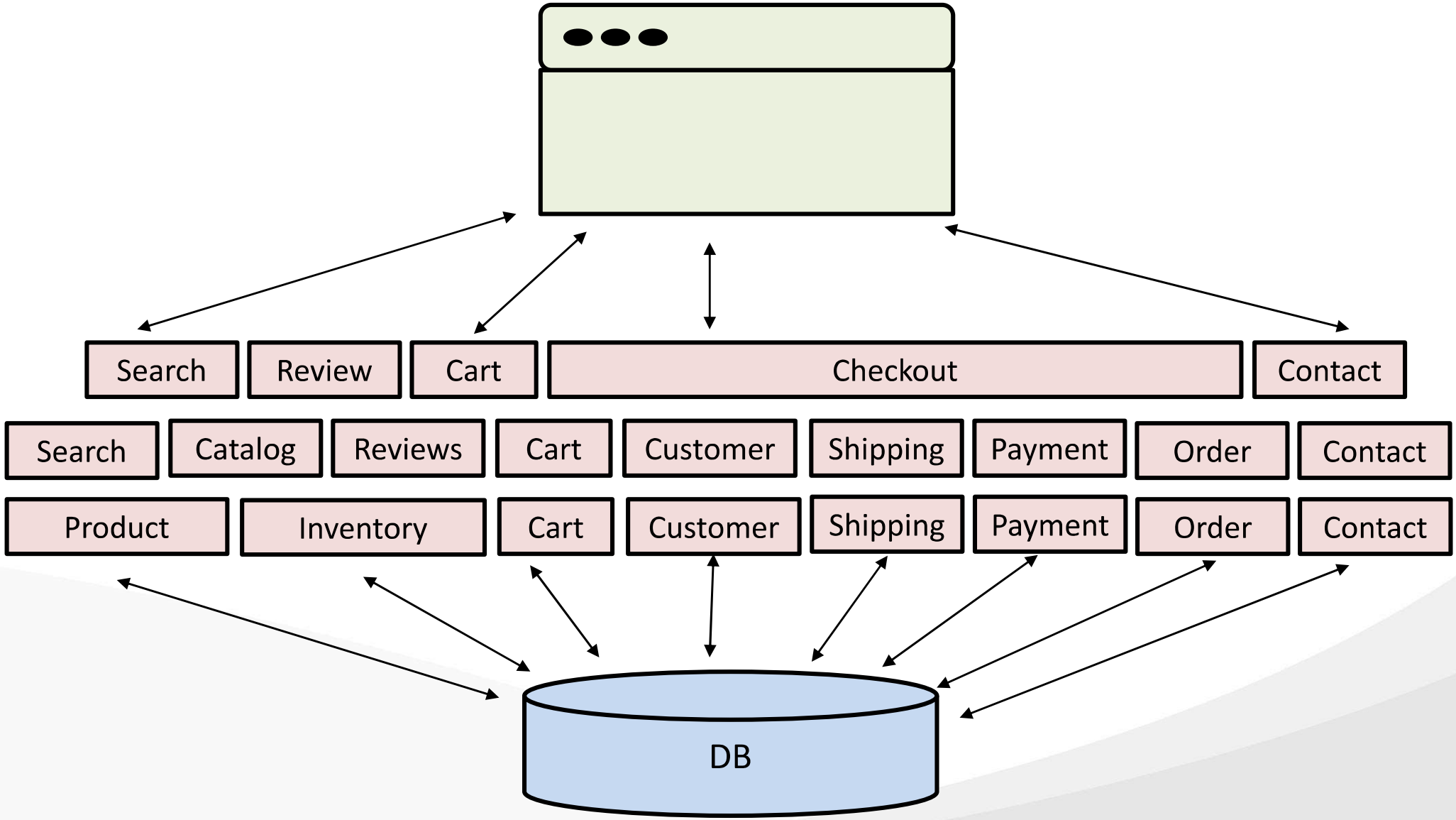
  Example : Suppose in a multi-module project where the JMS module and business module are dependent on the data access module. A simple change in the data access module means we need to re-package the JMS module and business module and deploy them in their server pool.
- **Team is based on Tier**: Often, we create teams based on the tier — UI developers, backend developers, database programmers, etc. They are experts in their domains, but they have little knowledge about other layers. So, when there's a critical problem, it encompasses each layer, and the blame game starts. Not only that, but it takes additional time to decide which layer's problem it is and who needs to solve the issue
- **Language/Framework Lock :** By definition monolithic applications are implemented using a single development stack (i.e., JEE or .NET), which can limit the availability of "the right tool for the job." We cannot experiment/ take advantage of emerging Technologies
- **Digestion :**
  - Single Developer cannot digest a large Codebase
  - Single team cannot manage a single large application
    <<Amazon's "2 Pizza "rule >>
- Scaling becomes challenging
- **CI/CD:** Continuous integration/ deployment become complex and time consuming. You may require dedicated team for build and deploy

Monolithic Challenges – Shopping Cart Application (1/4)



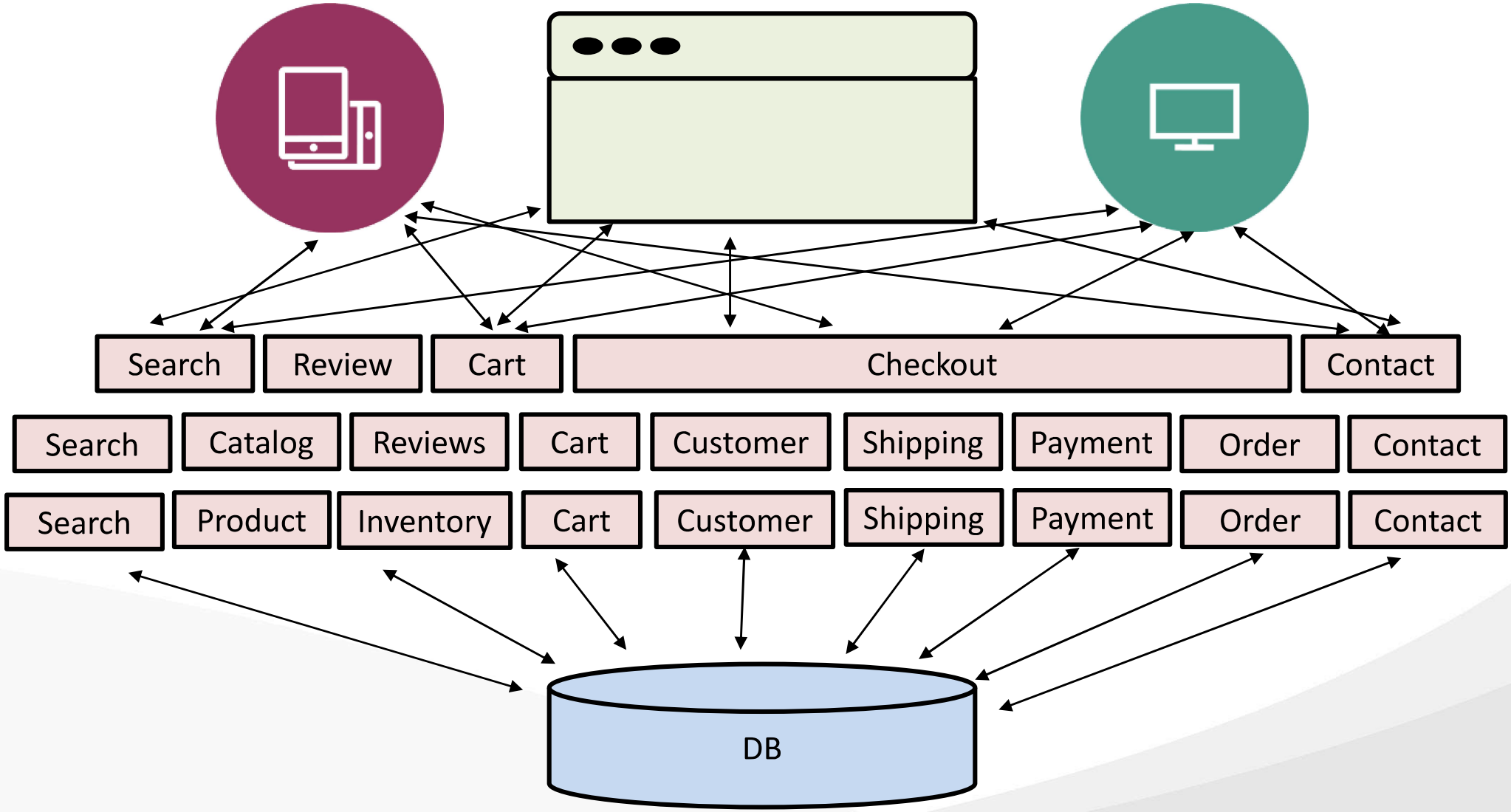
Monolithic Challenges – Shopping Cart Application (2/4)

Understanding the Monolithic Architecture



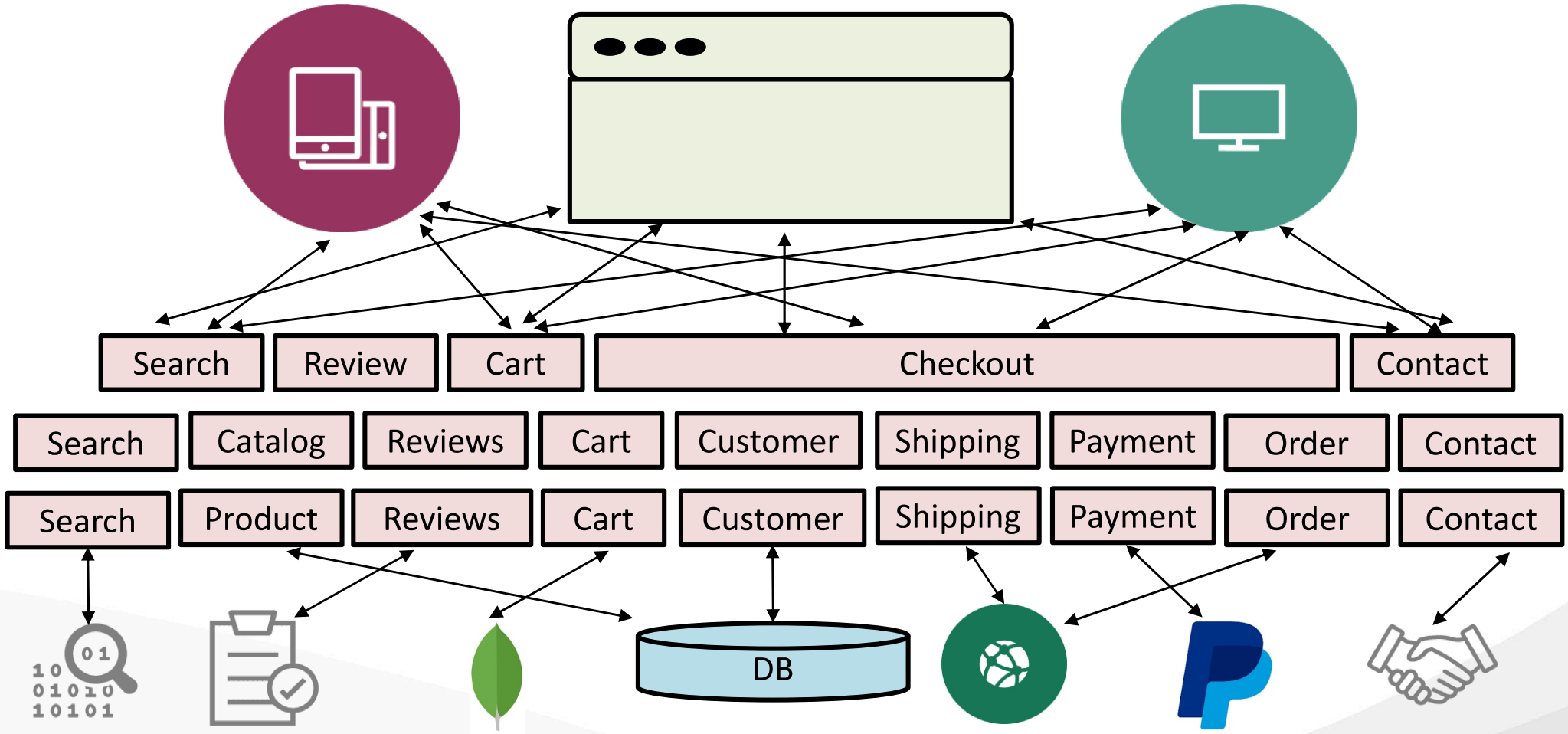
Monolithic Challenges – Shopping Cart Application (3/4)

New type of client applications



Monolithic Challenges – Shopping Cart Application (4/4)

New type of persistence/ services



We are more likely to embrace a plethora of backend Technologies, and our Single Application using Single language must stretch a lot to accommodate various types of backend needs.

Agenda

- Monolithic Architecture
- **What & Why of Microservices?**
- Why Microservices?
- Core characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

What is Microservices?

- An Architecture Style
- An alternative to more traditional ‘monolithic’ applications
- Decomposition of single system into a suite of small services, each running as independent processes and intercommunicating via open Protocols (HTTP/ Rest or messaging)
- Developing single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often HTTP resource API –**Martin Fowler**
- Fined-grained SOA -Adrian Cockcroft -**Netflix**
- Microservices are not a single Bullet. It involves drawbacks and risks

Working Definition

- Composing a single application using a suite of small services Rather than single, monolithic application
- Each running as independent process not merely modules/components with a single executable
- Intercommunication via open protocols ,Like HTTP/REST or messaging
- Separately written, deployed, scaled and maintained ,Potentially in different languages
- Services encapsulate business capability ,Rather than language constructs (classes, packages) as primary way to encapsulate
- Services are independently replaceable and upgradable

Why are Microservices Architectures Popular?

Desire for faster changes

Need for greater availability

Motivation for fine-grained scaling

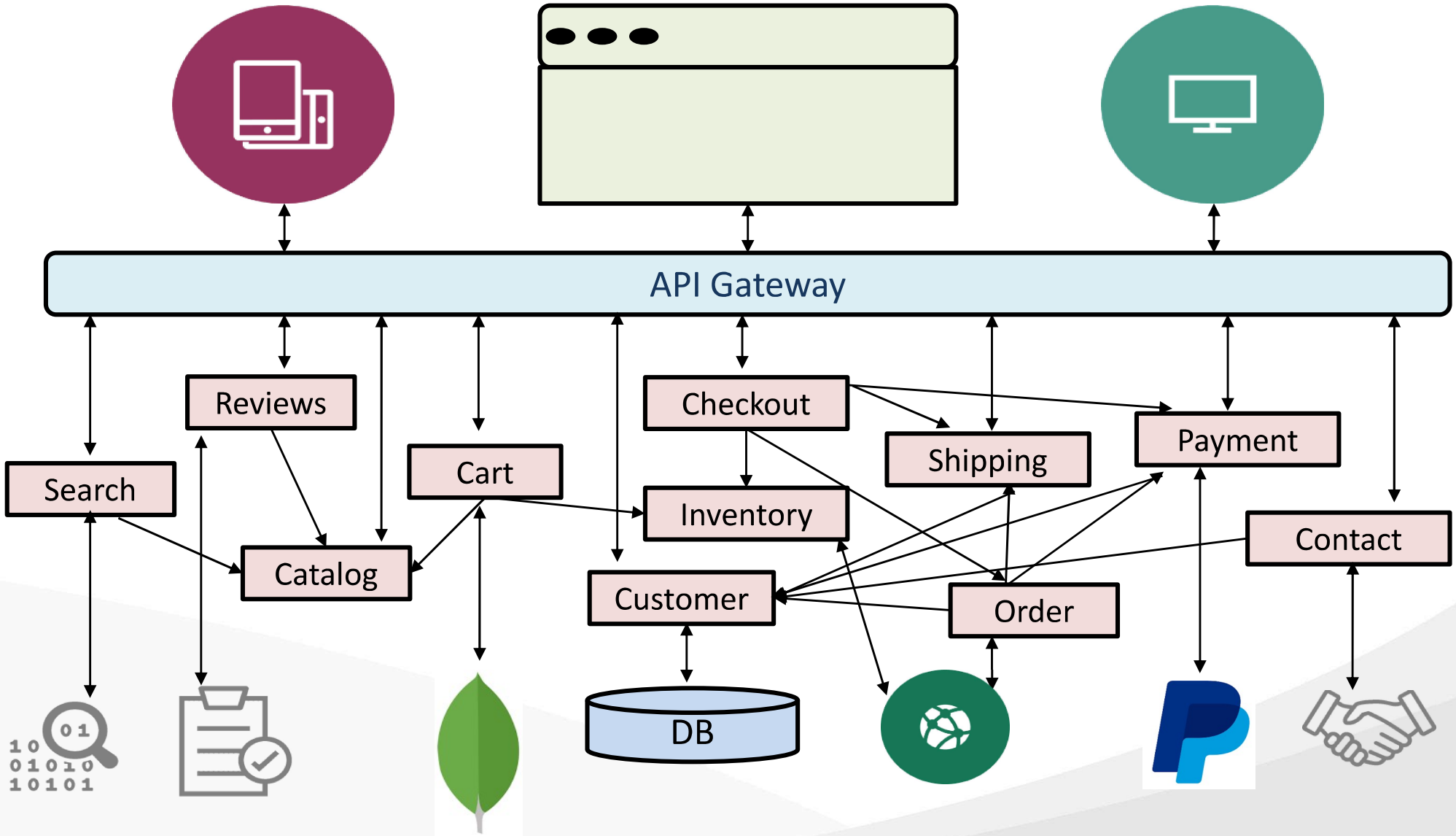
Compatible with a DevOps mindset

Agenda

- Monolithic Architecture
- What & Why of Microservices?
- **Core characteristics of Microservices**
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

Core Characteristics of Microservices (1/6)

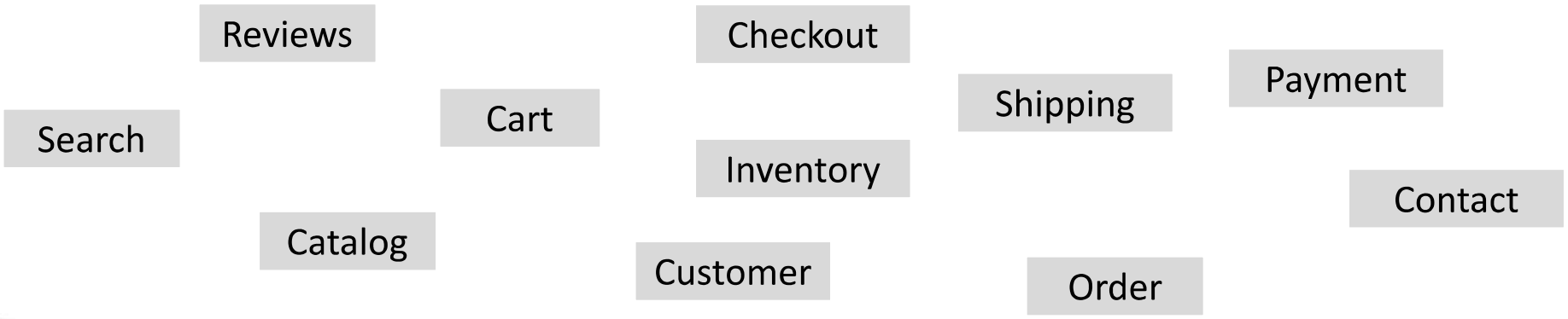
Microservices architecture



Core Characteristics of Microservices (2/6)

Componentization via Services:

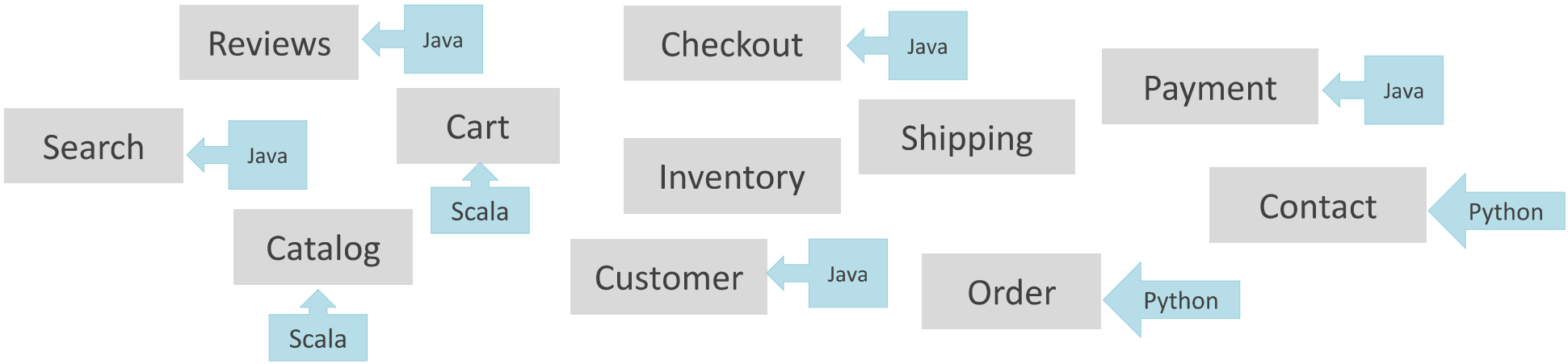
- NOT language constructs
- Where services are small, independently deployable applications
- Forces the design of clear interfaces
- Changes scoped to their affected service



Core Characteristics of Microservices (3/6)

Composed using suite of small

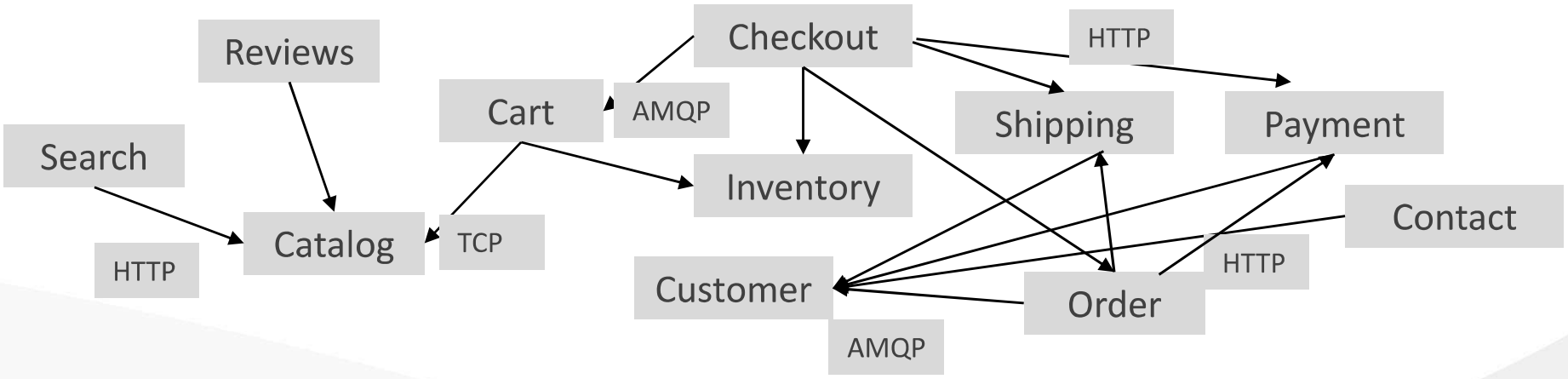
- Services are small, independently deployable
- Not a single codebase
- Polygot Programming- Not (necessarily) a single language/framework
- Modularization not based on language /framework constructs



Core Characteristics of Microservices (4/6)

Communication based on lightweight protocols

- HTTP, TCP, UDP, Messaging etc.
  - Payloads: JSON, BSON, XML, Protocol Buffers etc.
- Forces the design of clear interfaces
- Netflix's Cloud Native Architecture – Communicate via API's
  - Not Common Database



Core Characteristics of Microservices (5/6)

Services encapsulate business capabilities

- Not based on Technology Stack
- Vertical slices by business function (i.e. cart, catalog, checkout)
- Suitable for cross-functional teams

Search	Reviews	Cart	Contact
PUT/Search	GET/review/123 POST/review	POST/cart GET/cart/123 POST/cart/123/item DELETE/cart/123 PUT/cart/123/item/1 DELETE/cart/123/item/1	GET/review/123 POST/post

Core Characteristics of Microservices (6/6)

Decentralized Governance

- Use right tool (language, framework) for the job
- Services evolve at different speeds, deployed and managed according to different needs
- Consumer-Driven Contracts
- Antithesis of ESB
  - Services are not Orchestrated but Choreographed

Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- **Challenges of Microservices and Importance**
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

Challenges of Microservices and Importance

- Complexity has moved out of the application but into the operation layer
  - Fallacies of Distributed computing
- Services may be unavailable
  - Never needed to worry about this in Monolith!
  - Design for failure, Circuit breakers
    - “Everything fails all the time “—Werner Vogels, CTO Amazon
  - Much more monitoring needed
- Remote calls more expensive than in-process calls
- Transactions : Must rely on eventual consistency over ACID
- Features span multiple services
- Change management becomes a different challenge
  - Need to consider the Interaction of Services
  - Dependency management/versions

Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- Challenges of Microservices and Importance
- **What is Spring Cloud?**
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

What is Spring Cloud ?

- Spring Cloud is a collection of tools from Pivotal that provides solutions to some of the commonly encountered patterns when building distributed systems
- Spring Cloud act as a Scaffolding for Microservices.

Released March 2015

Build common distributed systems patterns

Open source software

Optimized for Spring apps

Run anywhere

Includes Netflix OSS technology

Important Spring Cloud Projects

Spring Cloud Eureka	Spring Cloud Hystrix	Spring Cloud Ribbon
Spring Cloud Zuul	Spring Cloud Stream	Spring Cloud Flow

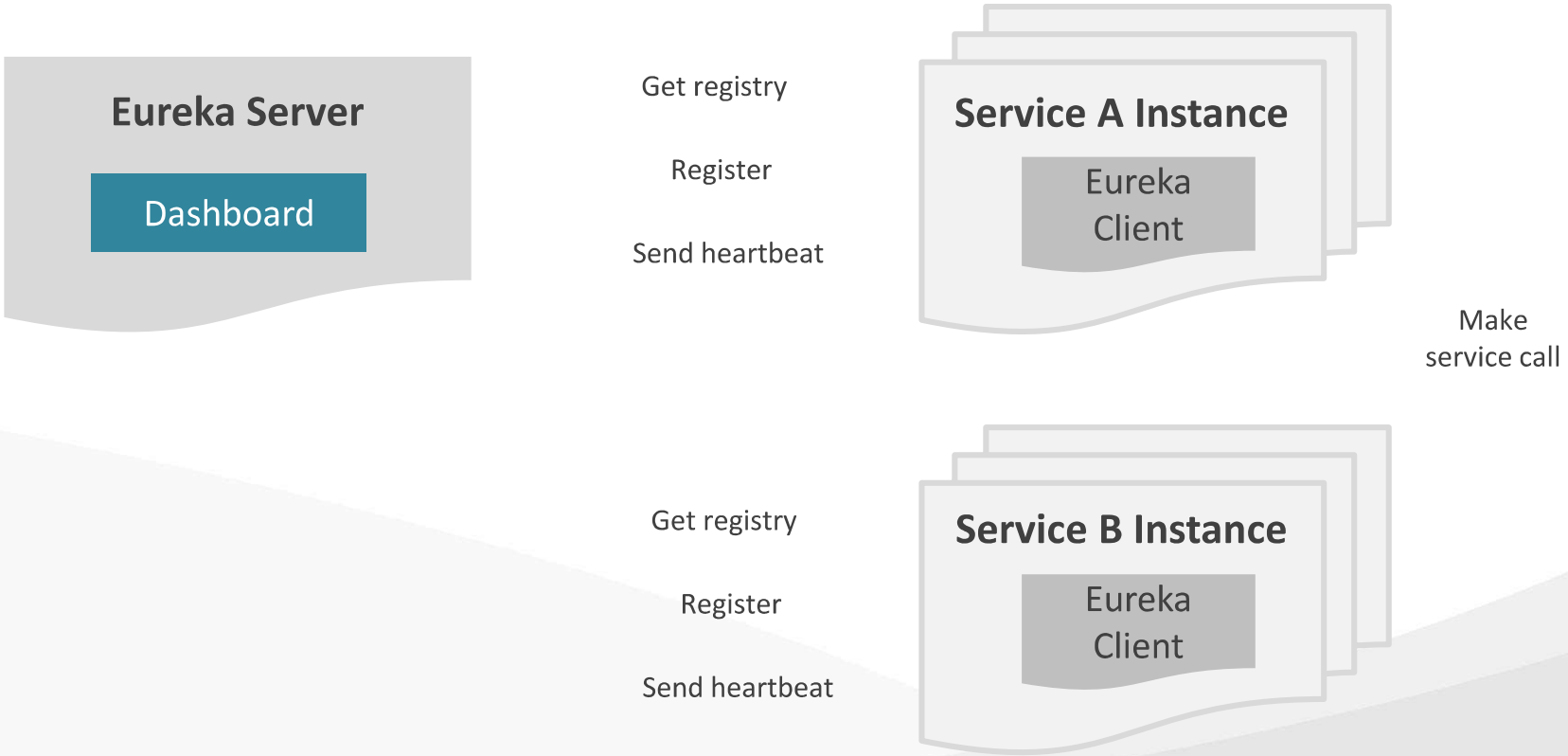
Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- **Overview of Netflix tools**
- Demo – Toll rate service
- Microservices Design Pattern
- Microservices Usage Pattern

Overview of Netflix tools (1/10)

Eureka (Service Discovery)- Registry that act as a phone-book for services

- Role of Service discovery in Microservices:
  - Recognize the dynamic environment
  - Have a live view of Healthy Services
  - Avoid hard coding to service locations
  - Centralized list of available services



Overview of Netflix tools (2/10)

Registering with Eureka

Eureka in classpath leads to registration

Service name, host info sent during bootstrap

@EnableDiscoveryClient and @EnableEurekaClient

Sends heartbeat every 30 seconds

Heartbeat can include health status

HTTP or HTTPS supported

Overview of Netflix tools (3/10)

Discovering a service with Eureka

@EnableDiscoveryClient and @EnableEurekaClient	Client works with local cache	Cache refreshed, Reconciled regularly
Manually load balance or use Ribbon	Can prefer talking to registry in closet Zone	May take multiple heartbeats to discover new services

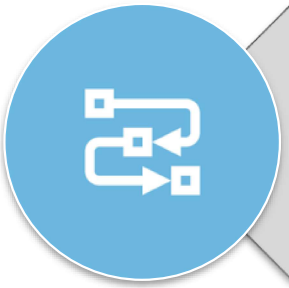
Overview of Netflix tools (4/10)

Spring Cloud Hystrix

Library for enabling
resilience in Microservices.

Overview of Netflix tools (5/10)

What Hystrix Does ?



Supported patterns include bulkhead, fail fast, graceful degradation (e.g. fail silently with fallback response).

Hystrix wraps calls to external dependencies and monitors metrics in real time. Invokes failover method when encountering exceptions, timeouts, thread pools exhaustion, or too many previous errors.

Hystrix periodically sends request through to see if service has recovered.

Overview of Netflix tools (6/10)

How Spring Cloud Hystrix works?

Circuit breaker via annotations at class, operation level

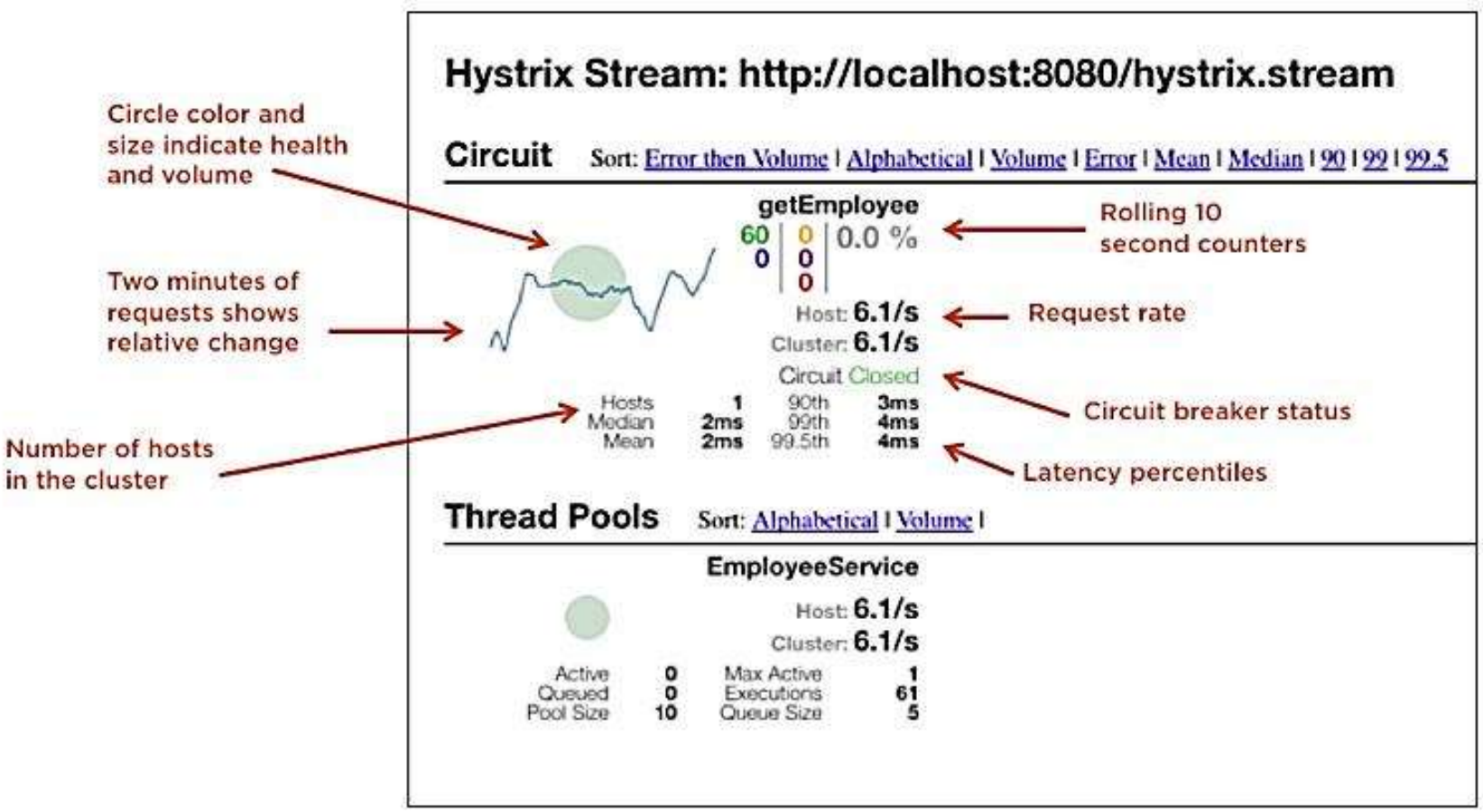
Hystrix manages the thread pool, emits metrics

Dashboard integrates with Eureka to look up services

Dashboard pulls metrics from instances or services

Overview of Netflix tools (7/10)

Hystrix Dashboard



Overview of Netflix tools (8/10)

Circuit breaker Pattern

- Used to wrap dangerous operations in a component
- Tripping of a circuit happens automatically in case of failure
- Differs from retries, this prevents usage instead of trying again
- Facilitates self- healing applications, as in case of open circuit application decides
  - To use another similar service
  - To stop using a service till timeout occurs

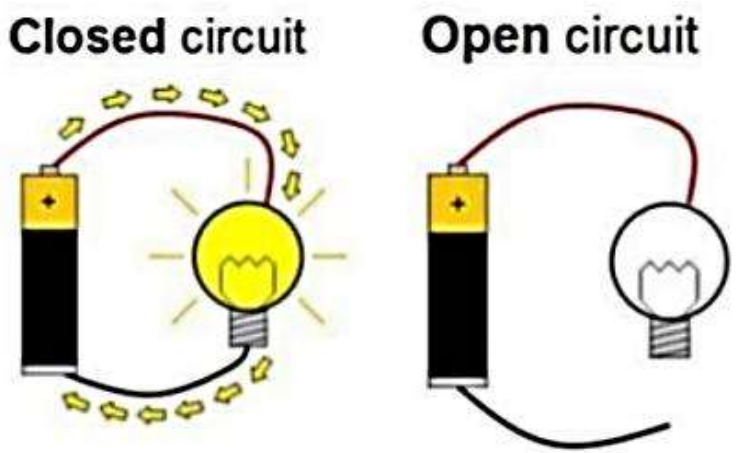
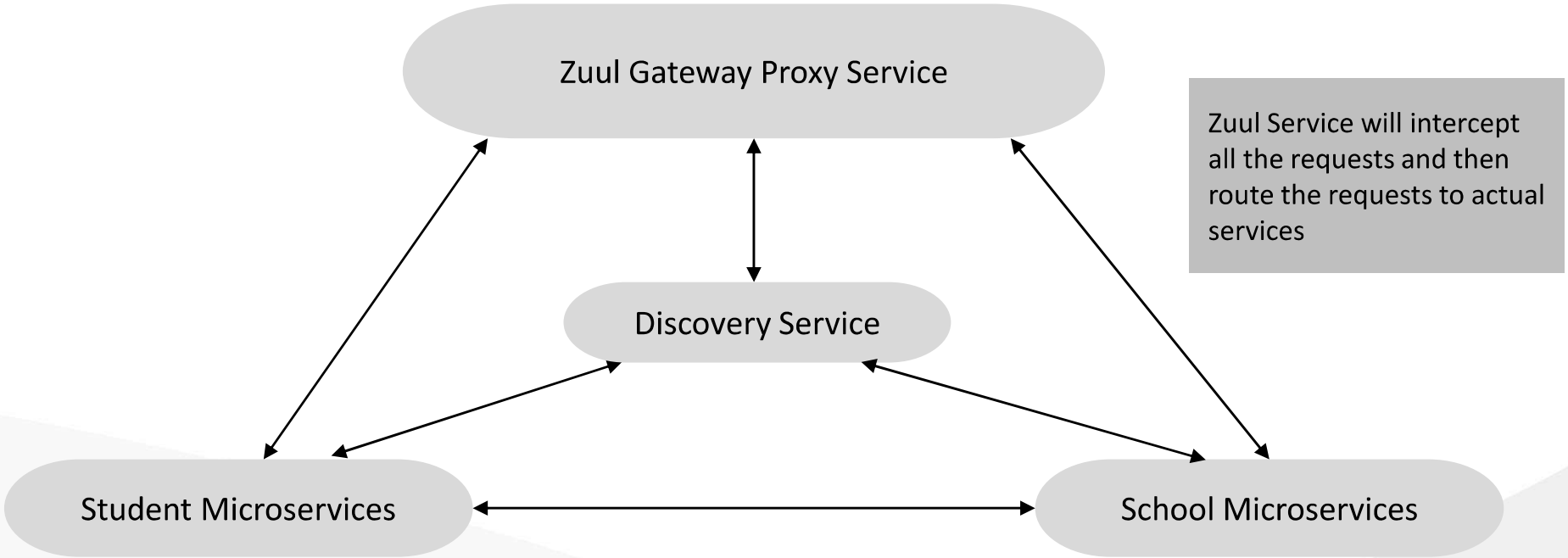


Figure 1: From internet

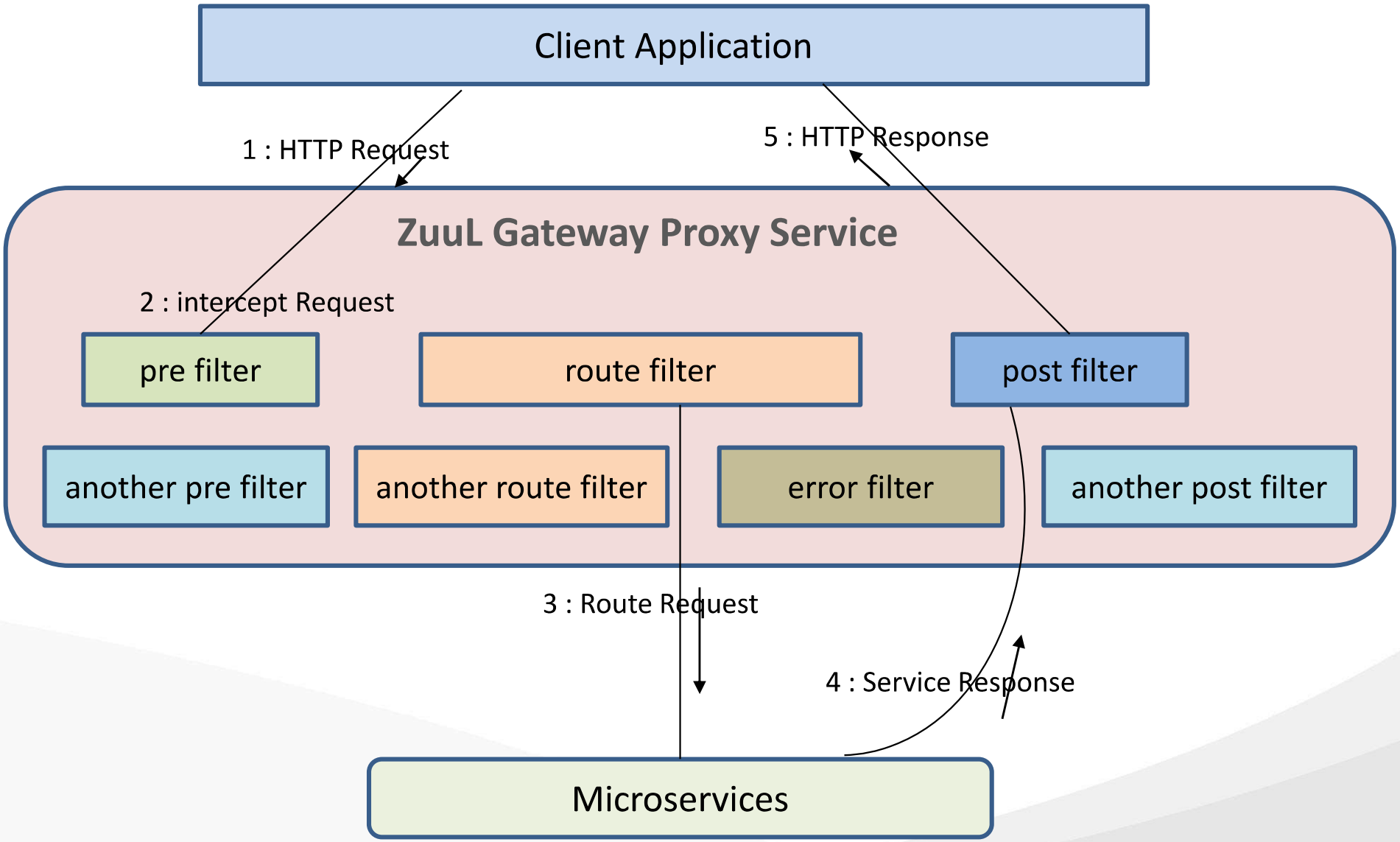
https://www.pinterest.com/pin/470696598529019437

Overview of Netflix tools (9/10)

- Zuul -Zuul is a JVM based router and server-side load balancer by Netflix. Zuul acts as an API gateway or Edge service. It receives all the requests coming from the UI and then delegates the requests to internal microservices.
- We usually create brand new microservice which is Zuul-enabled, and this service sits on top of all other microservices. It acts as an Edge service or client-facing service



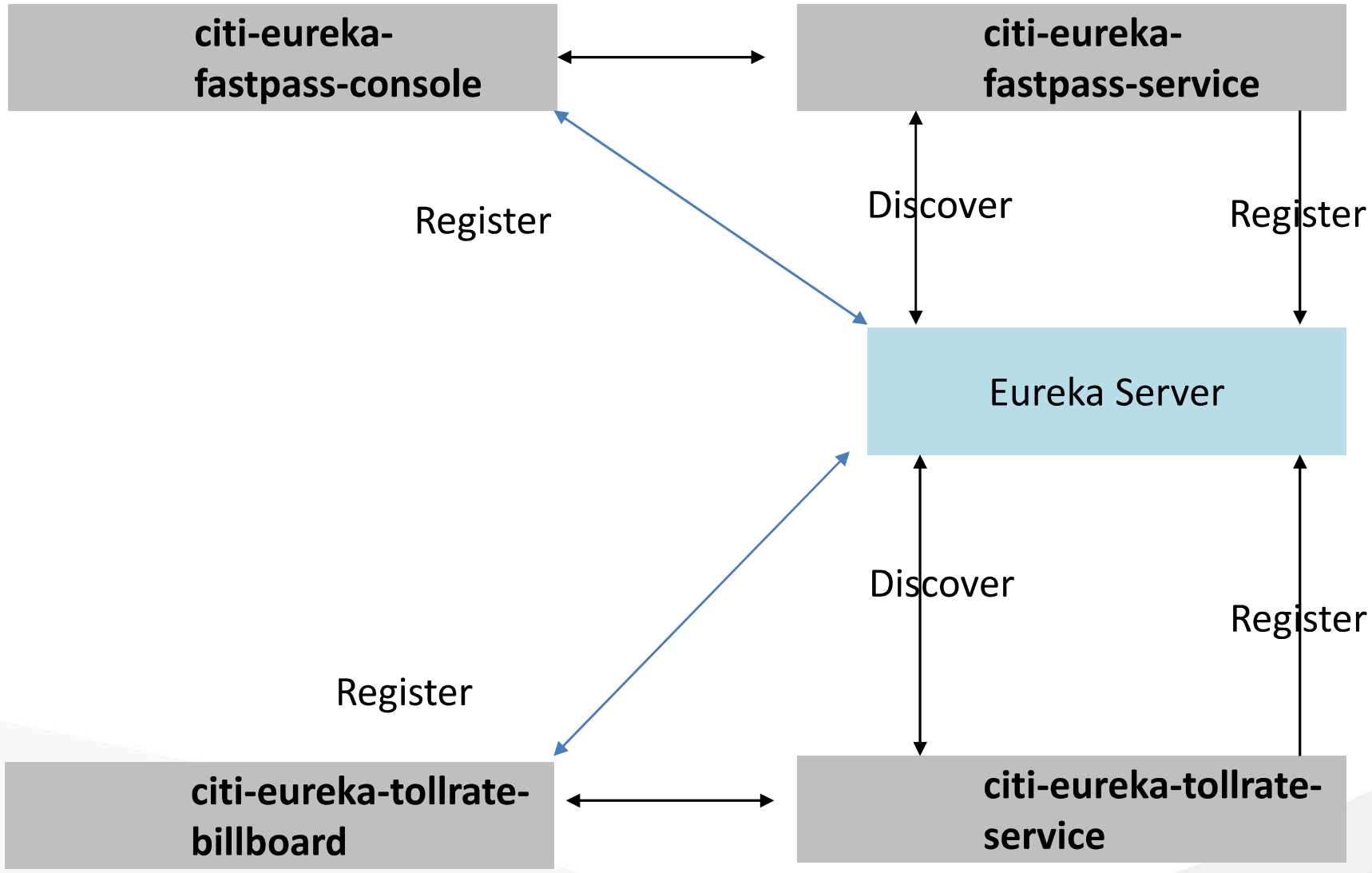
Overview of Netflix tools (10/10)



Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- **Demo – Toll rate service**
- Microservices Design Pattern
- Microservices Usage Pattern

Demo – Toll Rate Service (1/2)



Demo – Toll Rate Service (2/2)

List of Microservices

- **Providers**
  - **citi-eureka-tollrate-service**
    - Provides Toll rates http://localhost:49599/tollrate/2
  - **citi-eureka-fastpass-service**
    - Provides customer Details : http://localhost:64937/fastpass?fastpassid=101
- **Consumers** :
  - **citi-eureka-fastpass-console**
    - uses fastpass-service : http://localhost:8083/customerdetails?fastpassid=101
  - **citi-eureka-tollrate-billboard**
    - http://localhost:8082/dashboard?stationId=4

Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- **Microservices Design Pattern**
- Microservices Usage Pattern

Microservices Design Pattern

- Microservice architecture has become the de facto choice for modern application development. Though it solves certain problems, it is not a silver bullet. It has several drawbacks and when using this architecture, there are numerous issues that must be addressed. This brings about the need to learn common patterns in these problems and solve them with reusable solutions.
- Thus, design patterns for microservices need to be discussed. Before we dive into the design patterns, we need to understand on what principles microservice architecture has been built:
  - Scalability
  - Availibility
  - Resiliency
  - Independent ,Autonomous
  - Decentralized governance
  - Failure isolation
  - Auto-Provisioning
  - Continuous delivery through DevOps

Microservices Design Patterns : Decomposition (1/2)

- **Decompose by Business Capability**

  Intent : One o the strategy to decompose is by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except its business-oriented rather than technical.

- **Decompose by Subdomain**

  Intent: For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

Note: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

Microservices Design Patterns : Decomposition (2/2)

- **Strangler Pattern**

  Intent : The Strangler pattern is based on an analogy to a vine that strangles a tree that it's wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application "strangles" or replaces the original application until finally you can shut off the monolithic application..

Microservices Design Patterns : Integration (1/2)

API Gateway Pattern

Intent : An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above

- An API Gateway is the single point of entry for any microservice call.
- It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
- It can fan out a request to multiple services and aggregate the results to send back to the consumer.
- One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
- It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
- It can also offload the authentication/authorization responsibility of the microservice.

Microservices Design Patterns : Integration (2/2)

Aggregator Pattern

Intent : The talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

- An API Gateway is the single point of entry for any microservice call.
  - A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.
  - An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

Client-Side UI Composition Pattern

Intent: With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). This enables the app to refresh a particular region of the screen instead of the whole page.

Microservices Design Patterns : Database (1/4)

Shared Database per Service

Intent : This pattern dictates one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly.

For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service.

Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

Microservices Design Patterns : Database (2/4)

Shared Database per Service

- **Problem:**

  We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

- **Solution/Intent :**

  This pattern dictates one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly.

  For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service.

  Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

Microservices Design Patterns : Database (3/4)

Command Query Responsibility Segregation (CQRS)

- **Intent :**

  CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The event sourcing pattern is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

Microservices Design Patterns : Database (4/4)

Saga Pattern

- **Problem :** When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services?
- For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

Solution/Intent:

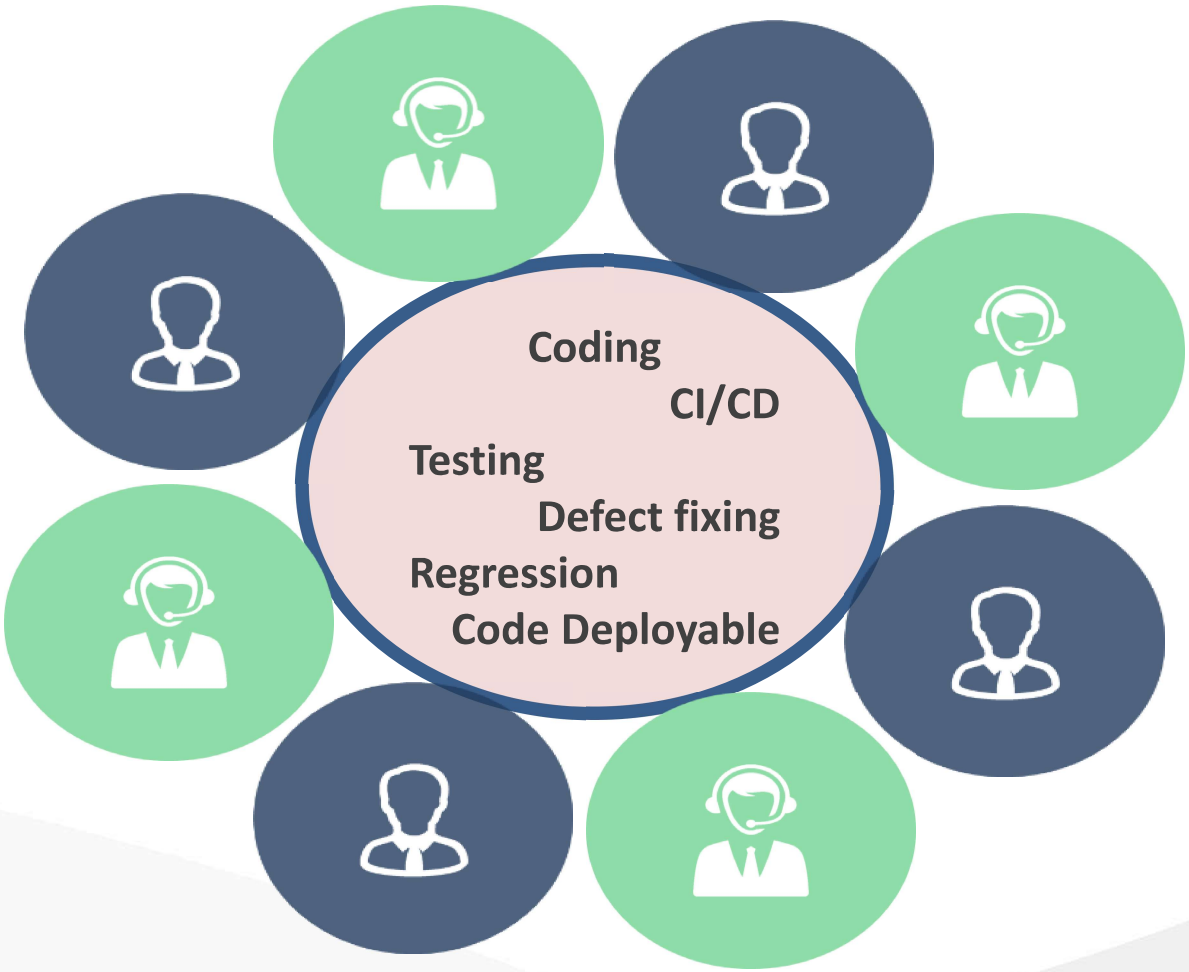
A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

- **Choreography** — When there is no central coordination, each service produces and listens to another service's events and decides if an action should be taken or not.
- **Orchestration —** An orchestrator (object) takes responsibility for a saga's decision making and sequencing business logic.

Agenda

- Monolithic Architecture
- What & Why of Microservices?
- Core Characteristics of Microservices
- Challenges of Microservices and Importance
- What is Spring Cloud?
- Overview of Netflix tools
- Demo – Toll rate service
- Microservices Design Pattern
- **Microservices Usage Pattern**

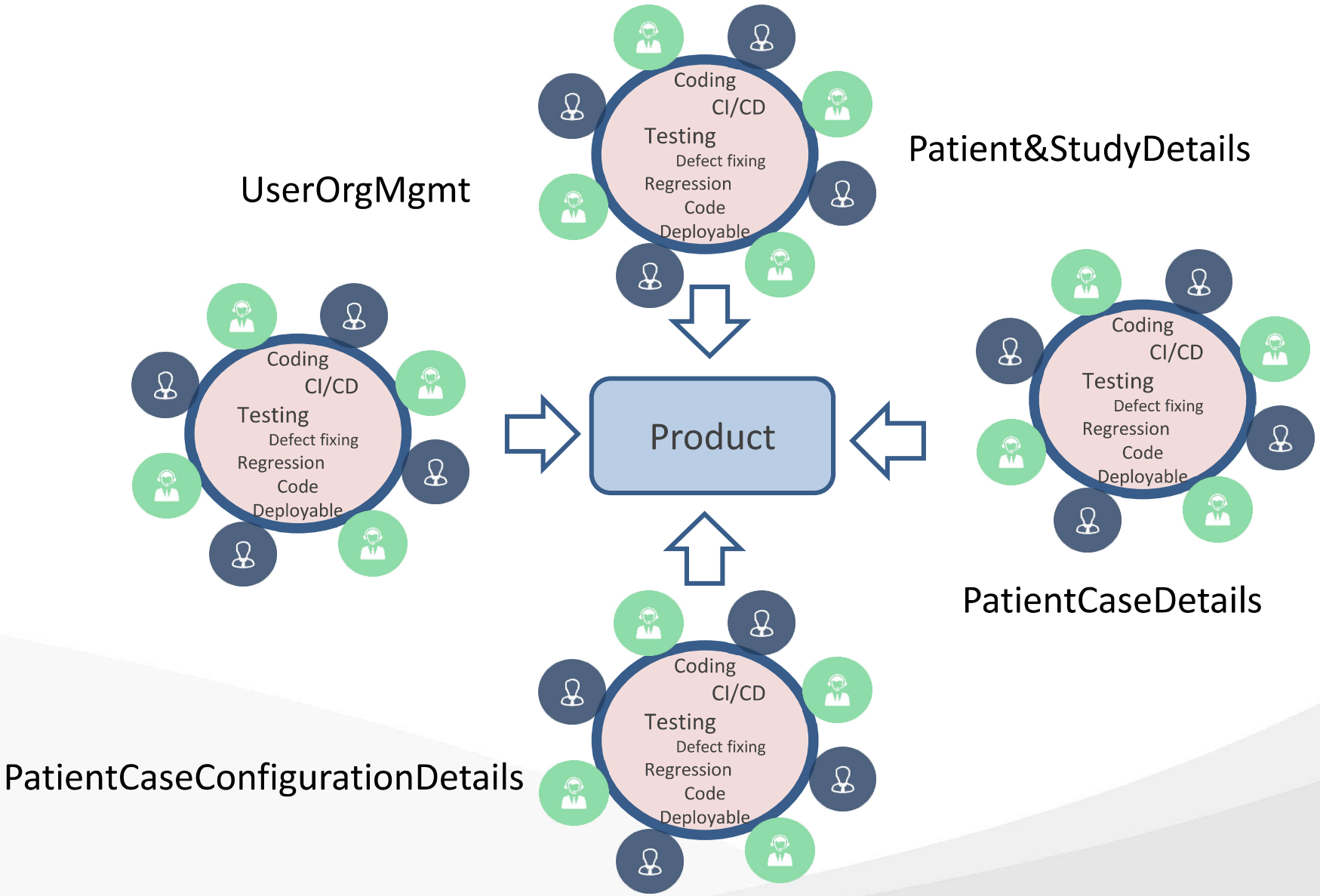
Microservices Usage in CT projects – Challenges faced (1/4)



Microservices Usage in CT projects – Challenges faced (2/4)

- Single huge code base (2380 code files)
- Bigger the code, taller the testing task (2000 + test cases)
- Lack of Y-Axis (functional) / Z-Axis (Data partitioning)/X-Axis (individual services) scaling in practical sense
- Longer time to debug /isolate & retest (single fix and run whole regression suite)
- Long compile time (include JUnit) and deploy time (CI/CD as well)
- High percentage of code overriding (cause less productive)
- Longer time to debug the issues and isolate the same
- Heavy weight deployable artifacts
- Bigger team Vs Agile team philosophy

Microservices Usage in CT projects – Features (3/4)



Microservices Usage in CT projects – Features (4/4)

Split of single huge code base into 4 microservice/projects (600 files each)

Smaller the code, lesser the test cases (500+ test cases)

Easy and quick defect fixing/isolating & testing

Shorter compile time (include JUnit) and deploy time (CI/CD as well)

Lesser percentage of code overriding due to multiple teams

Application breakdown in relatively smaller modules

Horizontal scaling by deploying microservice/modules in different container on different nodes

Light-weight deployable artifacts

Smaller team and Agile team philosophy

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com