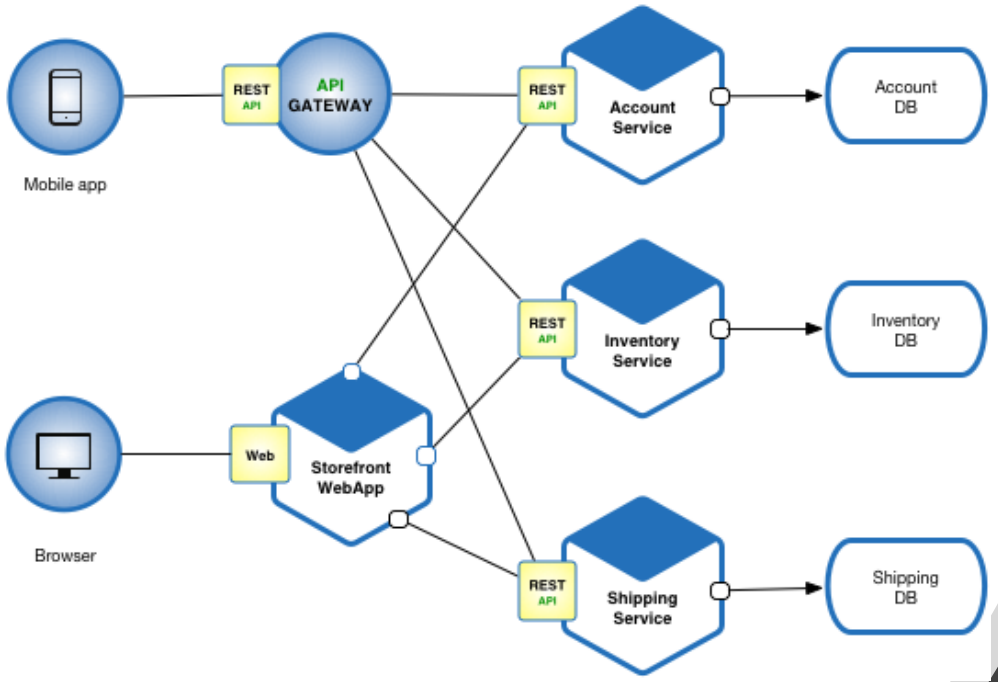
Microservices
What, how, when

- Pratik Das

Outline

- Defining the Problem – the Monolith
- Microservice Architecture – the solution
- Microservice Anatomy
- Coding options- RESTful, Event driven, FAAS (Serverless)
- Communication thru Events - Kafka
- Deployment Options –
  - Containers- Docker
  - Clusters- Kubernetes

What is MSA

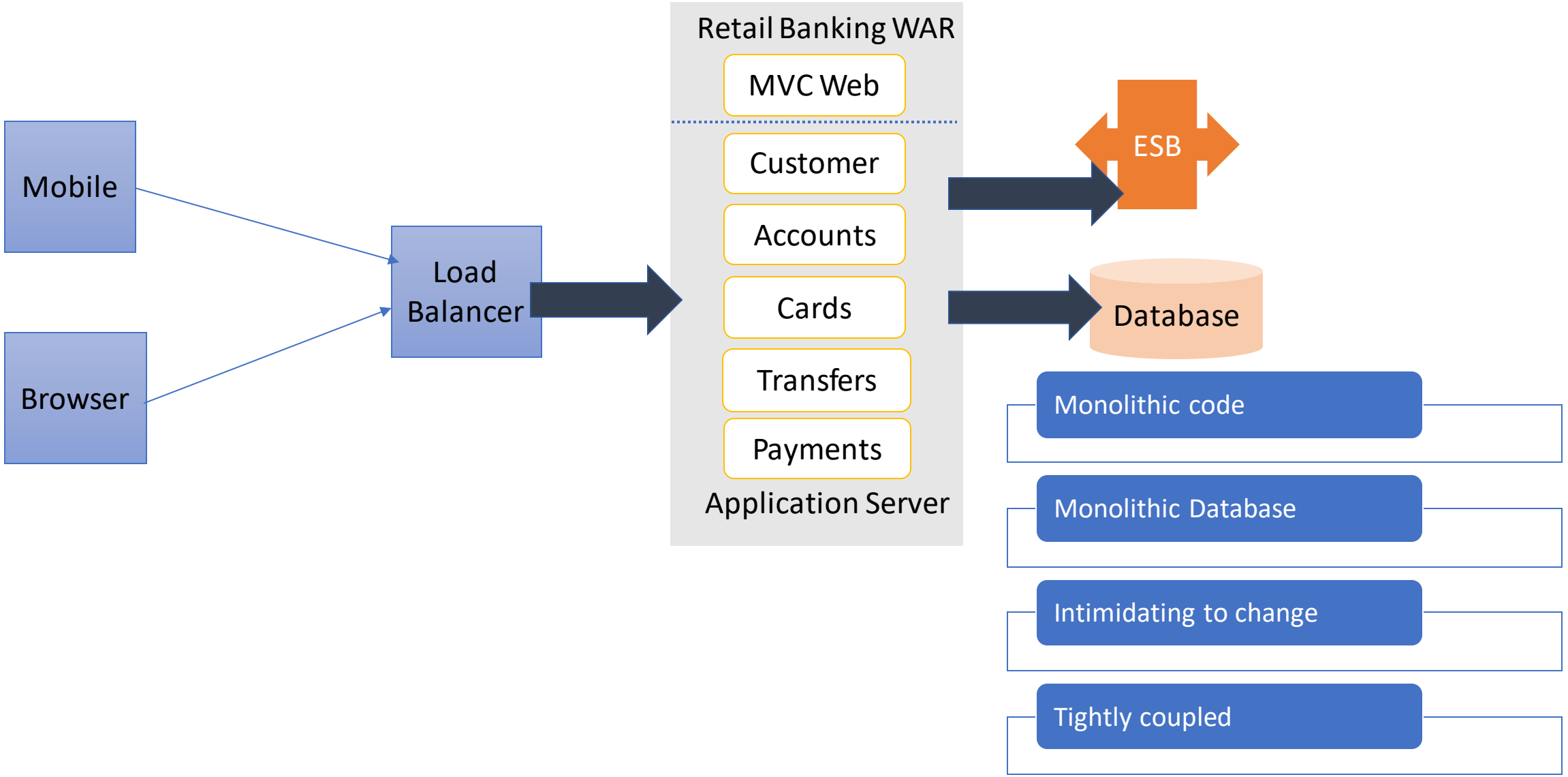


Architectural style that structures an application as a collection of services that are

- Autonomous
- Easily maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities.

Monolith the problem?

Traditional N-tier Applications



Microservice-the solution

Microservice Architecture Principles

Single Purpose- Organized around business capabilities.

Decentralized/Autonomous Teams

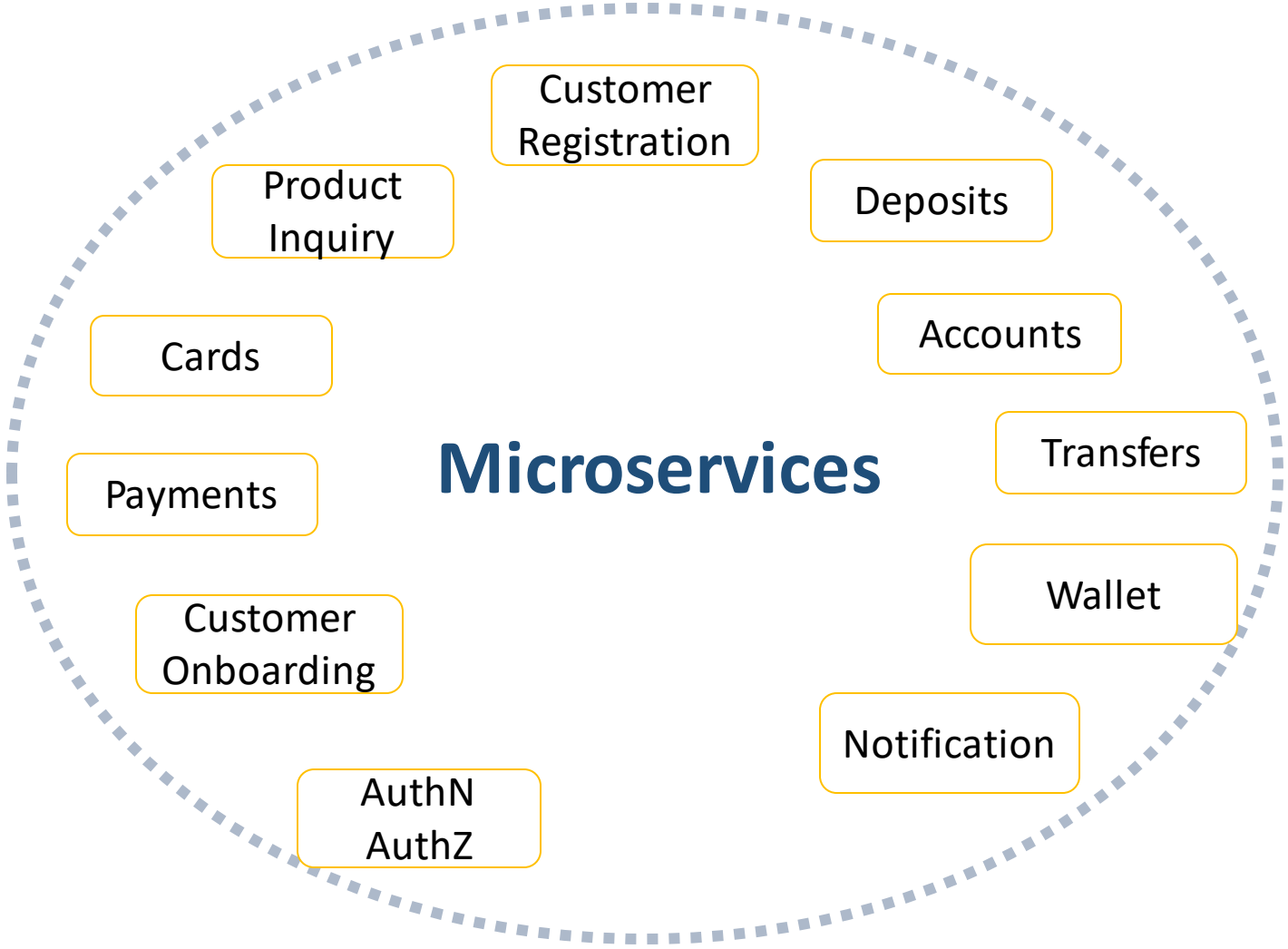
Independently deployable

Share nothing(code,data,functionality)

Isolate Failure

Culture of automation

Smart endpoints dumb pipes



Breaking up a Monolith

Monolith – pros and cons

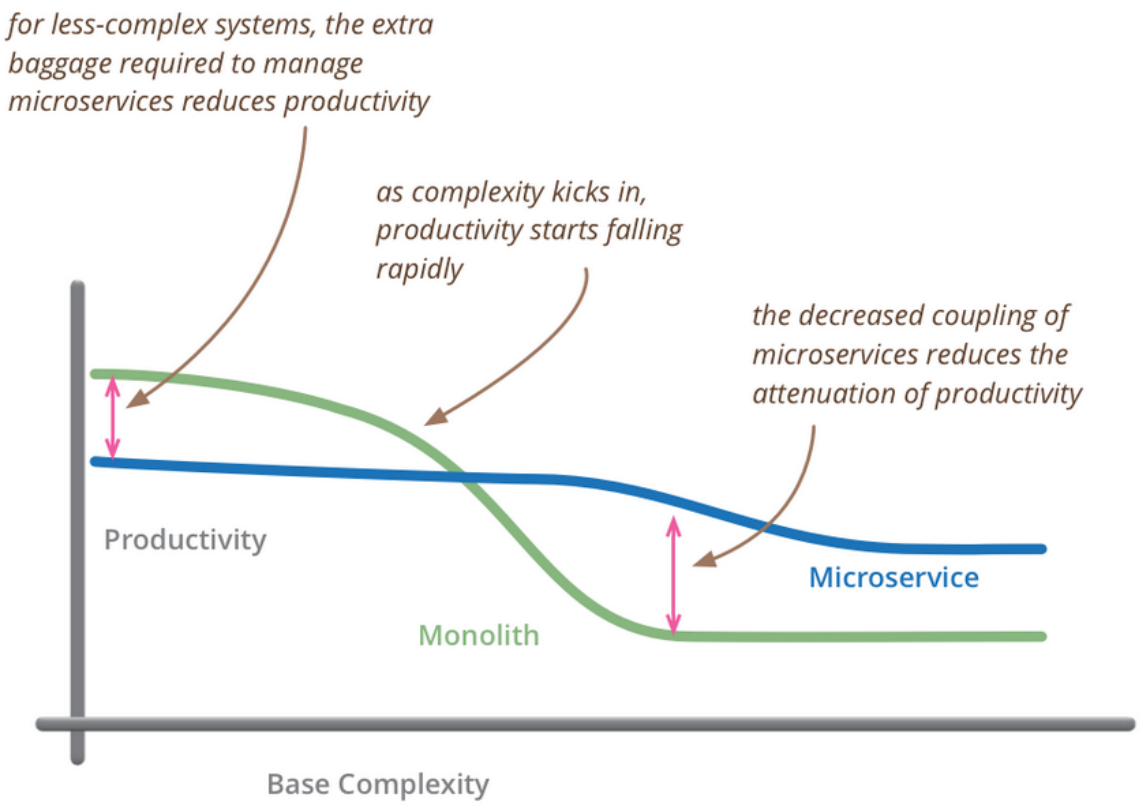
Pros

- Simple
- Easy to deploy
- Test
- Scale

Cons

- Change is Intimidating
- Discourages Frequent Deployments
- Overloads IDE
- High deployment times
- Conflicting scaling requirements of modules
- Constrains update of Technology Stack

When to break a Monolith



but remember the skill of the team will outweigh any monolith/microservice choice

Decomposition Techniques

- Single Responsibility Principle
- Bounded Context-DDD
- Ensure autonomous development and deployment
- Size is linked to autonomous delivery of business capability
- Should have cohesive operations/functionalities
- Start with broad service boundaries. Decompose to smaller ones based on business requirements.
- Apply Strangler pattern for breaking a monolith
- Strive to achieve DRY but be liberal to embrace WET

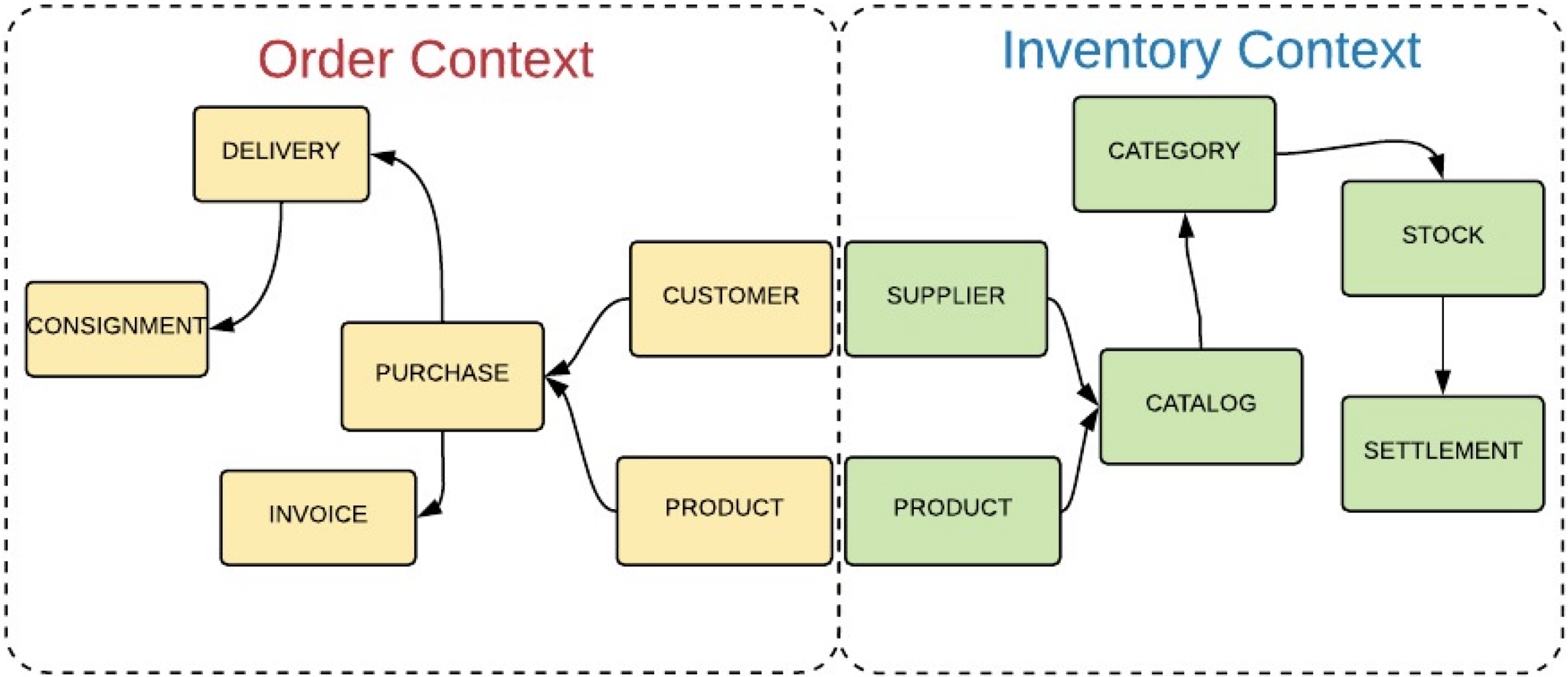
Splittting steps

- Identify service boundaries
- Refactor and split the database

Clues to Identify Service Boundaries

- Identify service boundaries(namespace, packages) and refactor
- Feature based
- Team location
- Security/transactions vs Inquiry
- Technology fitment

Functional Decomposition using DDD - Bounded Context



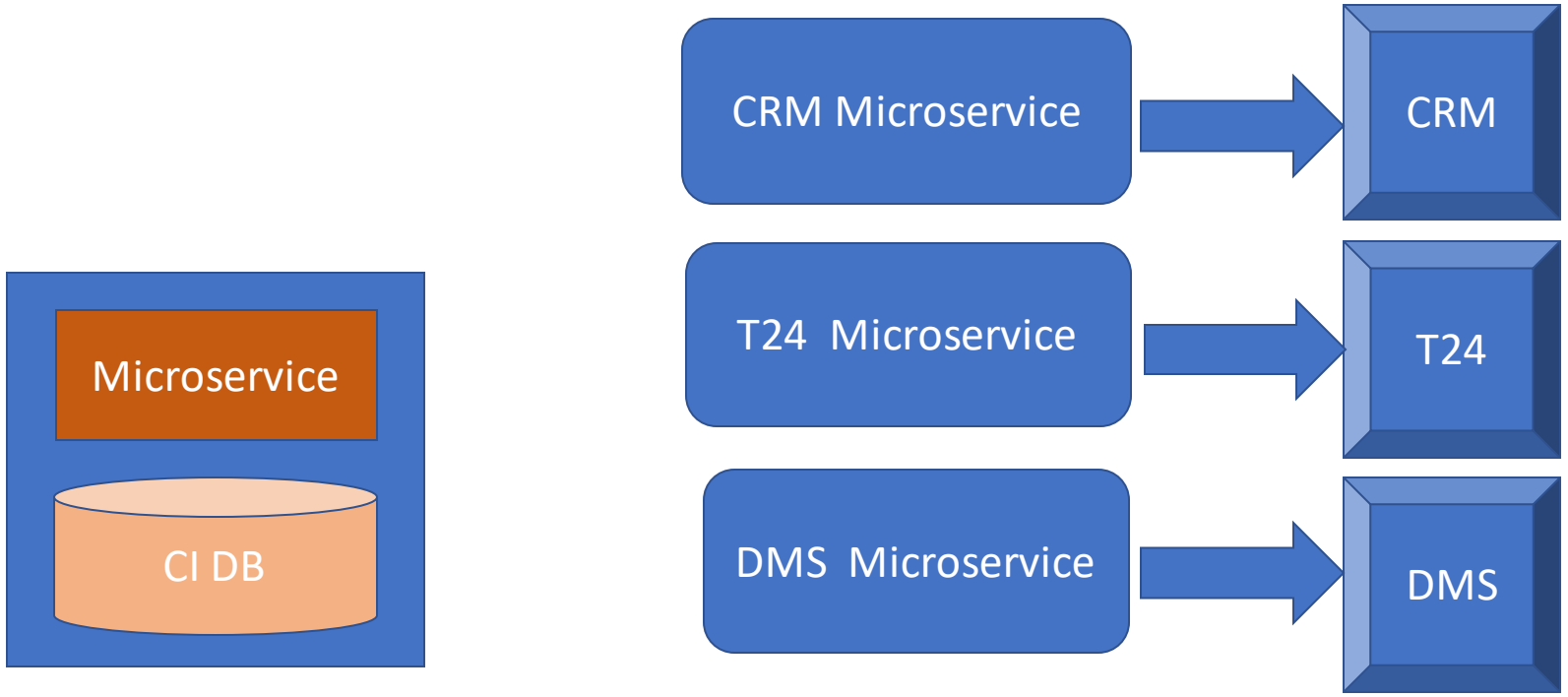
Refactoring databases

- Splitting the database
- Break down Foreign key relationships
- Handle Shared static data (Config files, separate service, duplicate table)
- Extract Shared data modelled in database in separate domain
- Handle Shared table
- Manage transactions
- How to handle Reporting

Integration with legacy and vendor systems

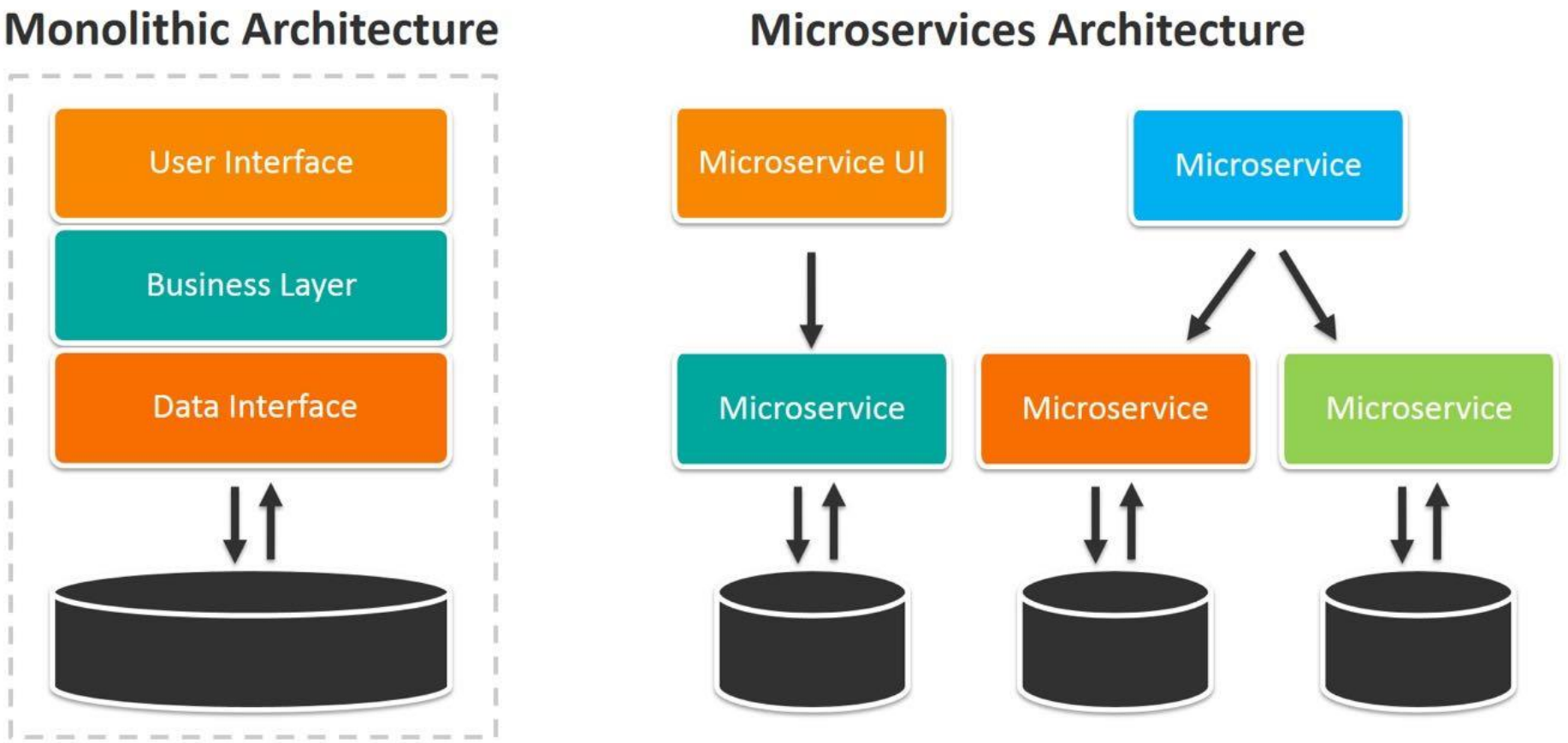
Integration with legacy and vendor systems

- Modelling legacy Application and Datastore as microservice
- Modelling Vendor Systems as Microservice



Data Management

Alternate Data Management to monoliths



Decentralized/Database per service

- Loose coupling == encapsulated data
- Each microservice will have its own private database to persist the data.
- A given microservice can only access the dedicated private database but not the databases of other microservices.
- Updating Database of other microservices if required should be done through API
- Extract Shared database to a separate Microservice

Shared Database

Problem: Monolithic database makes it easy to leverage shared data. How do we use this in Microservices?

Principle: Single System of Record

- Every piece of data is used by 1 service
- Every other data is read-only copy

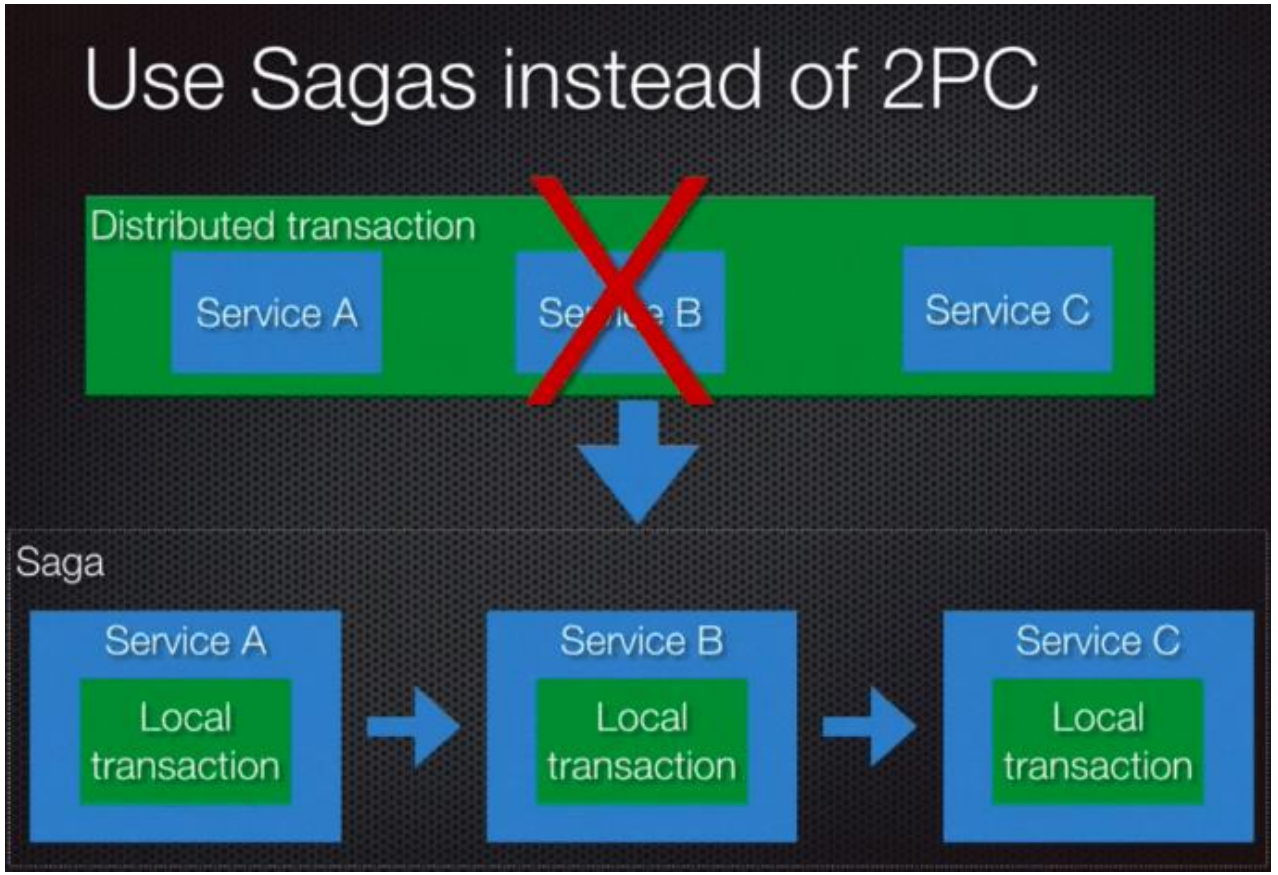
Manage Transaction- SAGAS

- 2 PC is not an option
- CAP theorem – impacts availability
- Reduced throughput due to locks
- Single point of failure
- Use sagas to maintain data consistency across services
- Use transactional messaging to make sagas reliable

Saga

Successful Saga

- Start Saga
- Start T1
- End T1
- Start T2
- End T2
- Start T3
- End T3
- End Saga

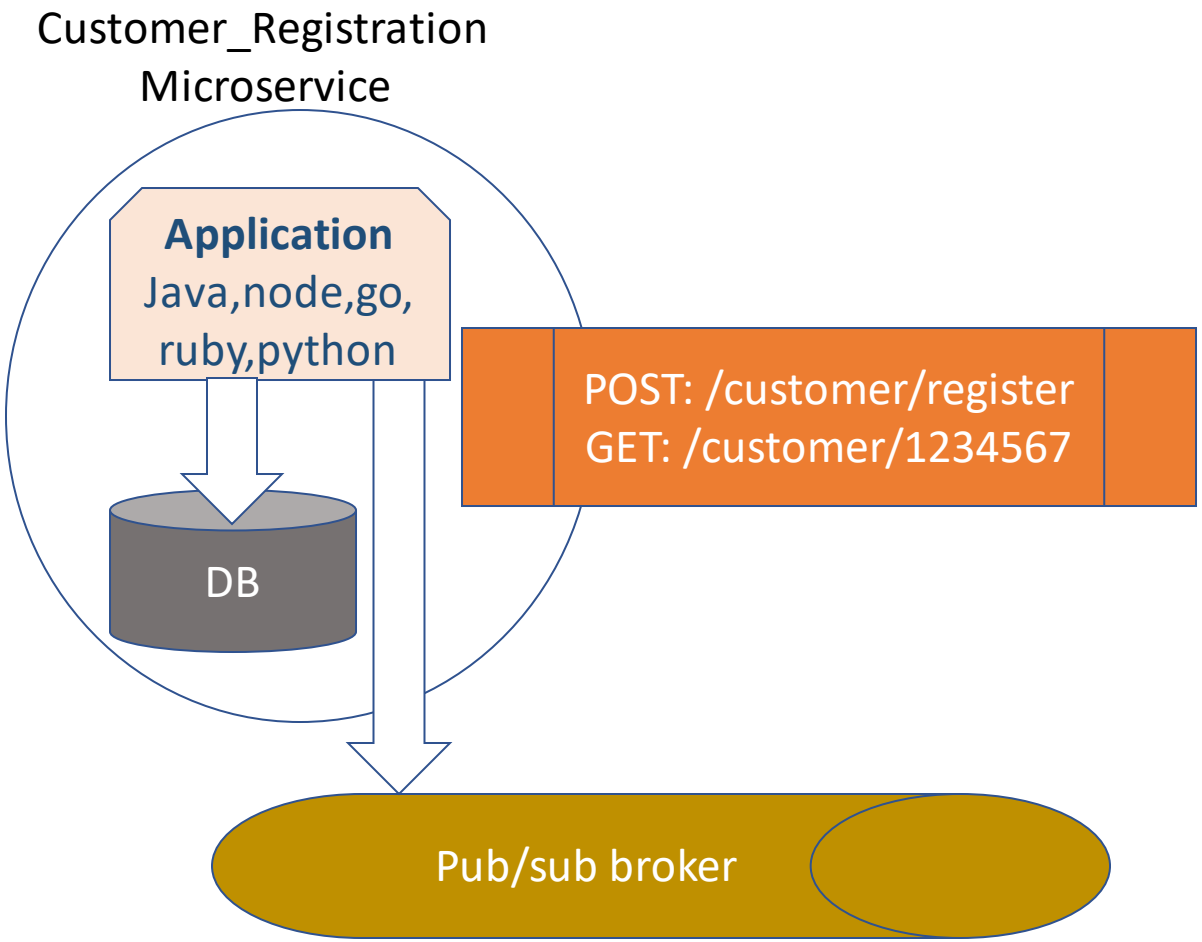


Failed Saga

- Begin Saga
- Start T1
- End T1
- Start T2
- Abort Saga
- Start C2
- End C2
- Start C1
- End C1
- End Saga

Microservice Anatomy

Web Application- read from db, publish event



Coding a Microservice

Dev frameworks-traits

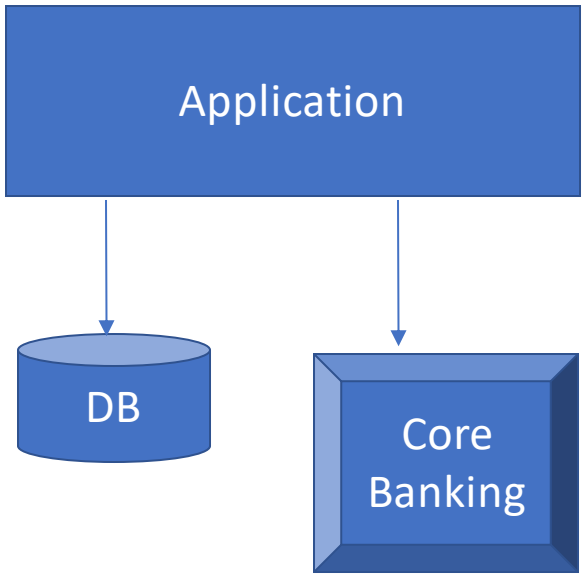
- Service Discovery
- Service registry
- Load Balancer
- Synchronous communication
- Asynchronous communication
- Resilience-circuit breakers, timeout, retry, fallback

How to design Microservice

Example – Retail Banking

Application features

- Customer Registration
- Product Inquiry
- Transfer Funds



What is a Good Service

- Loose coupling – Do not share internal details
- High Cohesion- Related behaviour sits together

Decomposition – using DDD bounded context

- Data is kept in silos, or Bounded Contexts
- All bounded context are treated as 1 big Application
- Only 1 Bounded Context is responsible for updating 1 datastore
- Handle Stale data by subscribing to changes in data of interest
- Embrace Eventual Consistency
- Cache only data of interest

Languages and Frameworks

- Java: Jakarta EE, Spring boot, Spring Cloud, Microprofile, Micronaut
- NodeJS: Express, Moleculer
- Golang: Go Micro, Go Kit, Gizmo, Kite
- Ruby: Sinatra
- Python: Django
- .Net

Spring Boot

- Opinionated Framework
- Convention over configuration
- Starter packs- Automatic config with defaults
- Basic REST App components: Resource, Service, Repository

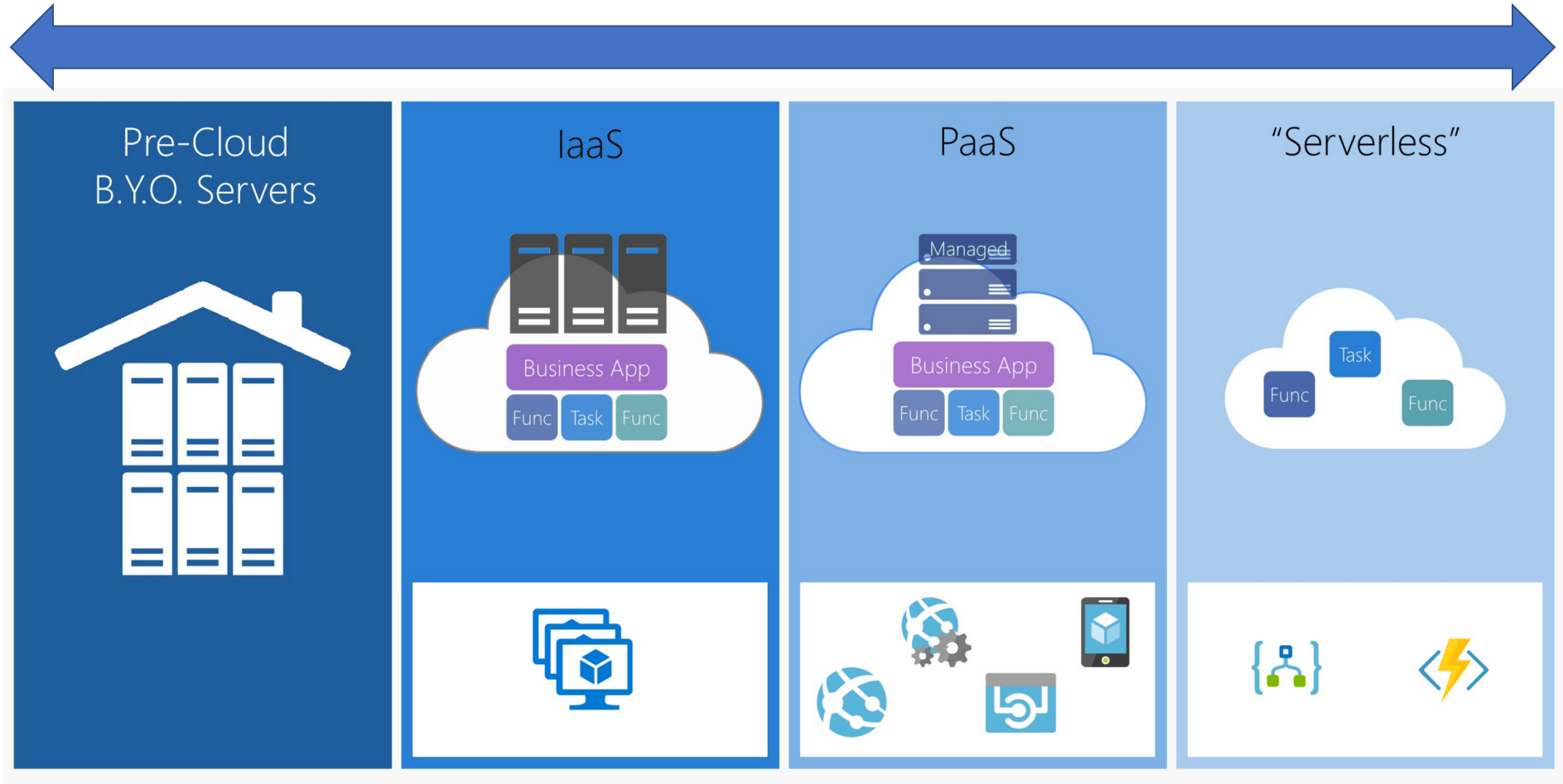
Spring boot example

- CustomerInquiry Microservice

➢ Components:

- ❖ REST endpoint : /customer/{id}
- ❖ CustomerResource
- ❖ CustomerInquiryService
- ❖ CustomerRepository
- ❖ Spring Application

B.Y.O, IAAS, PAAS, FAAS



Takeaways

Takeaways

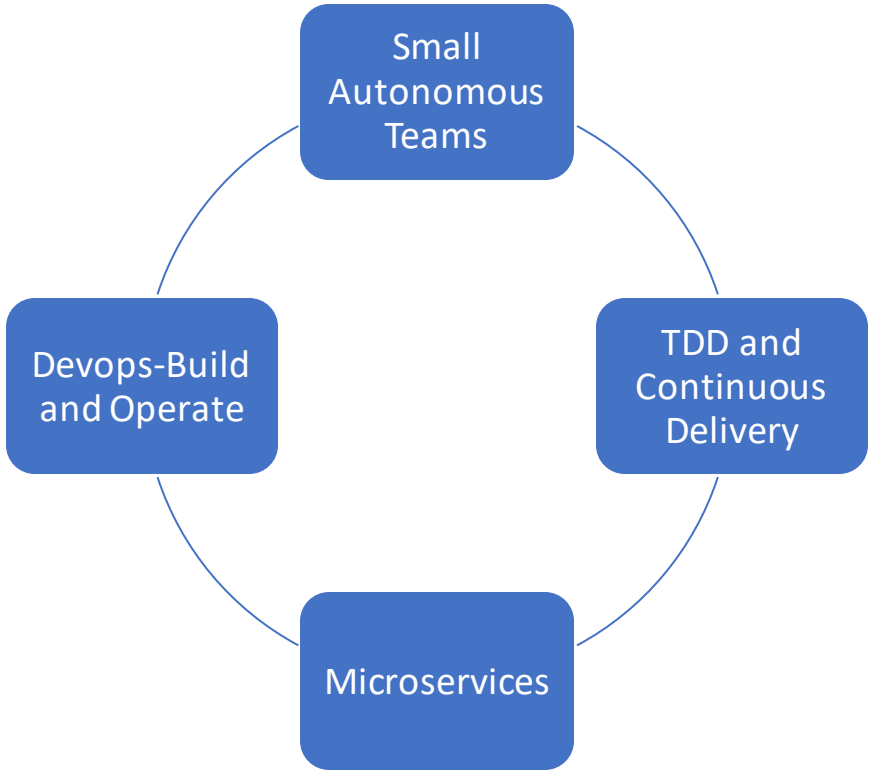
- Monolith is not an antipattern and good for small applications
- Implementing Microservices is complex but has many benefits
- Essentially about functional decomposition
- Use DDD bounded context to identify Microservices
- Database and config are part of the service
- Microservice Anatomy- RESTful apps or FAAS/serverless
- Multiple options for languages and frameworks

Challenges and Solutions

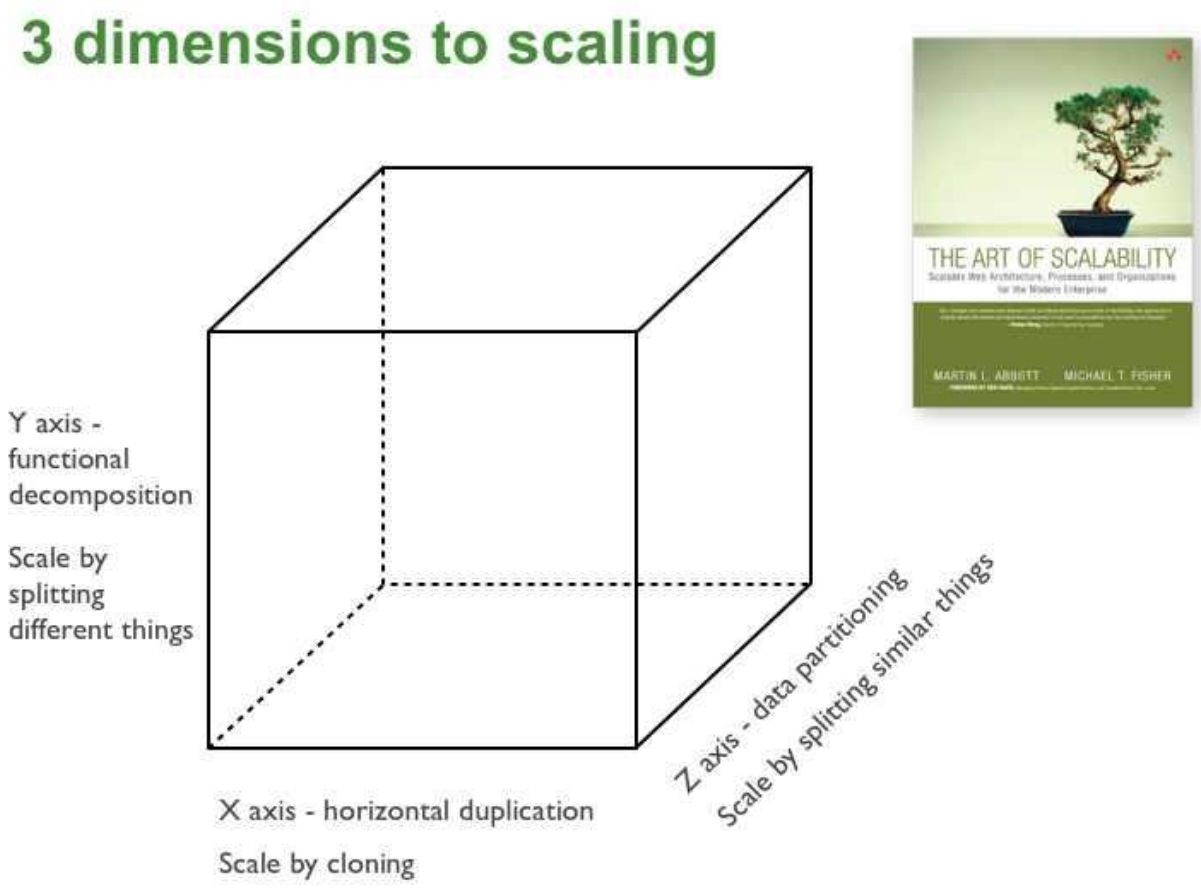
Challenges

- Dependency-Inter service request-Hystrix test with fault injection testing
- Scale
- Variance
- Change

Microservices-supported by Org processes



Achieve Infinite Scaling - Scale Cube



Microservice == Functional Decomposition

Integration

Interface design-Principles

- Avoid breaking changes
- Technology and tool agnostic
- Simple consumer friendly
- Hide internal details (separate models for interface and internal implementations)

Integration

- Shared Database
- Request-response
  - RPC: SOAP, gRpc, Thrift
  - REST:
- Event based
- Prefer Choreography over orchestration

Communication

- Request-Response- REST, Thrift, gRPC, Avro

1. Point to point
2. API Gateway
3. Service Mesh

- Asynchronous

1. Dumb Pipes and smart endpoints