

Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: [@vinicius3w](https://twitter.com/vinicius3w) :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
<https://github.com/IF1007/if1007>

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-
Compartilhagual 3.0 Não Adaptada

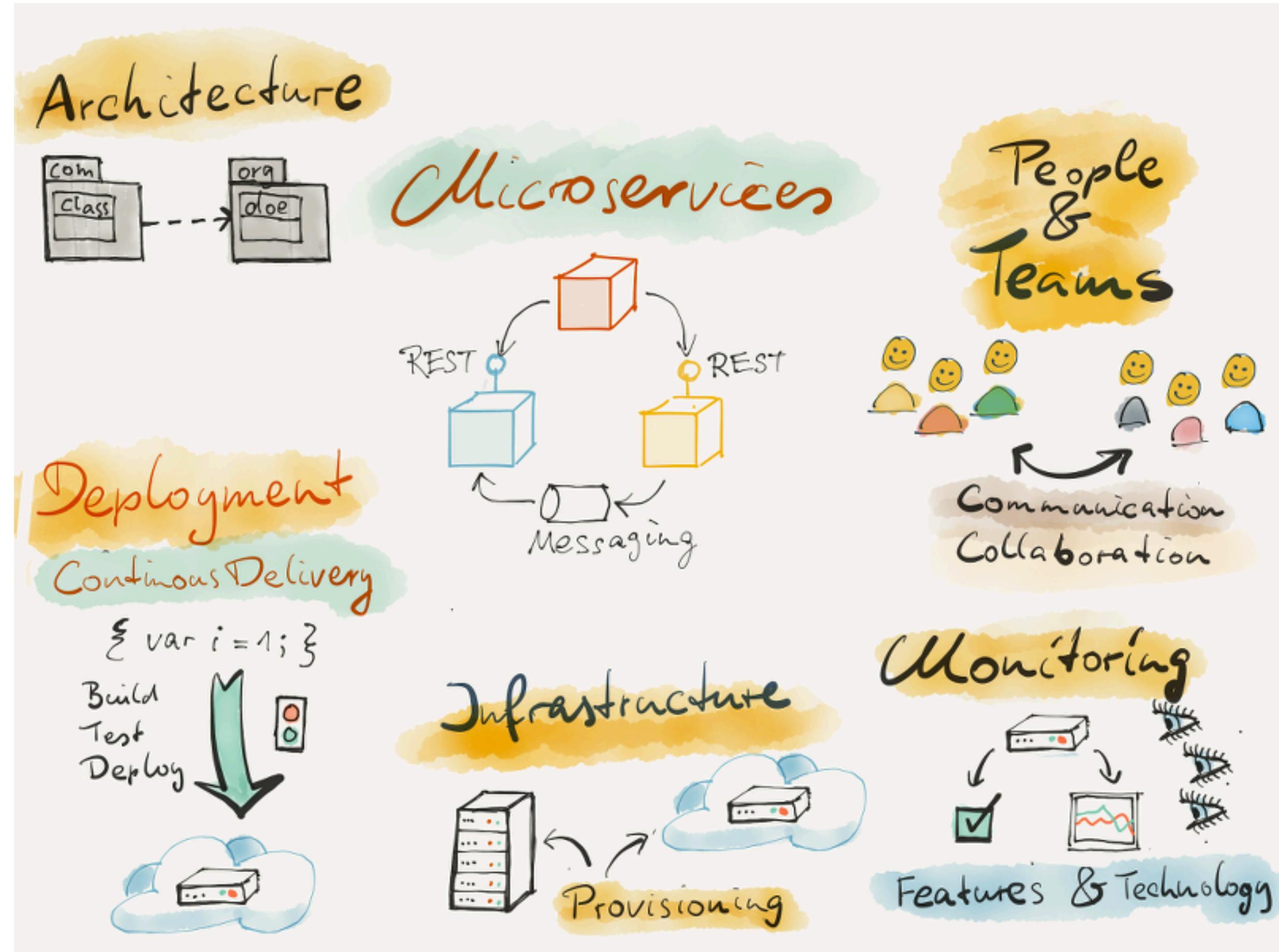


Mais informações visite

<http://creativecommons.org/licenses/by-nc-sa/3.0/>
deed.pt

Resources

- There is no textbook required. However, the following are some books that may be recommended:
 - [Building Microservices: Designing Fine-Grained Systems](#)
 - [Spring Microservices](#)
 - [Spring Boot: Acelere o desenvolvimento de microserviços](#)
 - [Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers](#)
 - [Migrating to Cloud-Native Application Architectures](#)
 - [Continuous Integration](#)
 - [Getting started guides from spring.io](#)



Applying Microservices Concepts

Context

- Microservices are **good**, but can also be an **evil** if they are not properly conceived.
- **Wrong** microservice interpretations could lead to **irrecoverable** failures
- What are the technical **challenges** around **practical** implementations of microservices?
 - design decisions, solutions and patterns?

Patterns and common design decisions

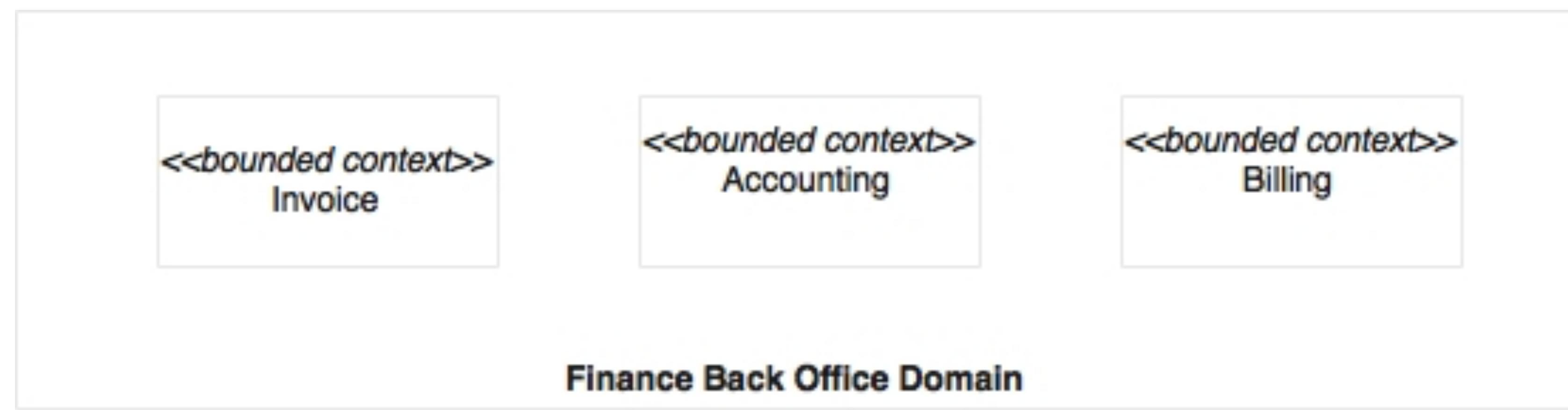
- Microservices are a vehicle for developing **scalable** cloud native systems, successful microservices need to be **carefully** designed to **avoid** catastrophes.
- Microservices **are not** the **one-size-fits-all**, universal solution for **all architecture** problems
- Generally speaking, microservices are a great choice for building a **lightweight**, **modular**, **scalable**, and **distributed system of systems**

Establishing appropriate microservice boundaries

- How **big** (mini-monolithic) or how **small** (nano service) can a microservice be, or is there anything like **right-sized services**?
 - Does size really **matter**?
- A quick answer could be
 - "one REST endpoint per microservice", or
 - "less than 300 lines of code", or
 - “a component that performs a single responsibility”

Establishing appropriate microservice boundaries

- Domain-driven design (DDD) defines the concept of a **bounded context**. A bounded context is a subdomain or a subsystem of a larger domain or system that is responsible for **performing a particular function**



Establishing appropriate microservice boundaries

- A **bounded context** is a good way to determine the **boundaries** of microservices
 - Each bounded context could be mapped to a **single microservice**
 - In the real world, communication between bounded contexts are typically **less coupled**, and often, **disconnected**

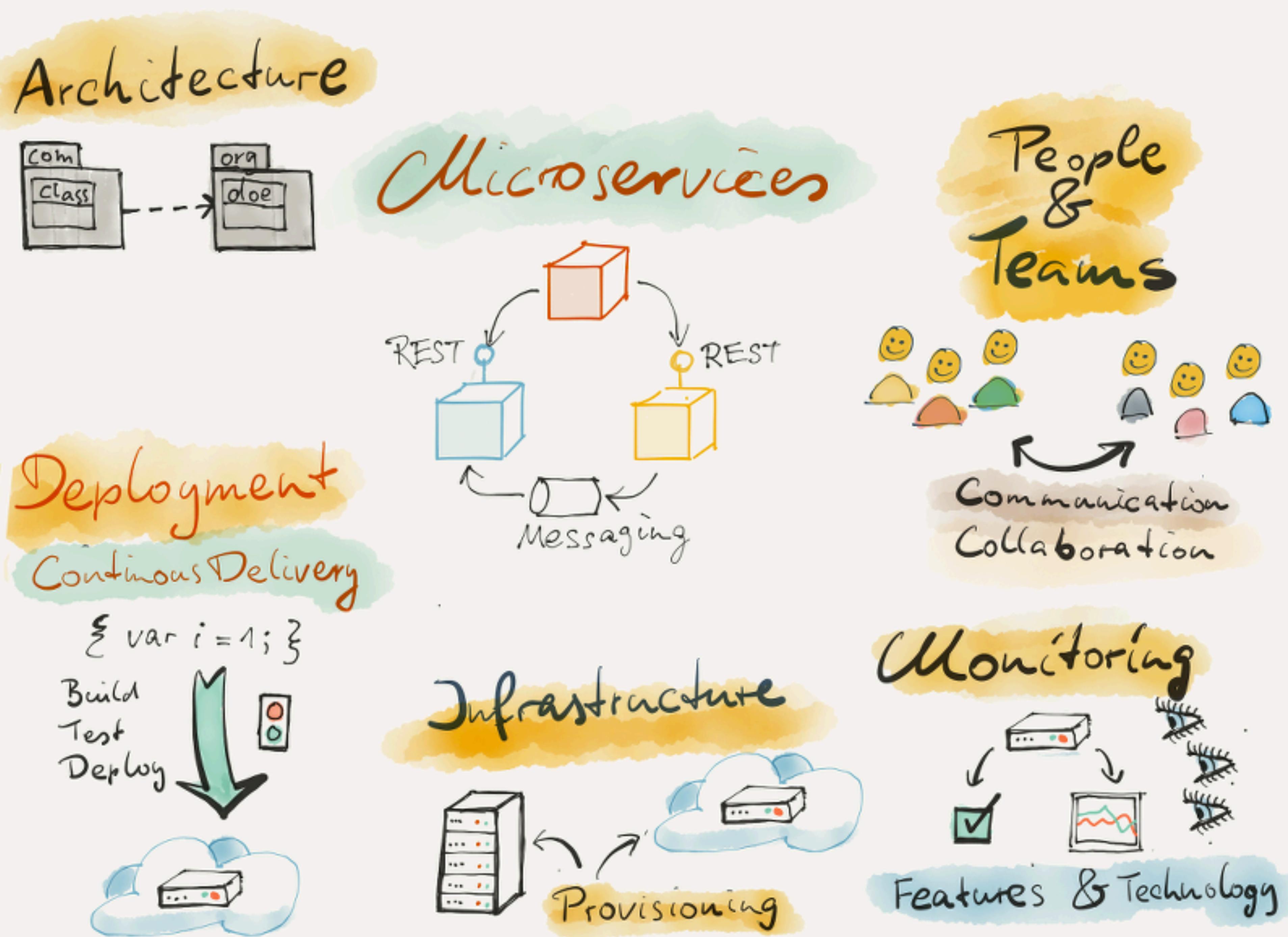
Establishing appropriate microservice boundaries

- There is no **silver bullet** to establish microservices boundaries
 - Establishing boundaries is much easier in the scenario of **monolithic application to microservices migration**, as the service boundaries and dependencies are **known** from the existing system.
 - On the other hand, in a **green field microservices development**, the dependencies are hard to establish upfront.
- The most **pragmatic** way to design microservices boundaries is to run a lot of **scenarios**

Scenarios could help in defining the microservice boundaries

- Autonomous functions
- Size of a deployable unit
- Most appropriate function or subdomain
- Polyglot architecture
- Selective scaling
- Small, agile teams
- Single responsibility (business capability or a technical capability)
- Replicability or changeability
- Coupling and cohesion
- Think microservice as a product

Think microservice as a product



Designing communication styles

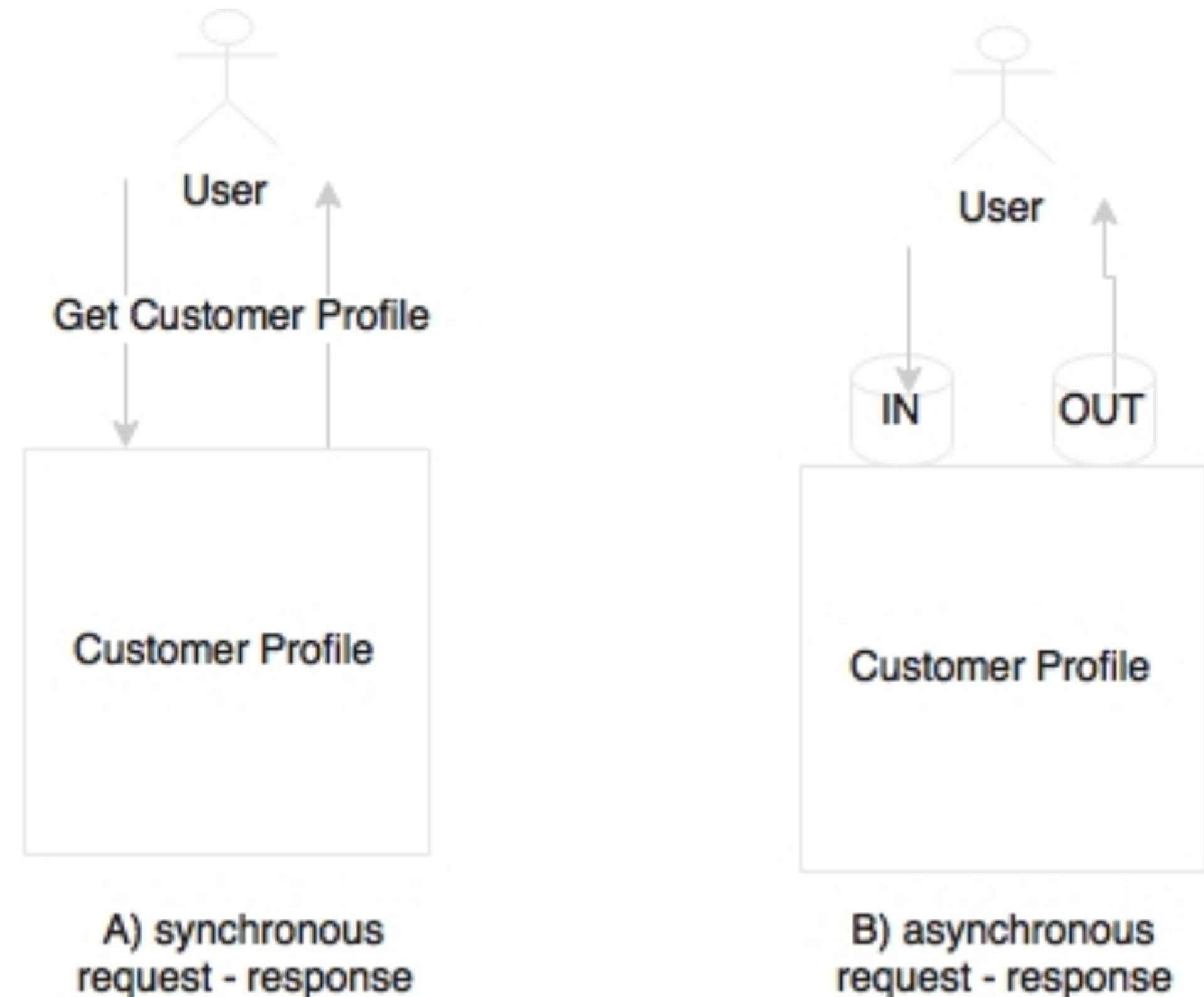
- Synchronous style communication
 - there is no shared state or object
 - When a caller requests a service, it passes the required information and waits for a response
 - Advantages and downsides?

Designing communication styles

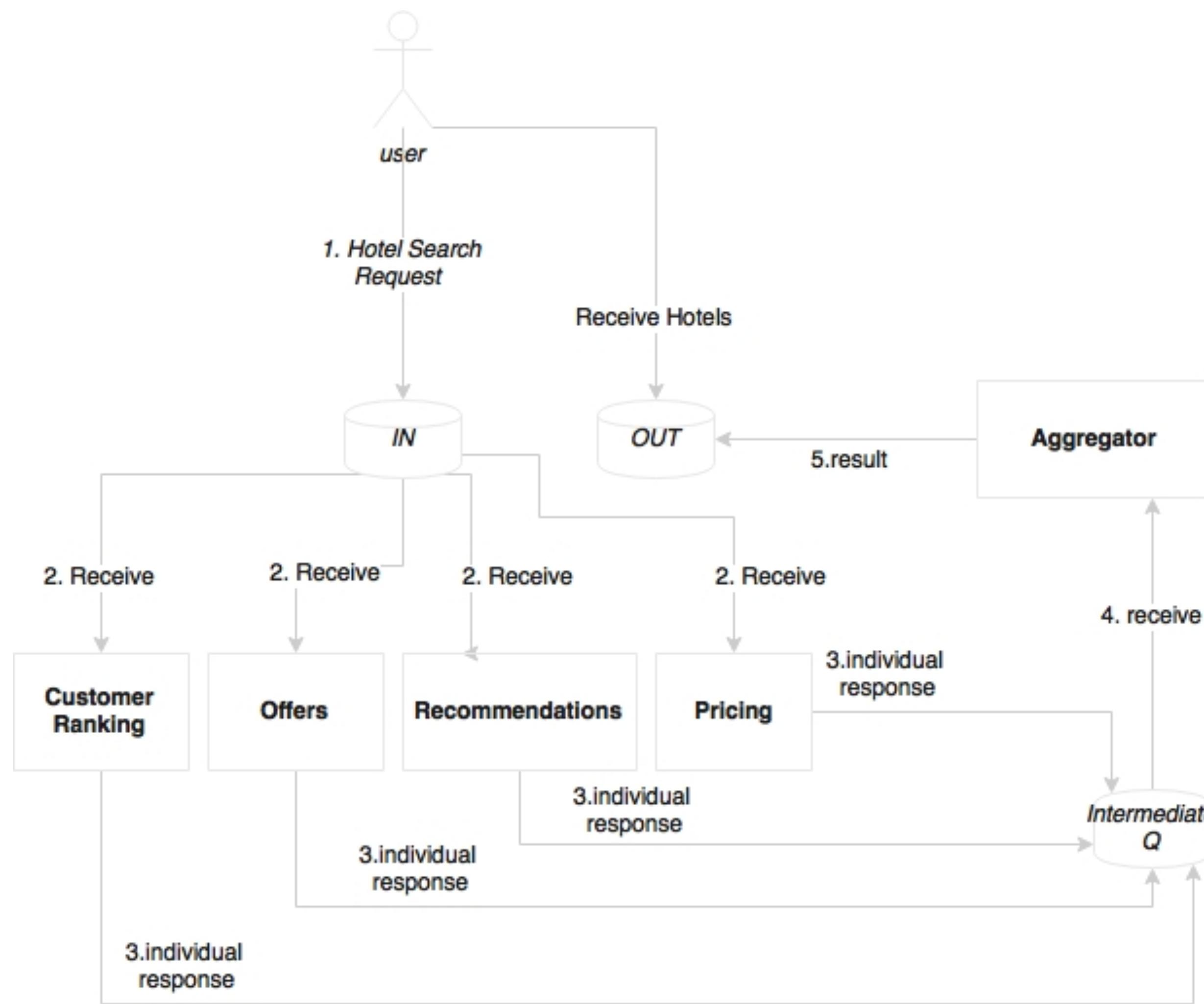
- Asynchronous style communication
 - The asynchronous style is based on **reactive event loop semantics** which decouple microservices.
 - This approach provides higher levels of **scalability**, because services are **independent**, and can internally spawn threads to handle an increase in load.
 - When overloaded, messages will be **queued** in a messaging server for **later processing**

How to decide which style to choose?

- Both approaches have their own merits and constraints
 - In principle, the asynchronous approach is great for building true, scalable microservice systems
 - However, attempting to model everything as asynchronous leads to complex system designs

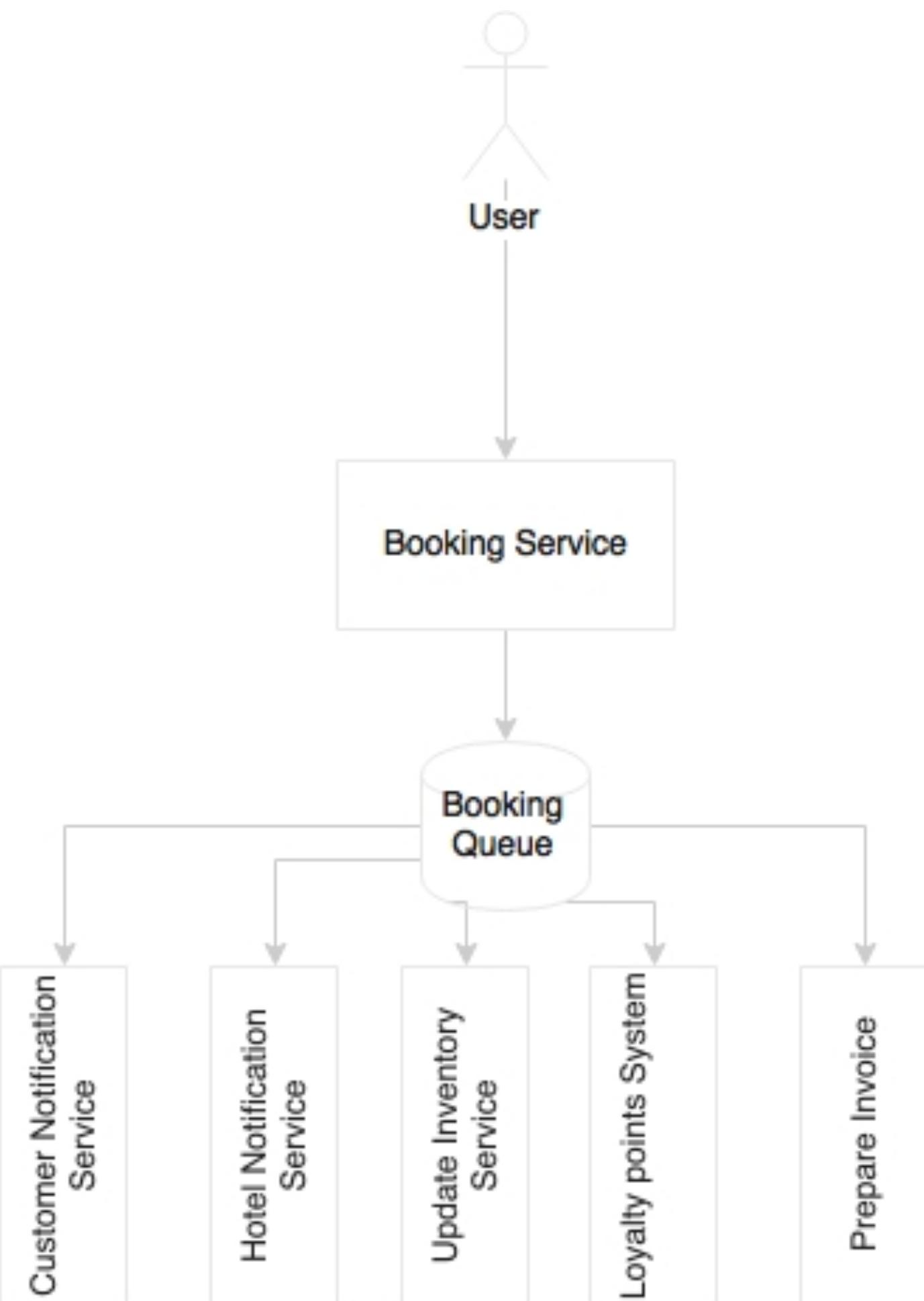


How to decide which style to choose?



How to decide which style to choose?

- The service is triggered when the user clicks on the booking function.
- It is again, by nature, a synchronous style communication.
- When booking is successful, it sends a message to the customer's e-mail address, sends a message to the hotel's booking system, updates the cached inventory, updates the loyalty points system, prepares an invoice, and perhaps more

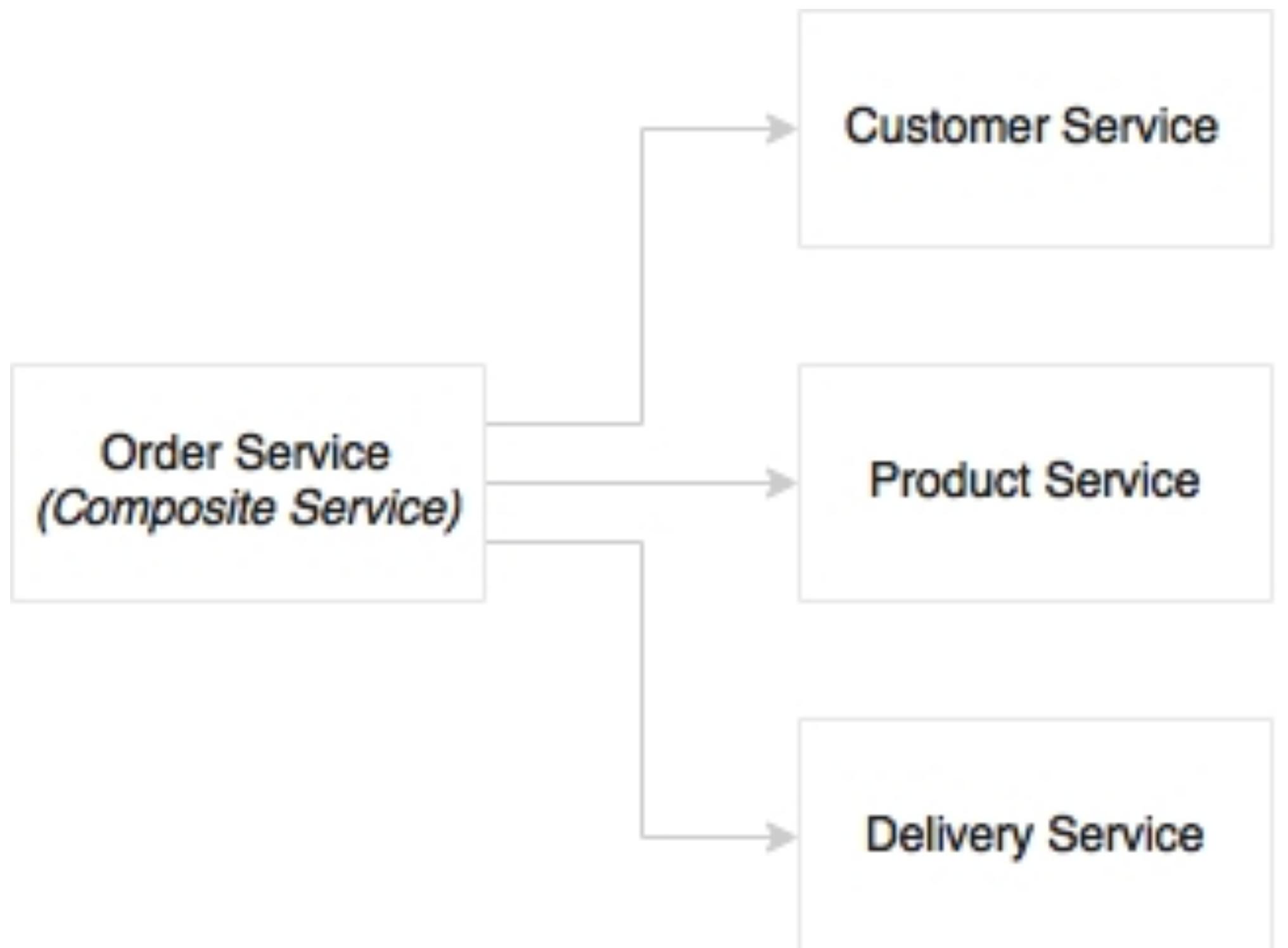


Orchestration of microservices

- Composability is one of the service design principles
 - Who is responsible for the composing services?
- SOA use ESBs (act as a proxy in some cases)
 - The first approach is orchestration

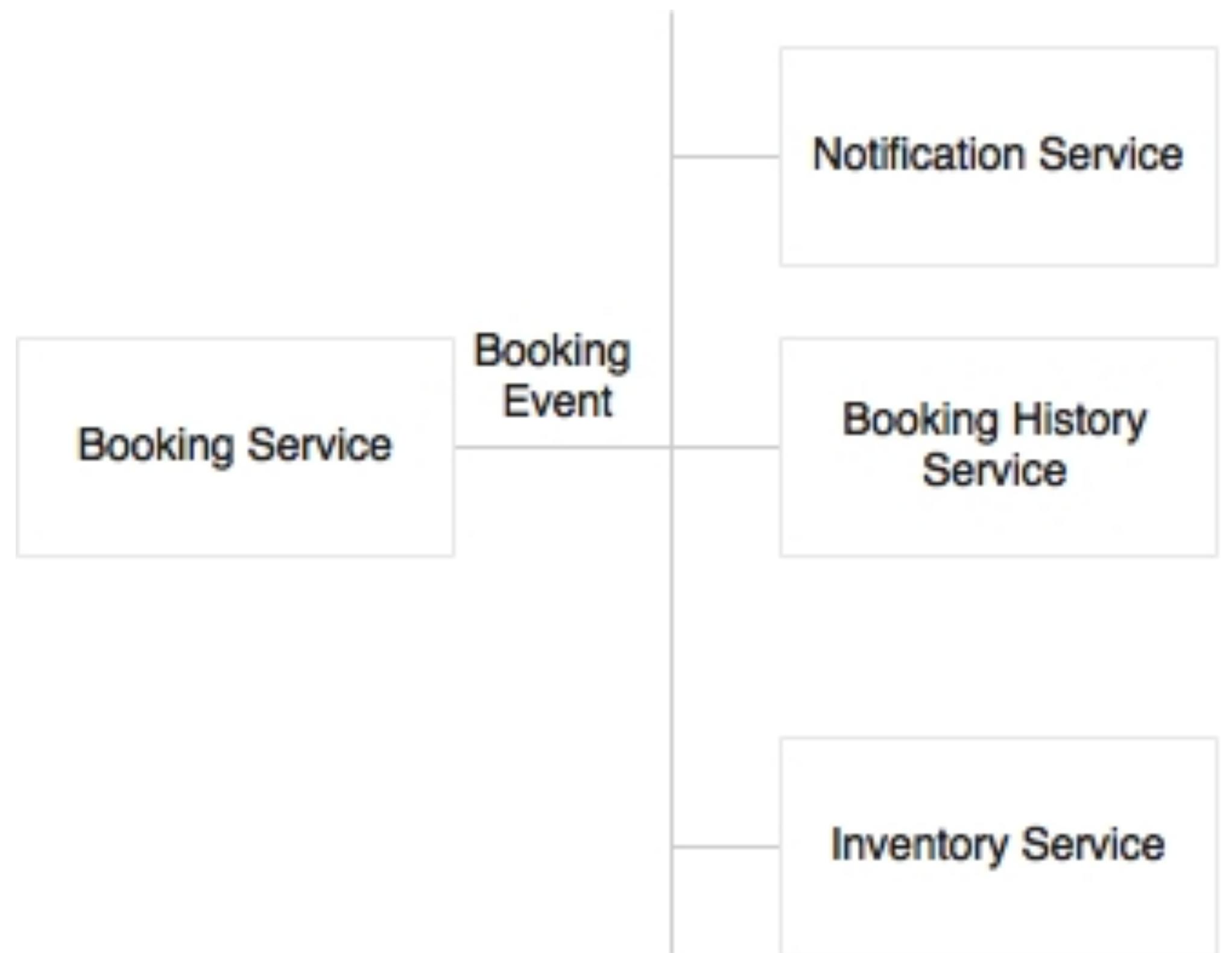
Orchestration of microservices

- Multiple services are **stitched together** to get a **complete function**. A central brain acts as the **orchestrator**
- In the SOA world, ESBs play the role of **orchestration**
- The orchestrated service will be exposed by ESBs as a composite service



Orchestration of microservices

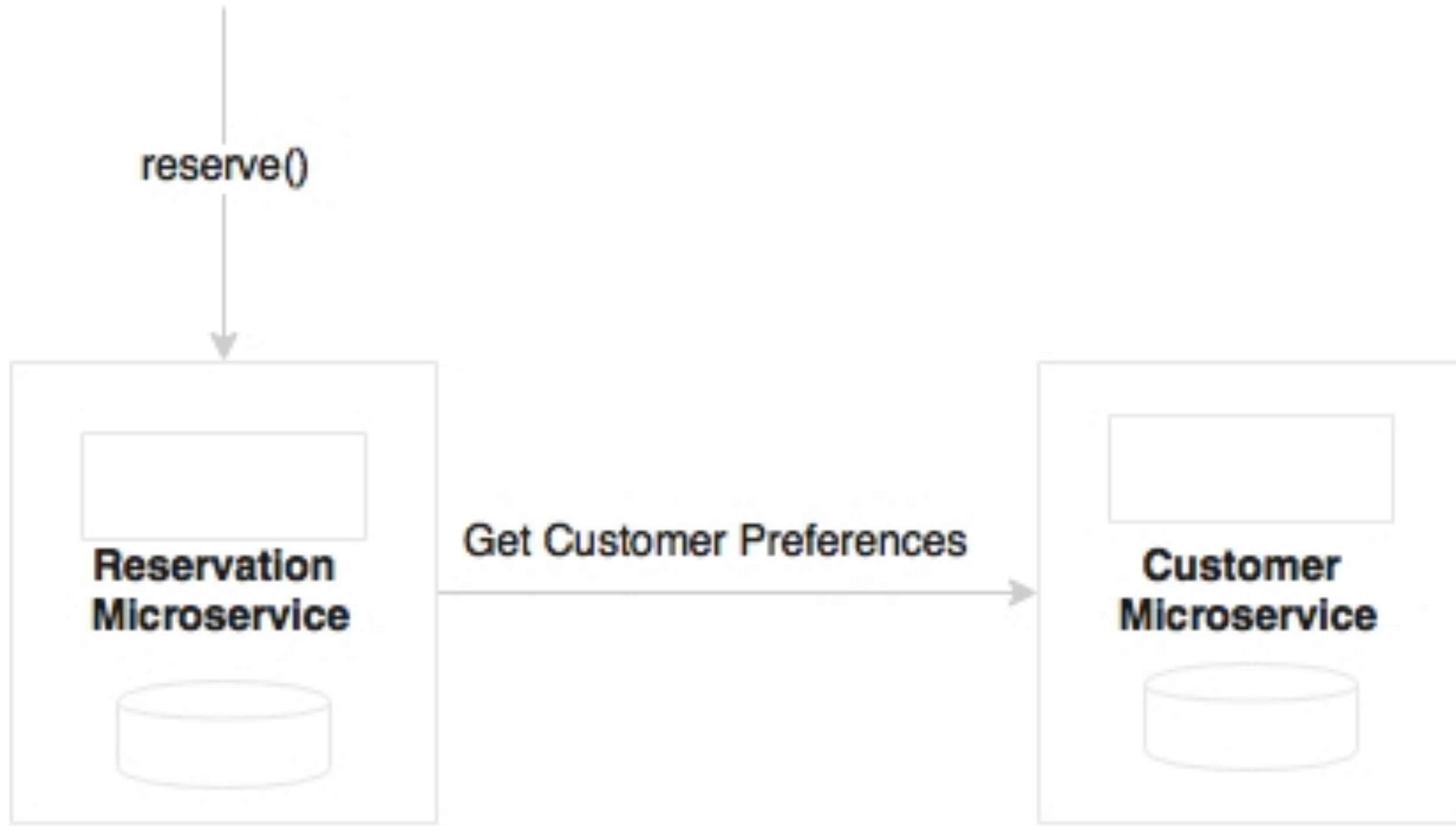
- The second approach is choreography, in where, there is no central brain
- In the SOA world, the caller pushes a message to the ESB, and the downstream flow will be automatically determined by the consuming services



Orchestration of microservices

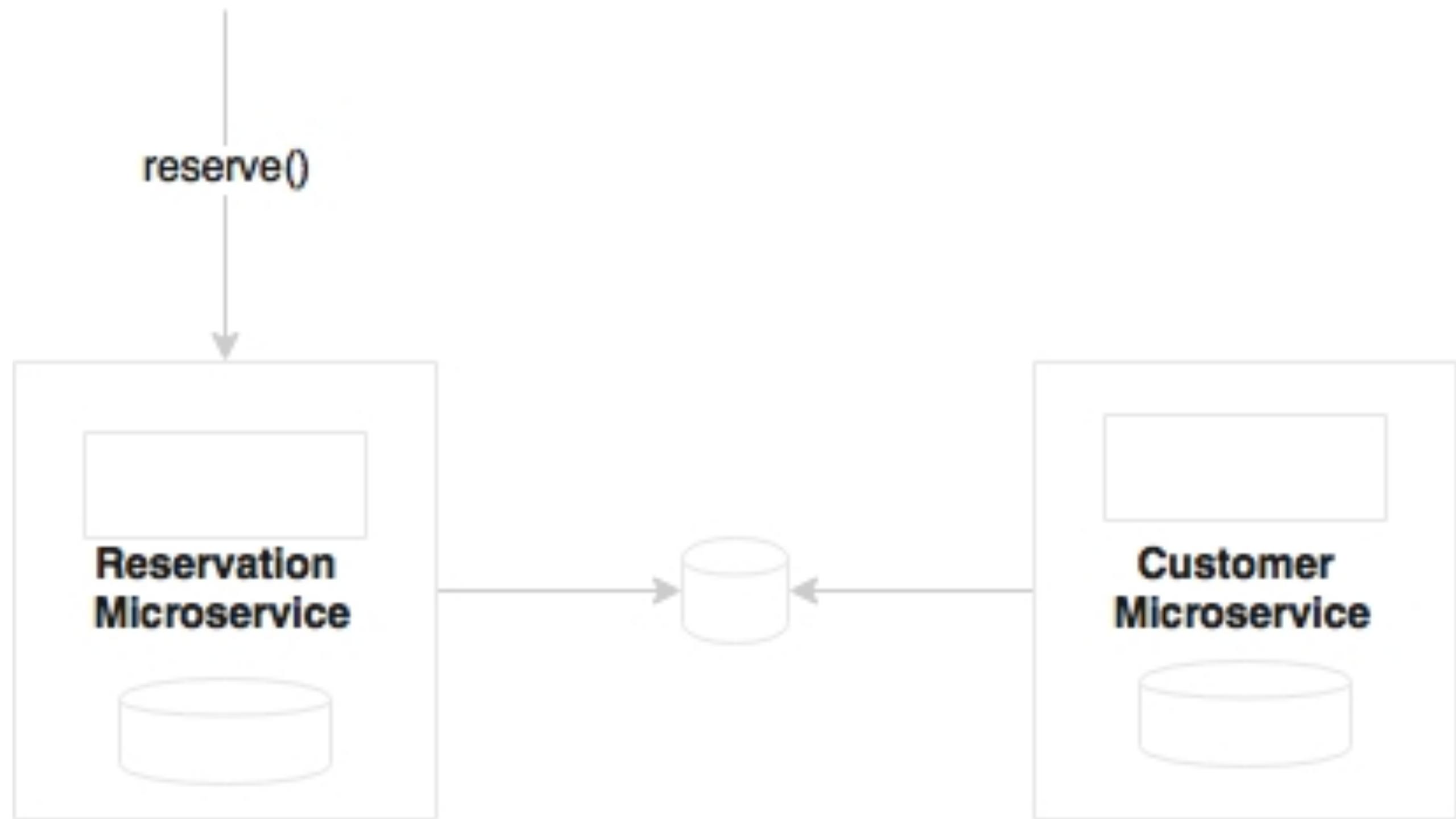
- Microservices are autonomous
 - all required components to complete their function should be within the service
- The service endpoints provide coarse-grained APIs
 - there are no external touch points required
 - microservices may need to talk to other microservices to fulfil their function

We can model choreography in all cases?



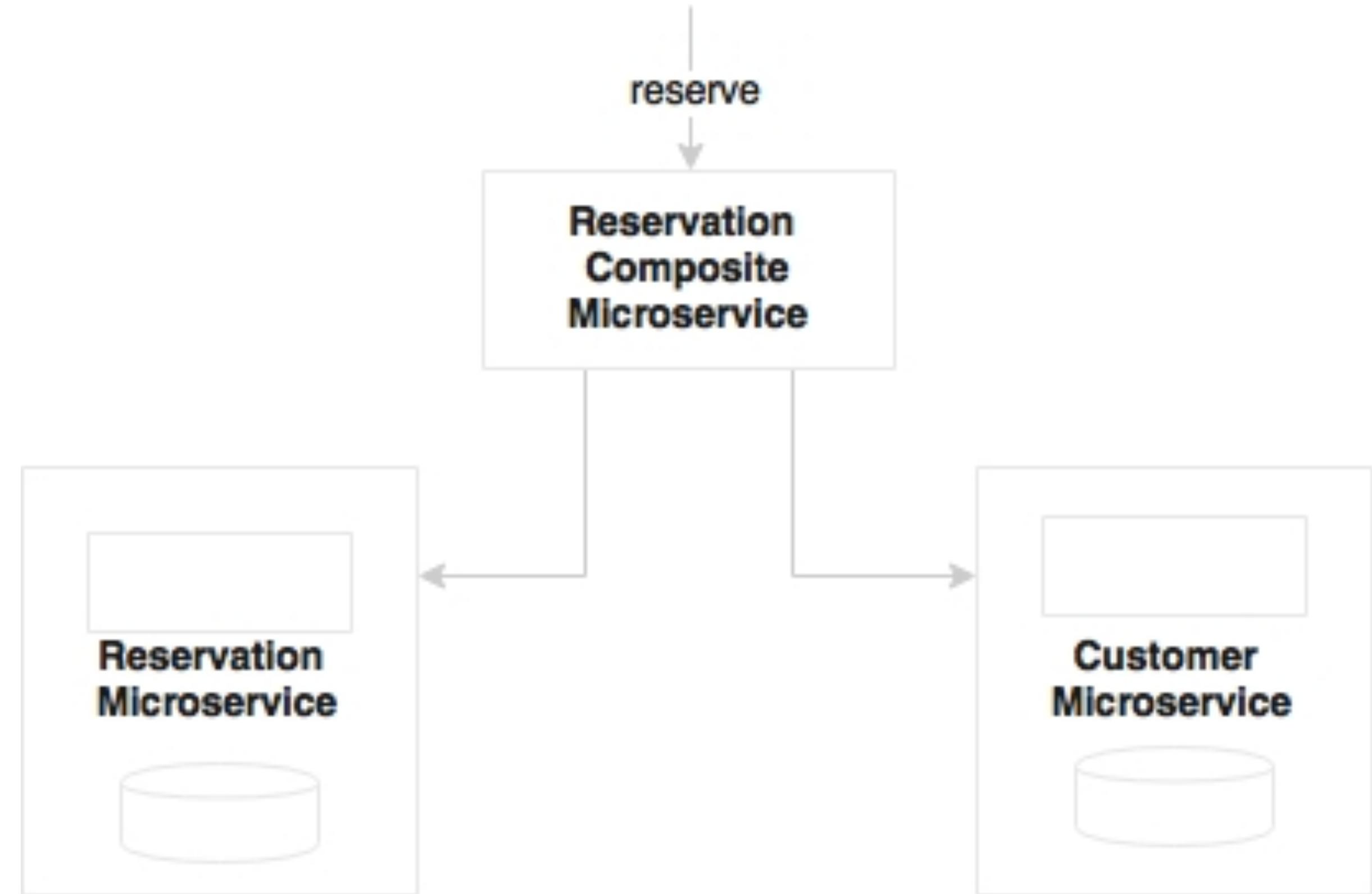
We can model choreography in all cases?

- Customer preference is required for Reservation to progress, and hence, it may require a synchronous blocking call to Customer
- Retrofitting this by modeling asynchronously does not really make sense.



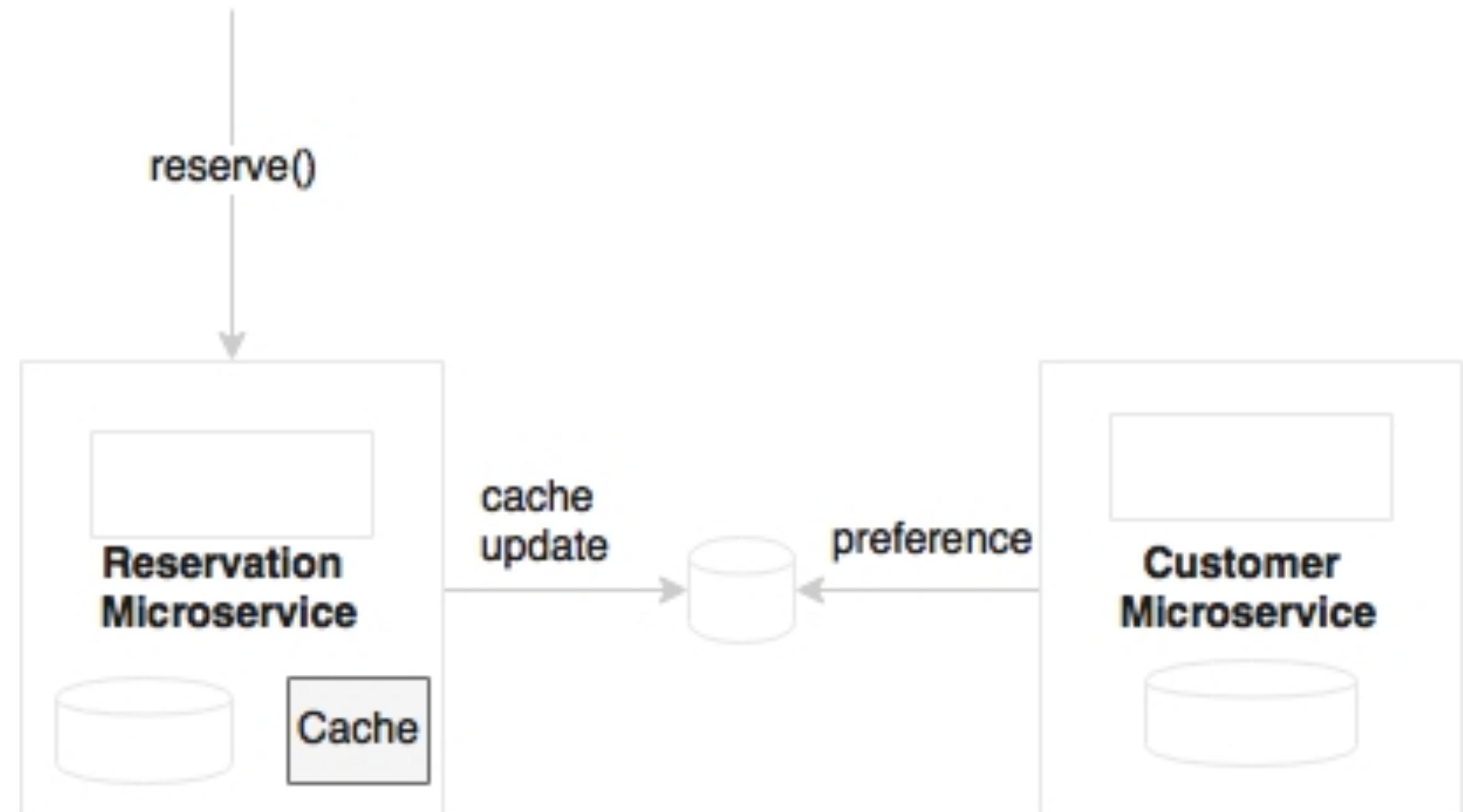
We can model choreography in all cases?

- This is acceptable in the approach for composing multiple components within a microservice
- But creating a composite microservice may not be a good idea.
- We will end up creating many microservices with no business alignment, which would not be autonomous, and could result in many fine-grained microservices



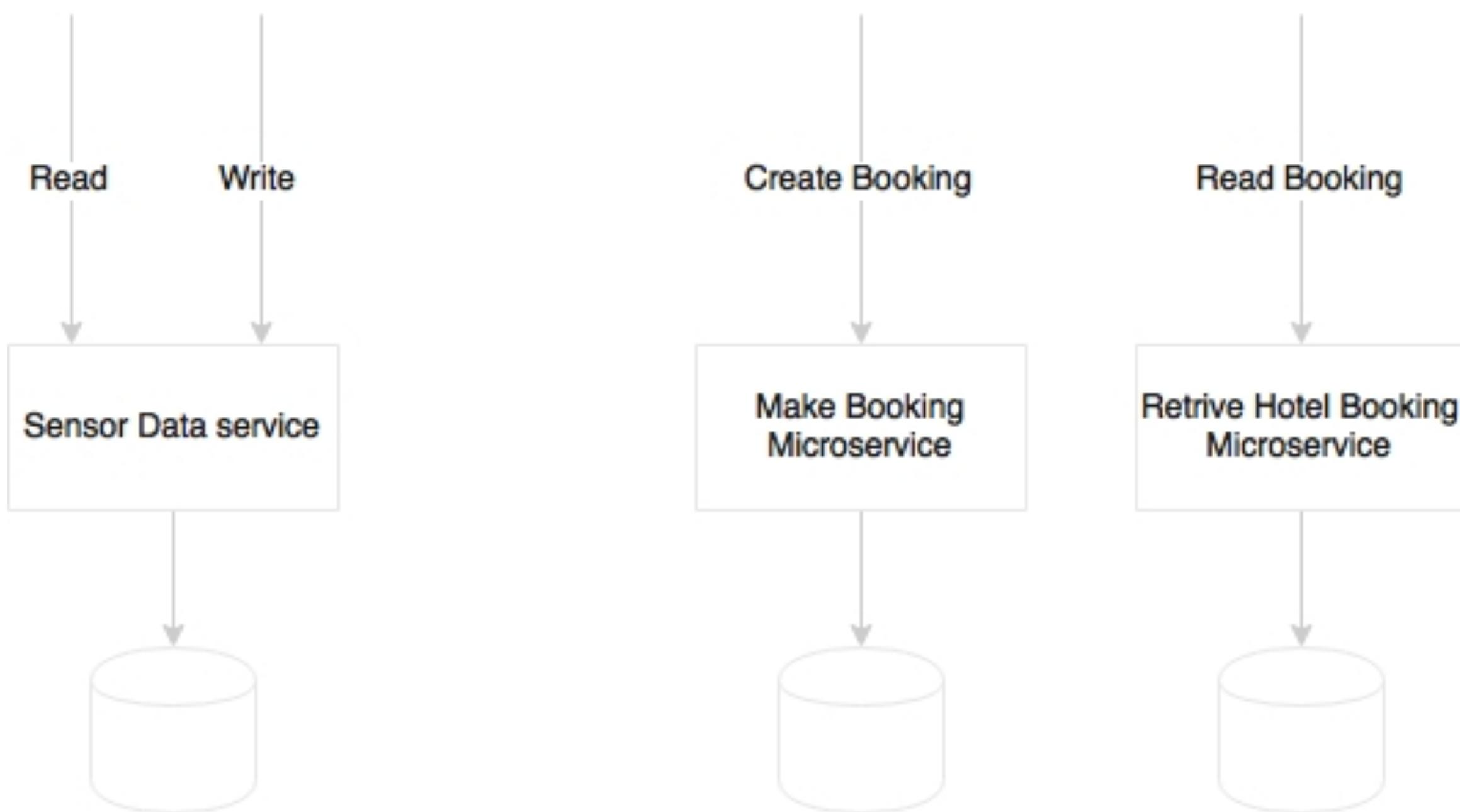
We can model choreography in all cases?

- Changes will be propagated whenever there is a change in the master
- Reservation can use customer preference without fanning out a call
- Today we replicate customer preference, but in another scenario, we may want to reach out to customer service to see whether the customer is black-listed from reserving



How many endpoints in a microservice?

- The question really is whether to limit each microservice with one endpoint or multiple endpoints
- Polyglot architecture could be another scenario where we may split endpoints into different microservices



One microservice per VM or multiple?

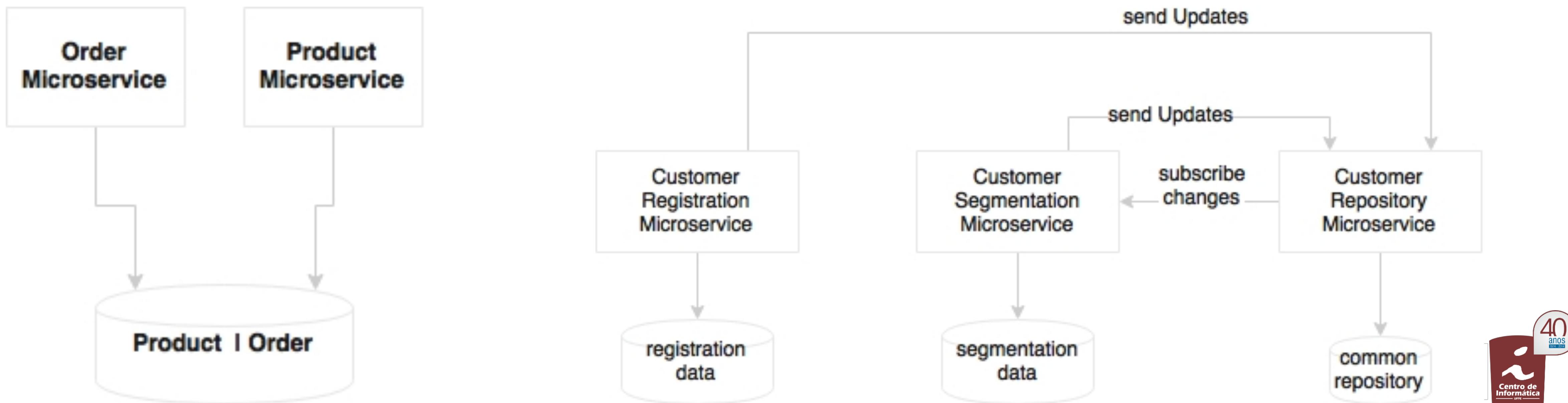
- Whether multiple microservices could be deployed in one virtual machine?
- The simplest way to approach this problem is to ask a few questions:
 - Does the VM have **enough capacity** to run both services under **peak usage**?
 - Do we want to treat these services **differently** to achieve **SLAs** (selective scaling)? For example, for scalability, if we have an all-in-one VM, we will have to replicate VMs which replicate all services.
 - Are there any **conflicting resource requirements**? For example, different OS versions, JDK versions, and others

Rules engine – shared or embedded?

- Either we hand code rules, or we may use a **rules engine**
- Many enterprises manage rules **centrally** in a **rules repository** as well as execute them centrally
 - **Drools** is one of the popular open source rules engines

Can microservices share data stores?

- In principle, microservices should abstract presentation, business logic, and data stores.
- If the services are broken as per the guidelines, each microservice logically **could** use an independent database



Setting up transaction boundaries

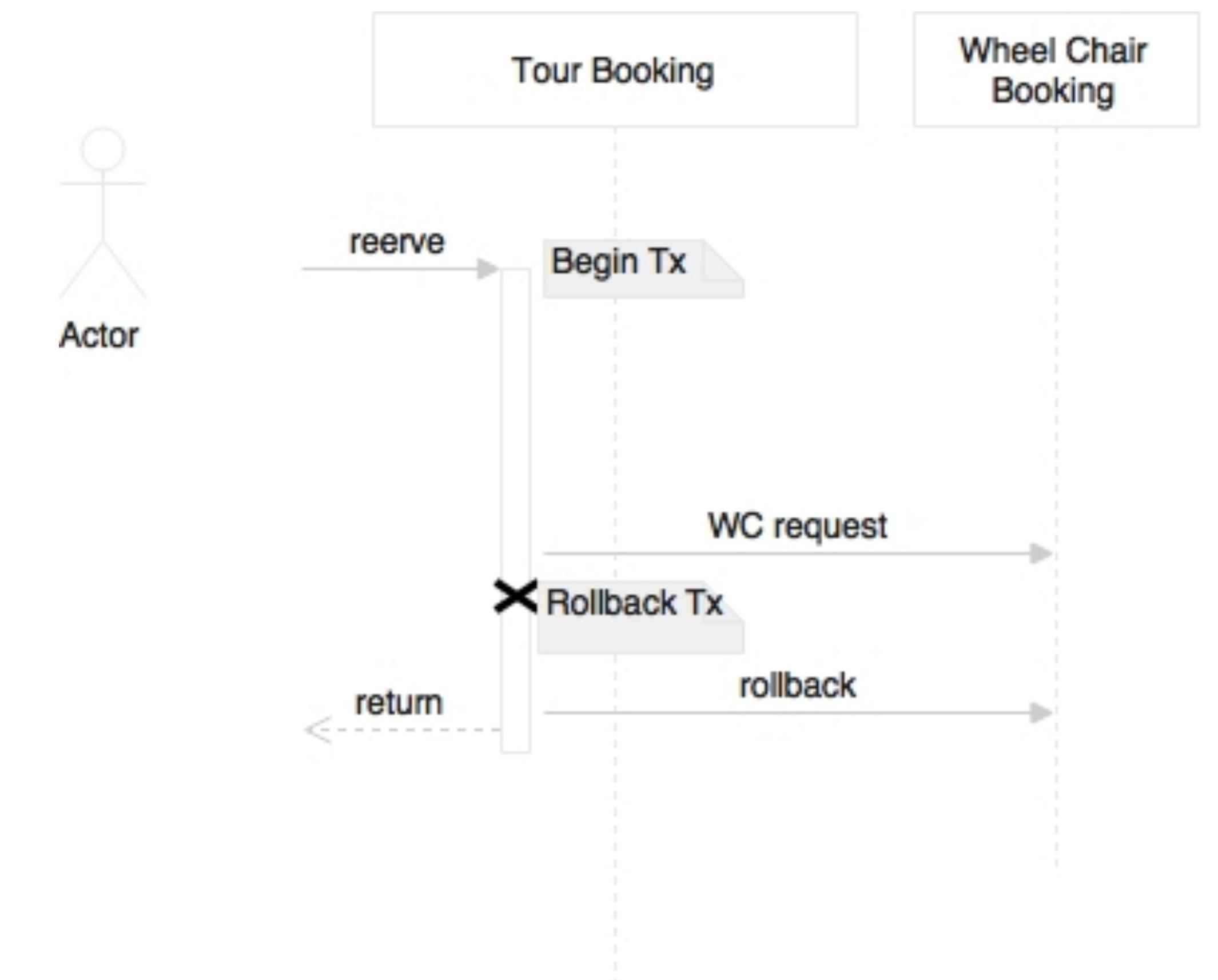
- Is there a place for transactions in microservices?
 - It is appropriate to define transaction boundaries within the microsystem using local transactions
 - However, distributed global transactions should be avoided in the microservices context

Altering use cases to simplify transactional requirements

- **Eventual consistency** is a better option than distributed transactions that span across multiple microservices
 - reduces a lot of overheads
 - but application developers may need to **re-think** the way they **write** application code

Distributed transaction scenarios

- The ideal scenario is to use local transactions within a microservice if required, and completely avoid distributed transactions
- There could be scenarios where at the end of the execution of one service, we may want to send a message to another microservice



Service endpoint design consideration

- Service design has two key elements: **contract design** and **protocol selection**
- **Contract design**
 - A **complex** service contract **reduces** the usability of the service
 - KISS (Keep It Simple Stupid)
 - YAGNI (You Ain't Gonna Need It)
 - Evolutionary design is a great concept
 - Consumer Driven Contracts (CDC) and Postel's law (robustness principle)
 - Be conservative in what you send, be liberal in what you accept

Service endpoint design consideration

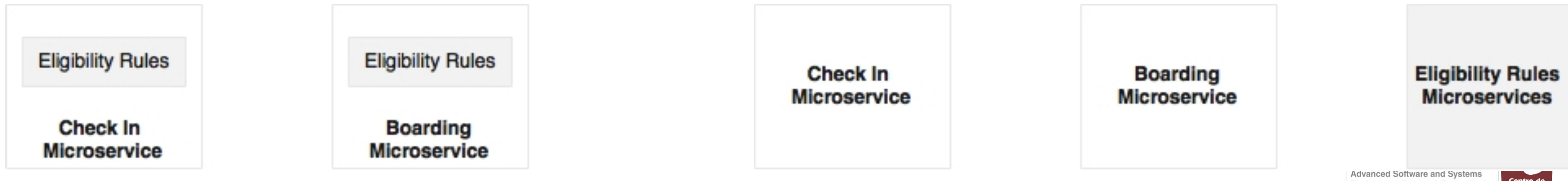
- Protocol selection
 - In the SOA world, HTTP/SOAP, and messaging were kinds of default service protocols for service interactions
 - increases the communication cost; susceptible to network failures; could result in poor performance of services
 - Message-oriented services (asynchronous style of communication)
 - HTTP and REST endpoints (interoperability, protocol handling, traffic routing, load balancing, security systems...)
 - Optimized communication protocols
 - API documentations

Homework 4.1

- What is the main concepts and principles of Consumer Driven Contracts (CDC)?
- How does it fit into the microservices approach?
- How Postel's law could be also relevant in this scenario?
- What are the main solutions (patterns, architectural styles, tools, etc.) to implement:
 - Message-oriented services;
 - HTTP and REST endpoints;
 - Optimized communication protocols;
 - API documentations

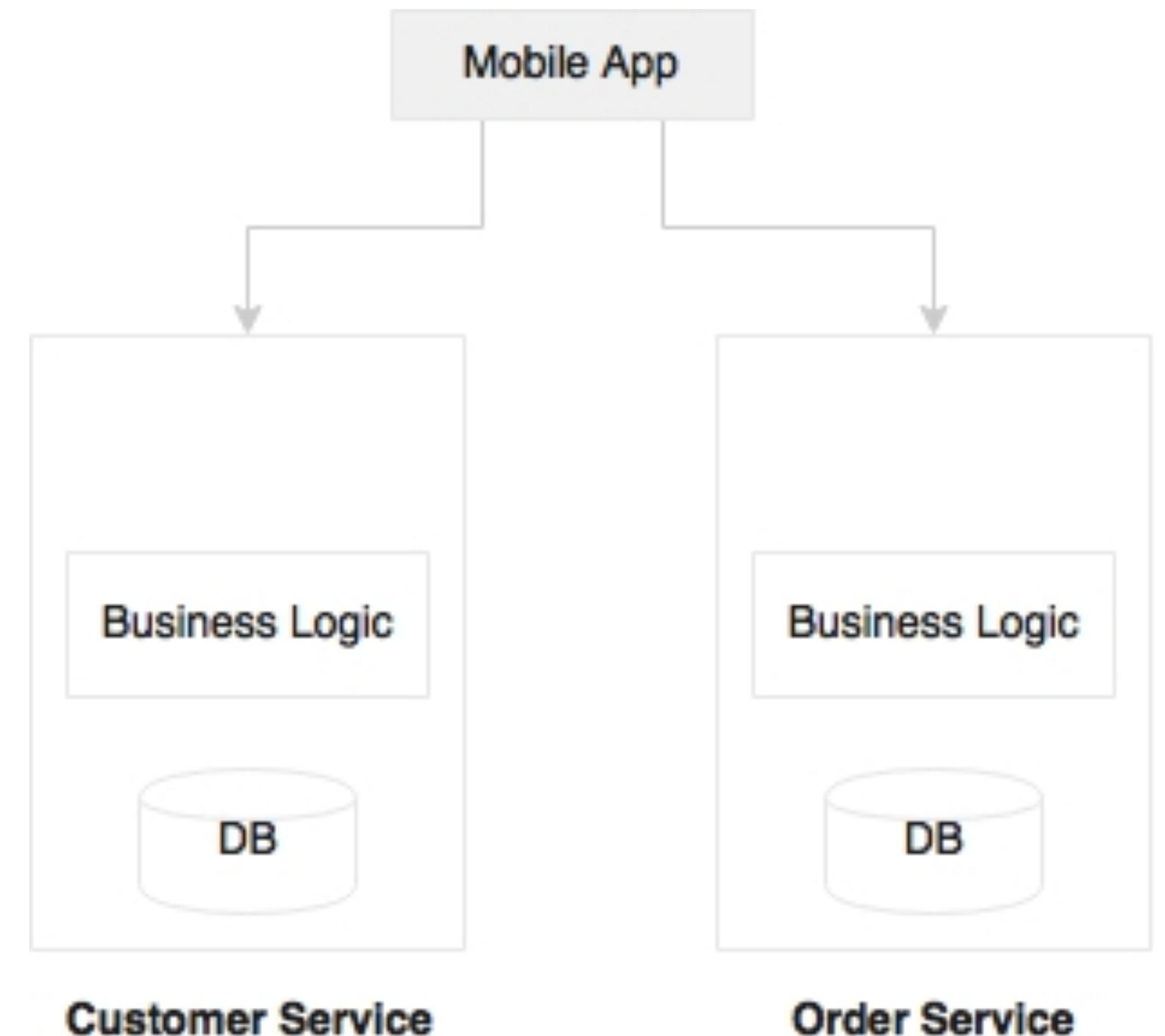
Handling shared libraries

- Microservices: Autonomous and self-contained
 - duplicate code and libraries



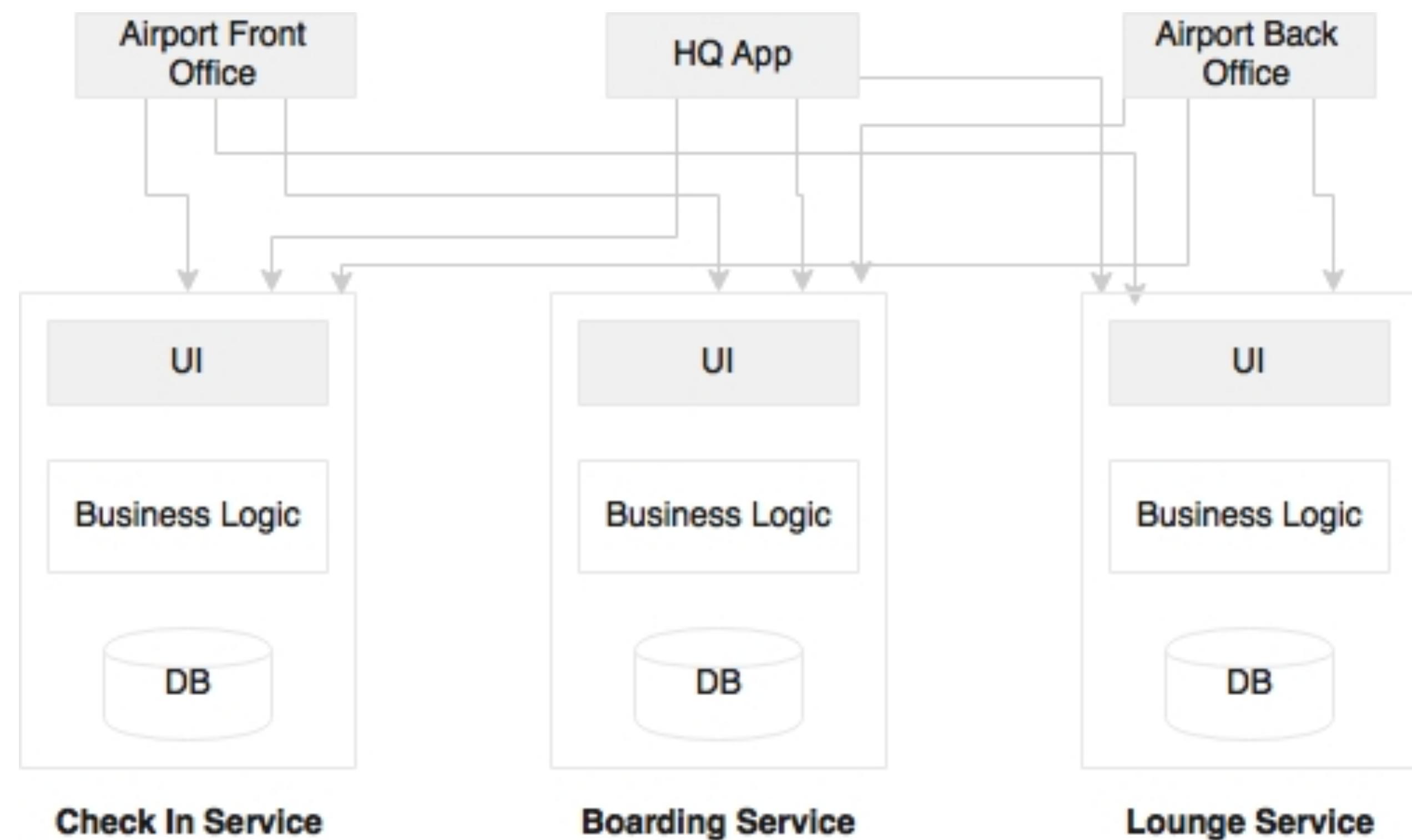
User interfaces in microservices

- The microservices principle advocates a microservice as a **vertical slice** from the database to presentation
 - build quick UI and mobile applications mashing up the existing APIs



User interfaces in microservices

- Build consolidated web applications targeted to communities
- One approach is to build a **container web application** or a **placeholder web application**, which links to multiple microservices at the backend
 - multiple placeholder web applications targeting different user communities



Use of API gateways in microservices

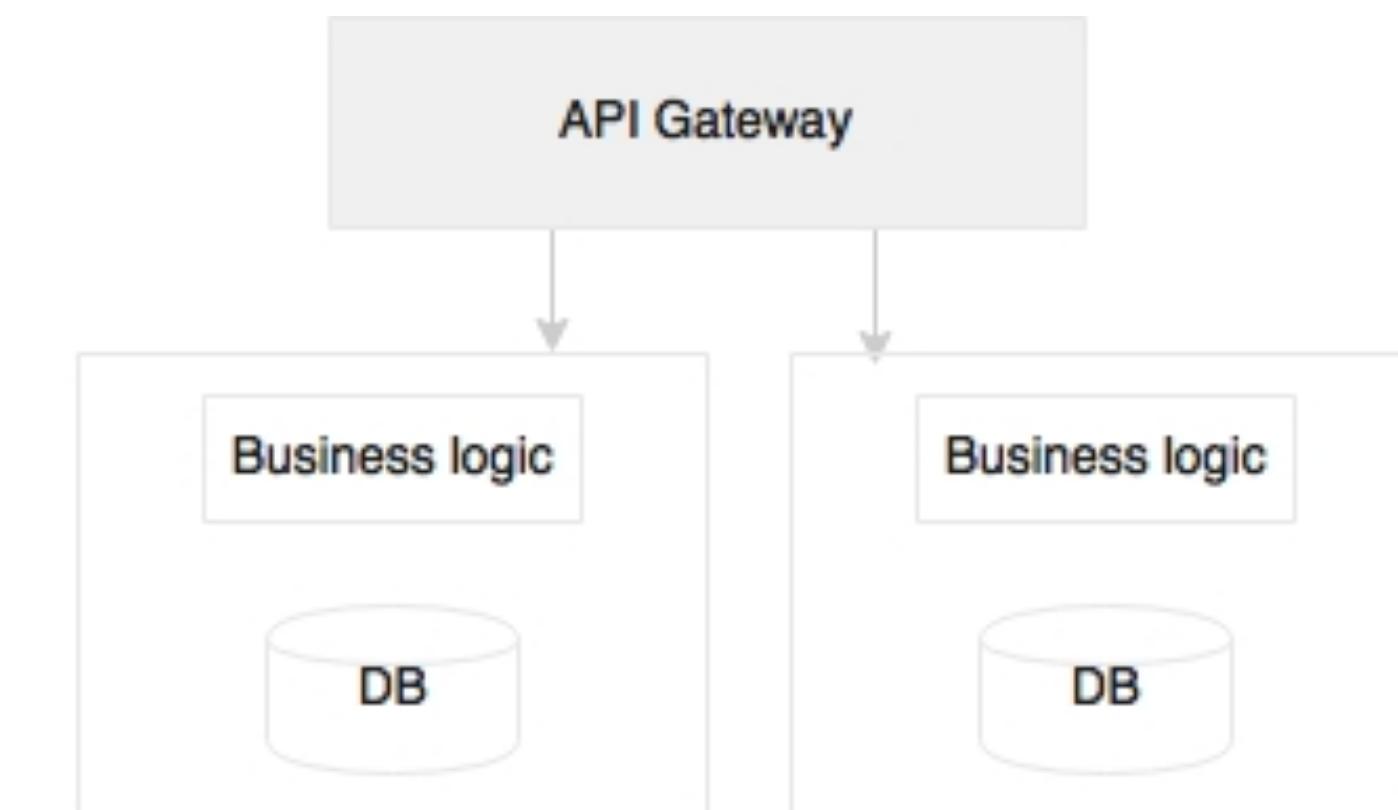
- With the advancement of client-side JavaScript frameworks, the server is expected to expose RESTful services
 - mismatch in contract expectations
 - minimal information is sent with links (HATEOAS)
 - multiple calls to the server to render a page
 - used when the client makes the REST call (also sends the required fields as part of the query string)

Use of API gateways in microservices

- Introduce a level of indirection
 - A gateway component sits between the client and the server, and transforms data as per the consumer's specification
 - do not compromise on the backend service contract
 - leads to **UI services**
 - In many cases, the API gateway acts as a **proxy** to the backend, exposing a **set of consumer-specific APIs**



A) API gateway is part of the micro service



B) Common API gateway

Use of ESB and iPaaS with microservices

- Theoretically, SOA is not all about ESBs, but the reality is that ESBs have always been at the center of many SOA implementations
- What would be the role of an ESB in the microservices world?
 - Microservices: fully cloud native systems, lightweight characteristics of microservices enable automation of deployments, scaling, and so on...
 - ESB: heavyweight in nature, not cloud friendly , protocol mediation, transformation, orchestration, and application adaptors ~> we may not need

Use of ESB and iPaaS with microservices

- Enterprises have **legacy applications**, **vendor applications**, and so on
 - Legacy services use ESBs to connect with microservices
- With the advancement of clouds, the capabilities of ESBs are not sufficient to manage integration between clouds, cloud to on-premise, and so on.
 - **Integration Platform as a Service** (iPaaS) is evolving as the next generation application integration platform, which further reduces the role of ESBs.
 - In typical deployments, iPaaS invokes API gateways to access microservices

Homework 4.2

- How can we have the same features present in ESB's with lightweight tools in the universe of microservices? Justify your answer.

Service versioning considerations

- Versioning helps us to release **new services** without **breaking** the existing consumers
- There are three different ways in which we can version **REST services**:
 - **URI versioning**: version number is included in the URL itself

```
/api/v3/customer/1234  
/api/customer/1234 - aliased to v3.  
  
@RestController("CustomerControllerV3")  
@RequestMapping("api/v3/customer")  
public class CustomerController {  
  
}
```

api/customer/100?v=1.5

- **Media type versioning**: version is set by the client on the HTTP Accept header

```
Accept: application/vnd.company.customer-v3+json
```

Service versioning considerations

- A less effective approach for versioning is to set the version in the custom header

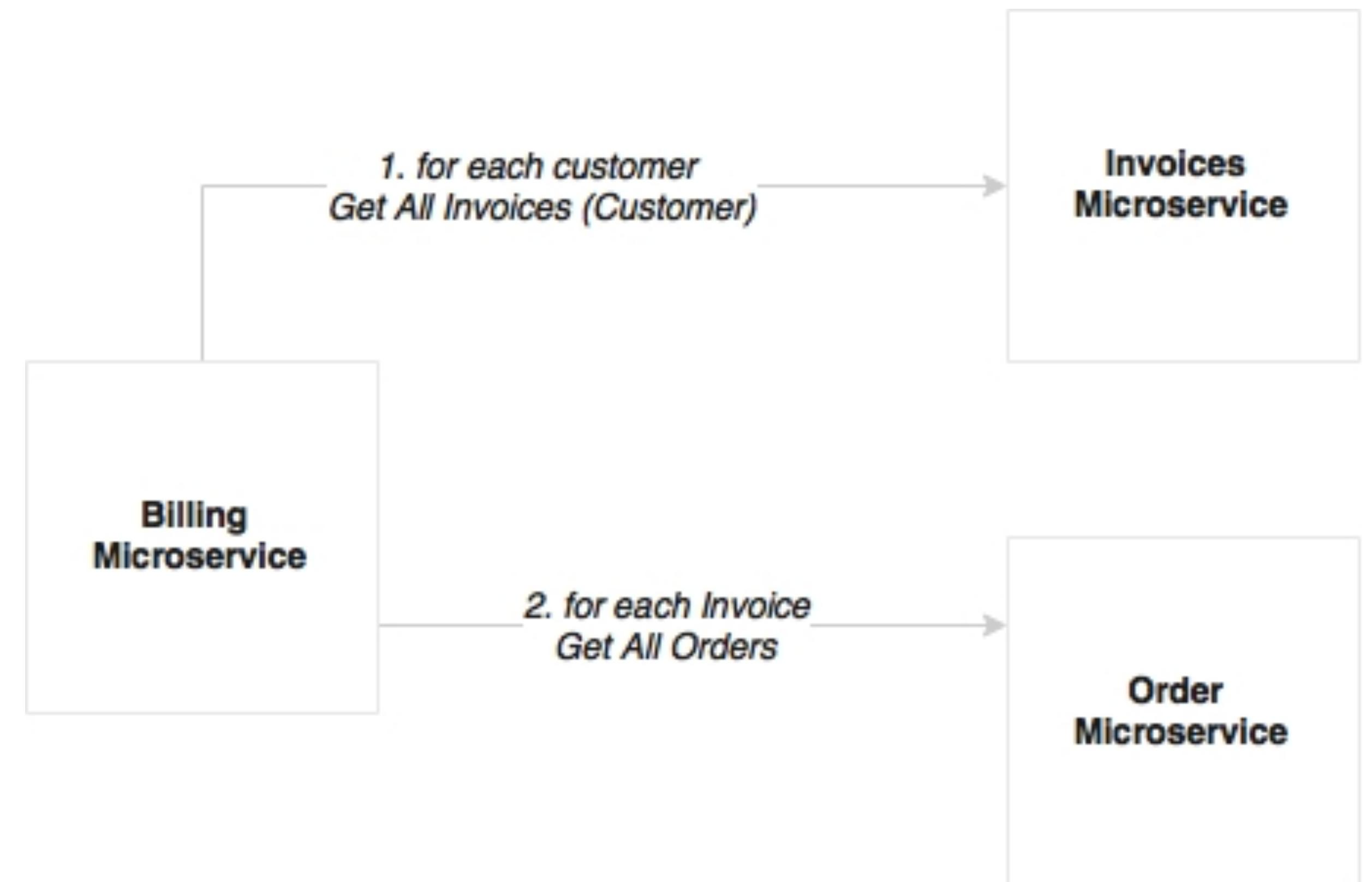
```
@RequestMapping(value = "/{id}", method  
= RequestMethod.GET, headers = {"ver-  
sion=3"})  
public Customer getCustomer(@PathVariable("id") long id,  
                           Service versioning considerations  
                           //other code goes here.  
}
```

Design for cross origin

- Composite UI web applications may call **multiple microservices** for accomplishing a task, and these could come from different domains and hosts
- CORS allows browser clients to send requests to services hosted on different domains
 - One approach is to enable all microservices to allow cross origin requests from other trusted domains
 - The second approach is to use an API gateway as a single trusted domain for the clients

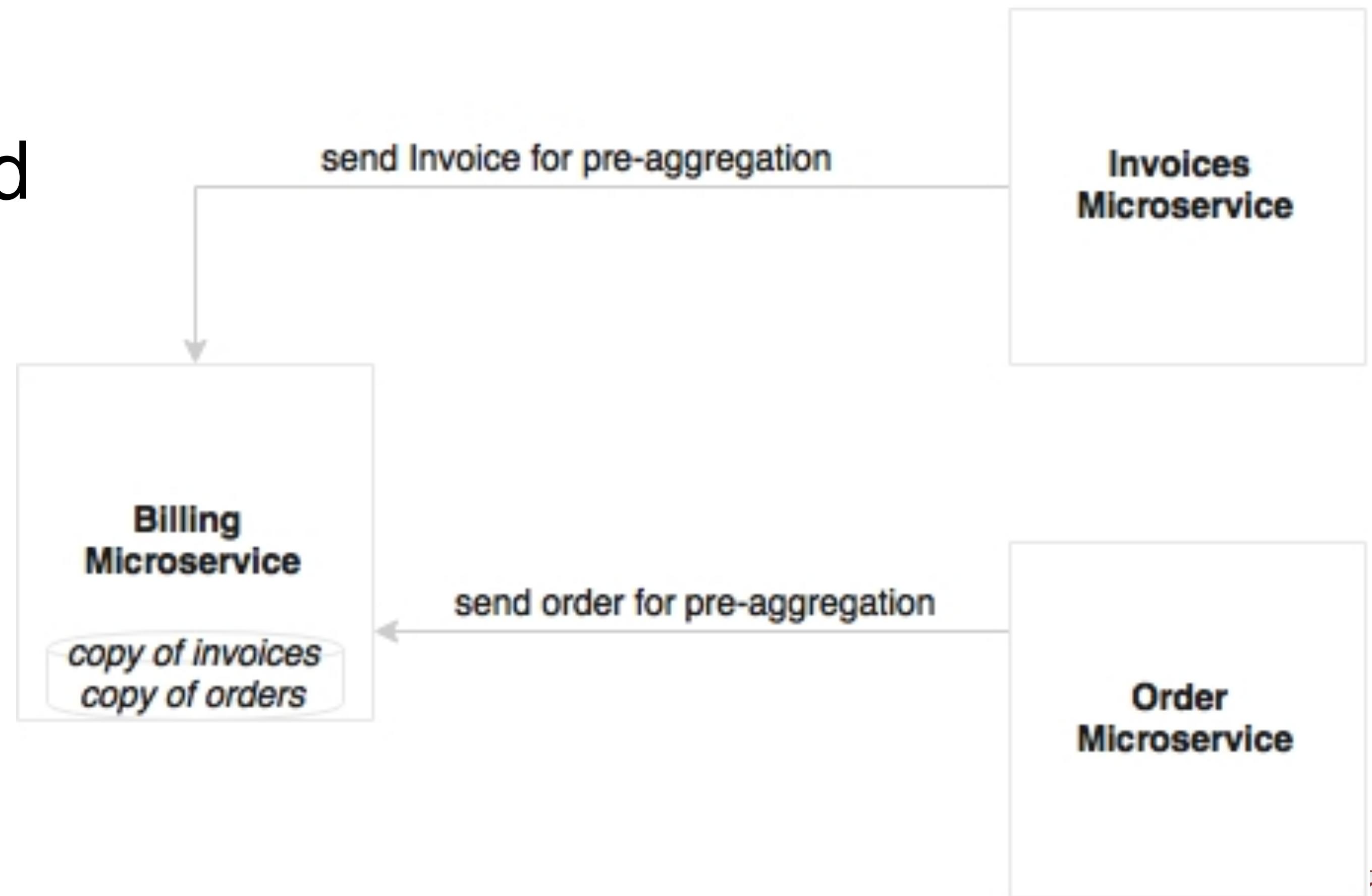
Microservices and bulk operations

- Since we have **broken** monolithic applications into **smaller**, focused services, it is no longer possible to use join queries **across** microservice data stores
- The challenge that arises is that the **Billing** service has to query the **Invoices** service for each **customer** to get all the invoices, and then for each invoice, call the **Order** service for getting the orders



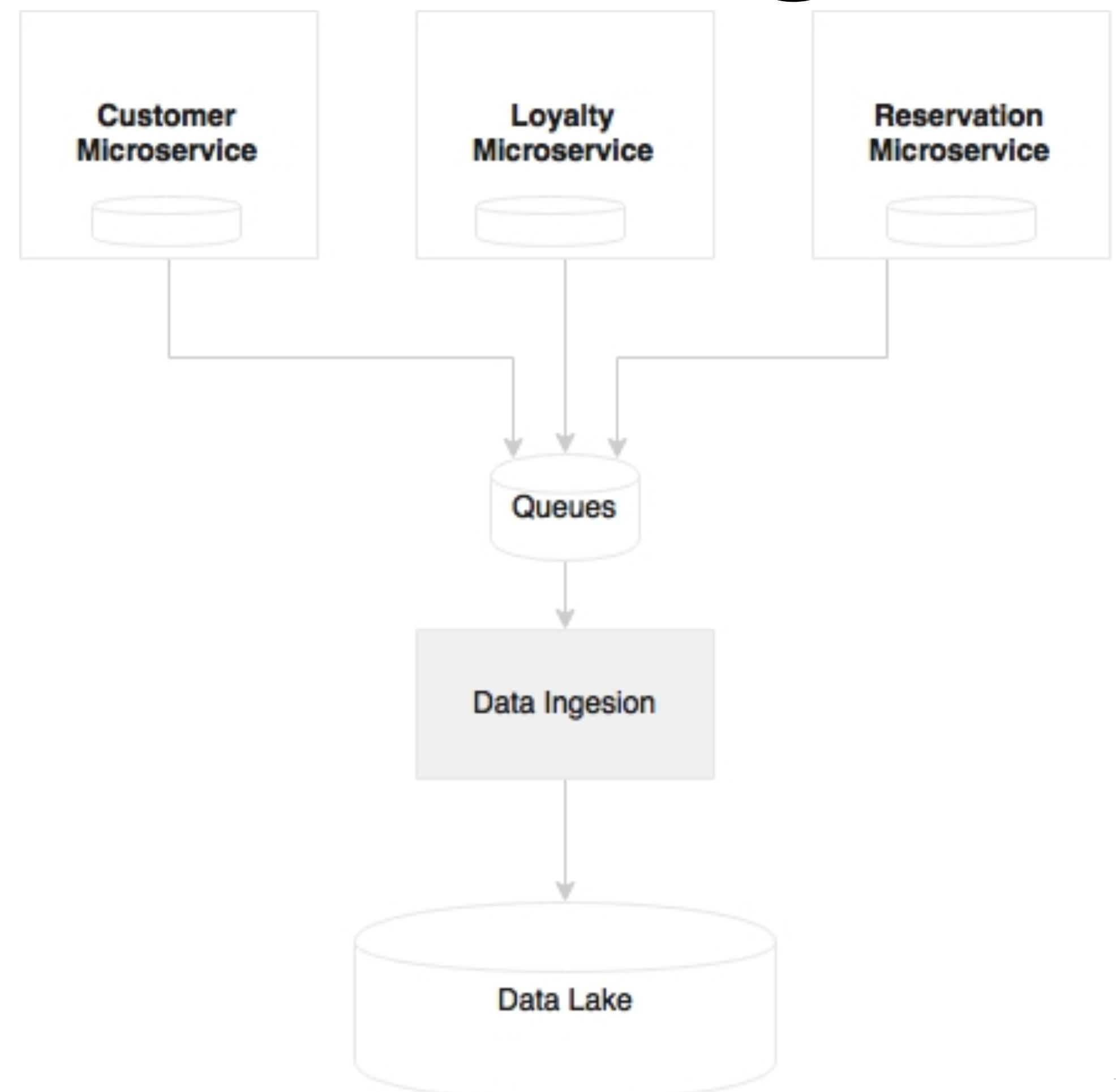
Microservices and bulk operations

- Pre-aggregate data as and when it is created
- Using of batch API
 - each batch further uses parallel threads



Microservices challenges

- Data islands
 - Fragmenting data into **heterogeneous** data islands
 - Traditional data warehouses like Oracle, Teradata, and others are used primarily for **batch reporting**
 - But with NoSQL databases (like Hadoop) and **microbatching** techniques, near real-time analytics is possible with the concept of **data lakes**

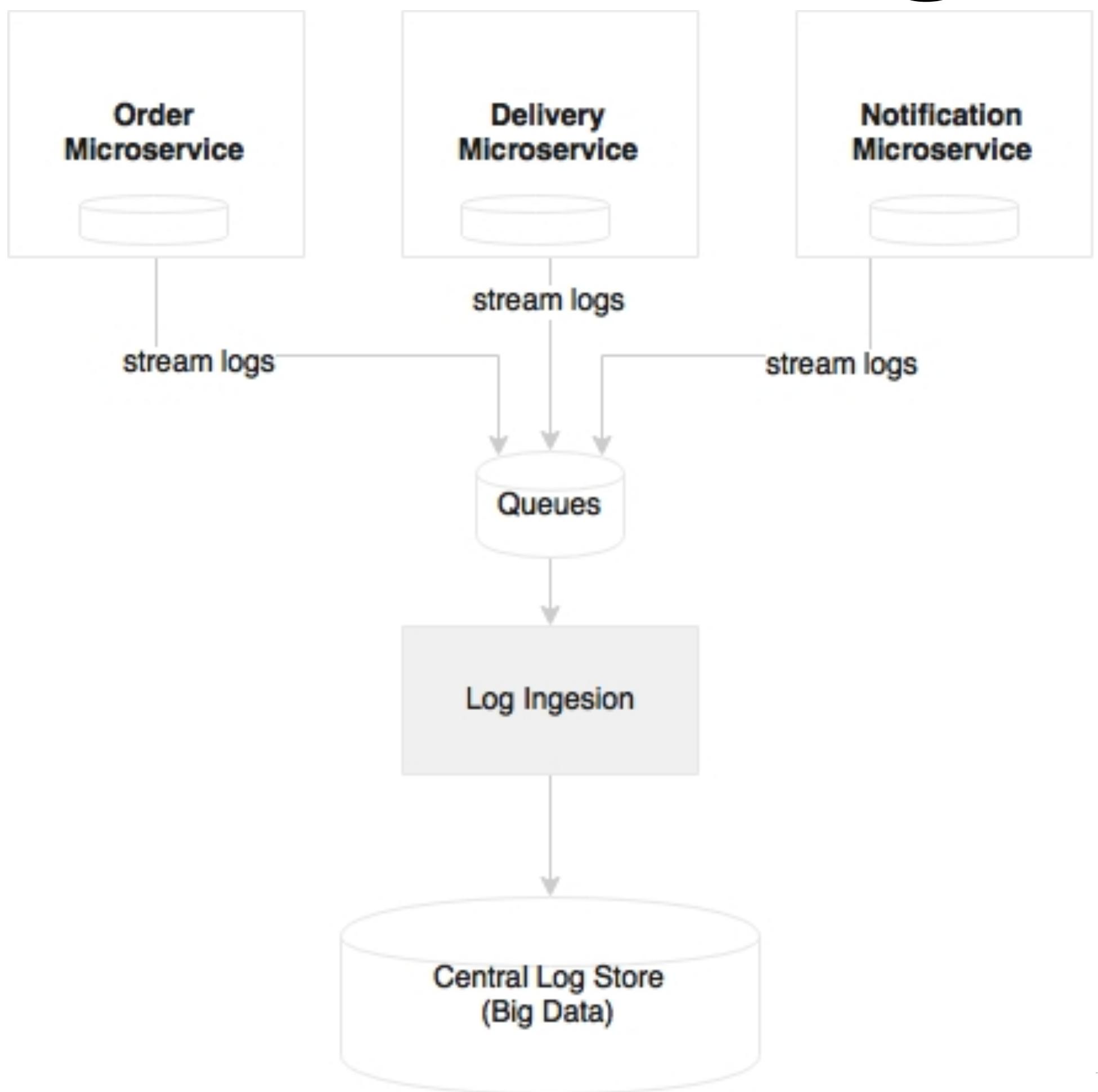


Homework 4.3

- How can be done data porting from microservices to a data lake or a data warehouse? Explain in details your answer.

Logging and monitoring

- Log files are a good piece of information for analysis and debugging
- When we scale services across multiple machines, each service instance could produce separate log files



Dependency management

- Too many dependencies could raise challenges in microservices.
Four important design aspects are stated as follows:
 - Reducing dependencies by properly designing service boundaries.
 - Reducing impacts by designing dependencies as loosely coupled as possible. Also, designing service interactions through asynchronous communication styles.
 - Tackling dependency issues using patterns such as circuit breakers.
 - Monitoring dependencies using visual dependency graphs.

Organization culture

- Agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning...

Governance challenges

- Microservices impose **decentralized** governance, and this is quite in contrast with the traditional SOA governance
- There are number of **challenges** that comes with a decentralized governance model
 - How do we understand who is consuming a service?
 - How do we ensure service reuse?
 - How do we define which services are available in the organization?
 - How do we ensure enforcement of enterprise polices?

Operation overheads

- Microservices deployment generally **increases** the number of deployable units and virtual machines (or containers)
- the number of **configurable items** (CIs) becomes **too high**, and the **number of servers** in which these CIs are deployed might also be **unpredictable**

Testing microservices

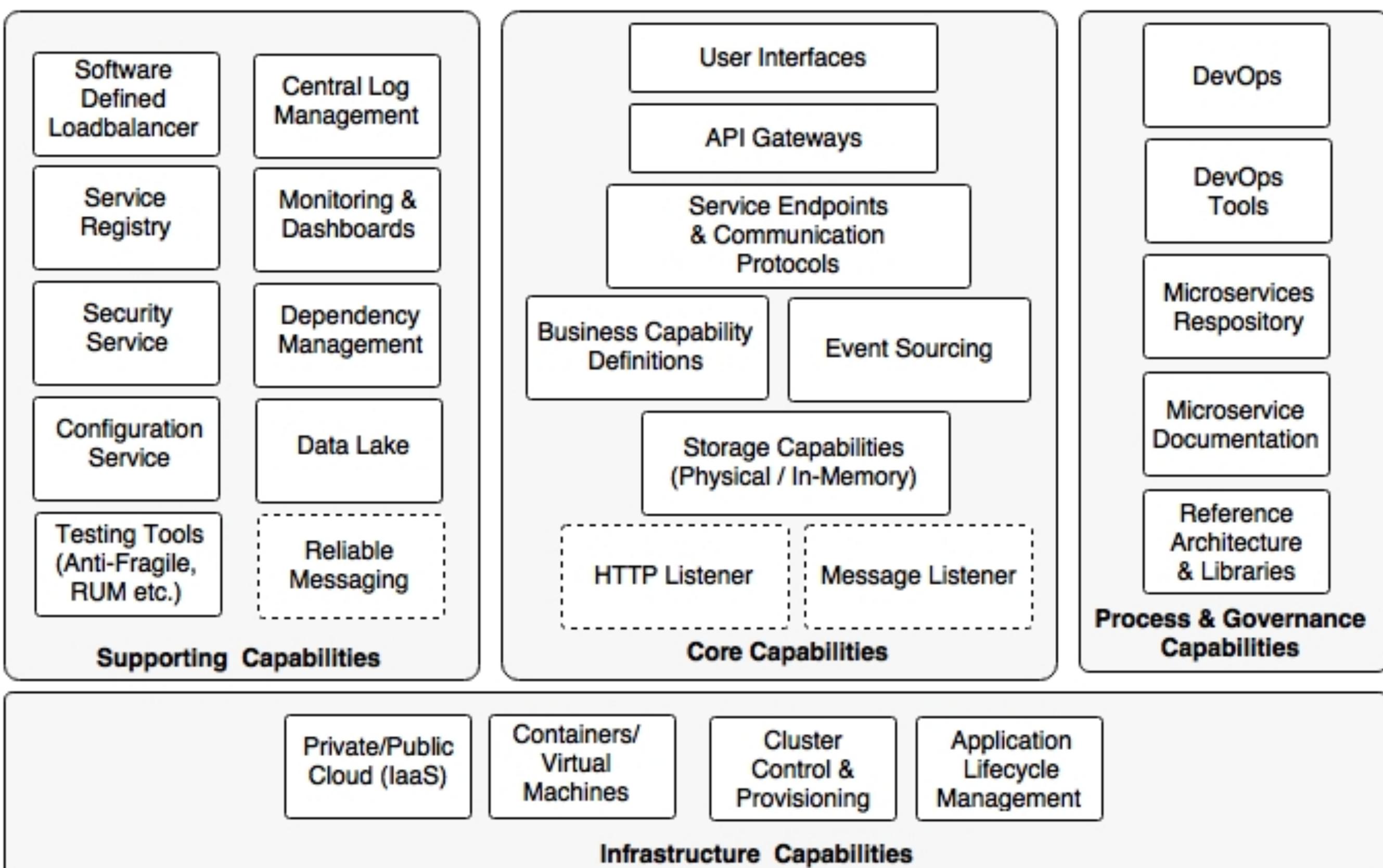
- The issue is how do we test an end-to-end service to evaluate its behavior?
 - Service virtualization or service mocking
 - Consumer driven contract
- Test automation, appropriate performance testing, and continuous delivery approaches such as A/B testing, future flags, canary testing, blue-green deployments, and red-black deployments, all reduce the risks of production releases

Infrastructure provisioning

- Manual deployment could severely challenge the microservices rollouts
- Microservices require a supporting **elastic cloud-like infrastructure** which can **automatically provision VMs or containers**, **automatically deploy applications**, **adjust traffic flows**, **replicate new version to all instances**, and **gracefully phase out older versions**

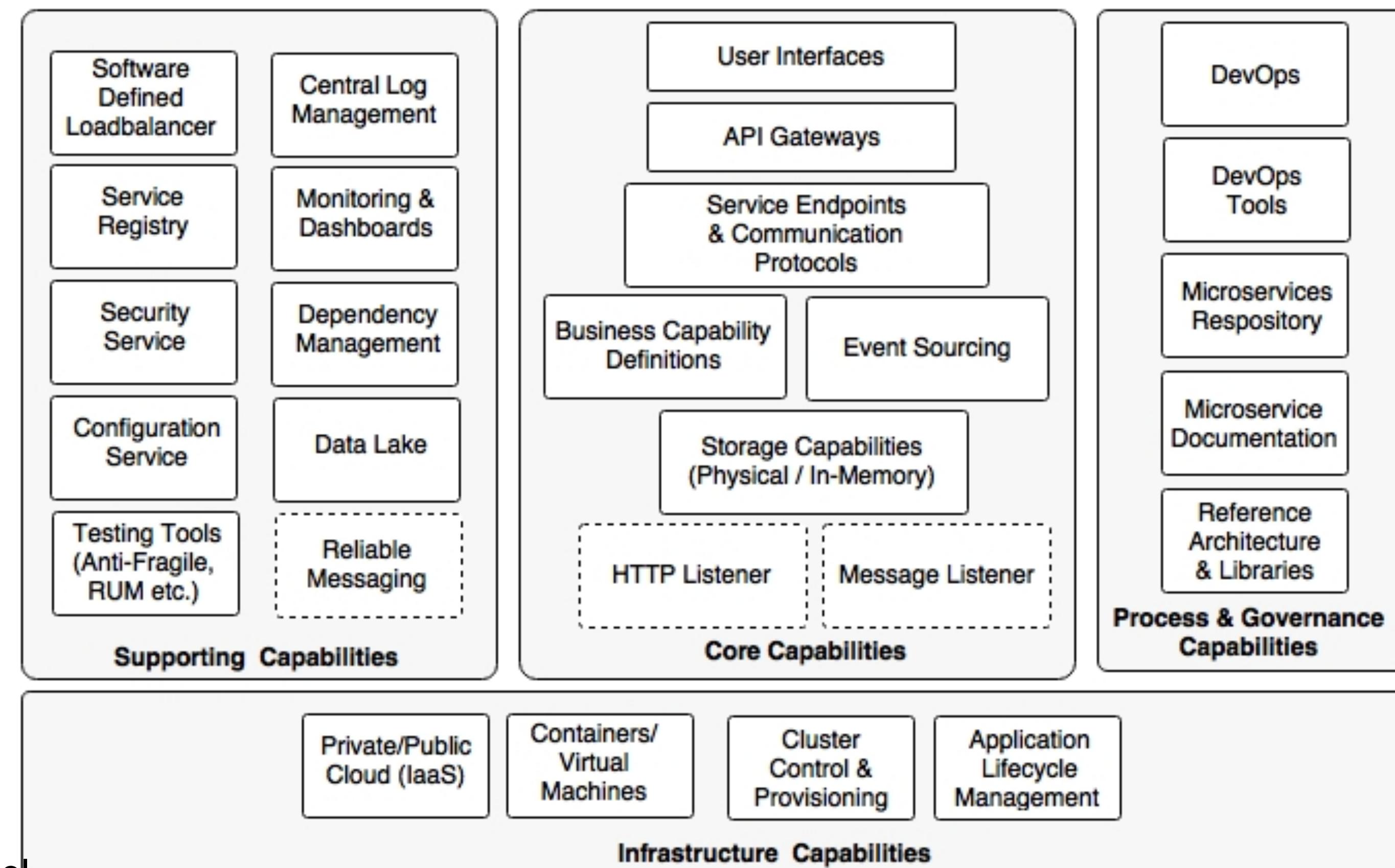
The microservices capability model

- The capability model is broadly classified in to four areas:
 - **Core capabilities**: These are part of the microservices themselves
 - **Supporting capabilities**: These are software solutions supporting core microservice implementations
 - **Infrastructure capabilities**: These are infrastructure level expectations for a successful microservices implementation
 - **Governance capabilities**: These are more of process, people, and reference information



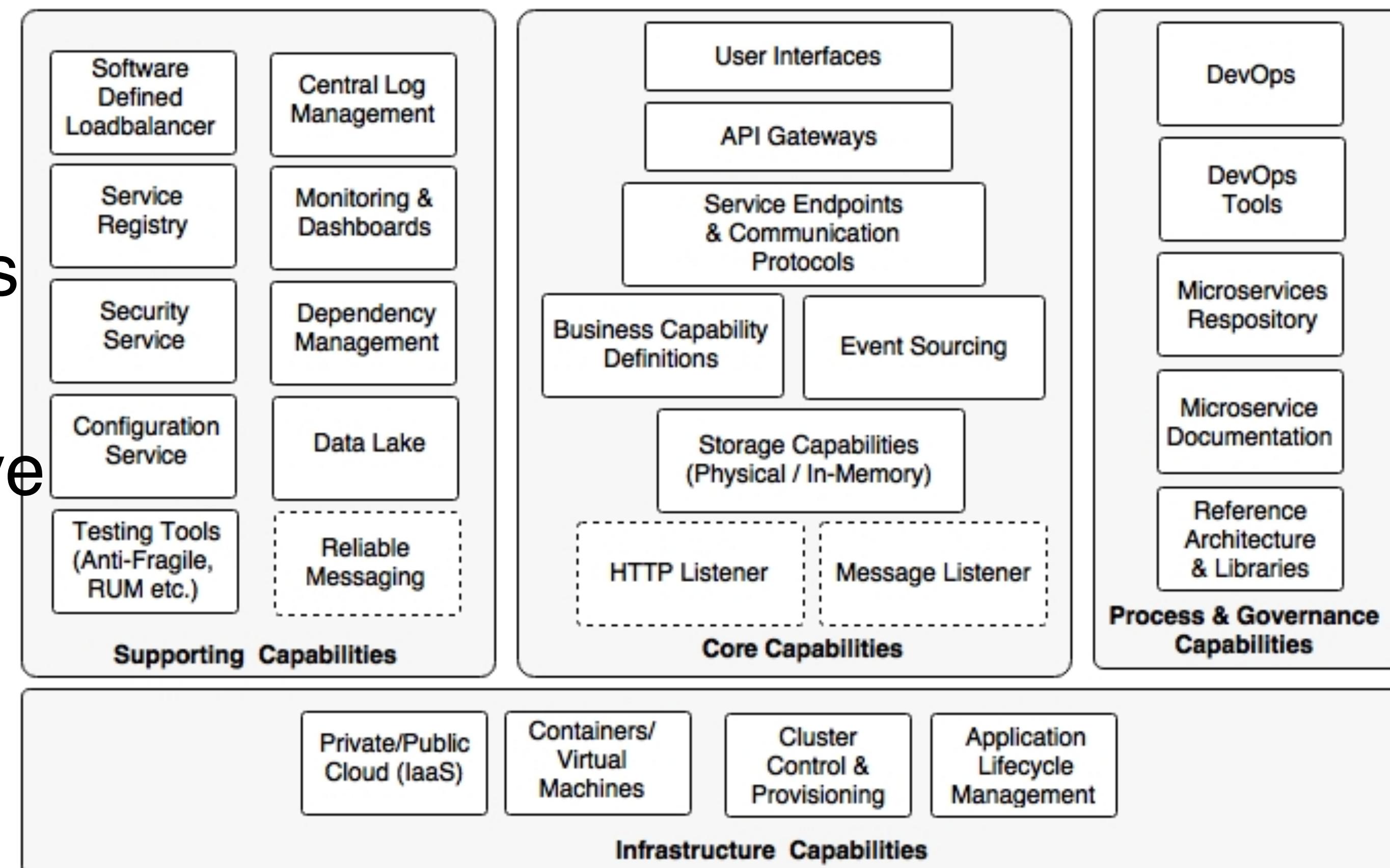
Core capabilities

- Service listeners (HTTP/messaging)
 - HTTP listener is embedded within the microservices, thereby eliminating the need to have any external application server requirement
- Storage capability
 - could be either a physical storage, or it could be an in-memory store
- Business capability definition
 - where the business logic is implemented
- Event sourcing
 - Microservices send out state changes
- Service endpoints and communication protocols
 - Define the APIs for external consumers to consume
- API gateway
 - useful for policy enforcements
- User interfaces
 - could be implemented in any technology, and are channel- and device-agnostic.



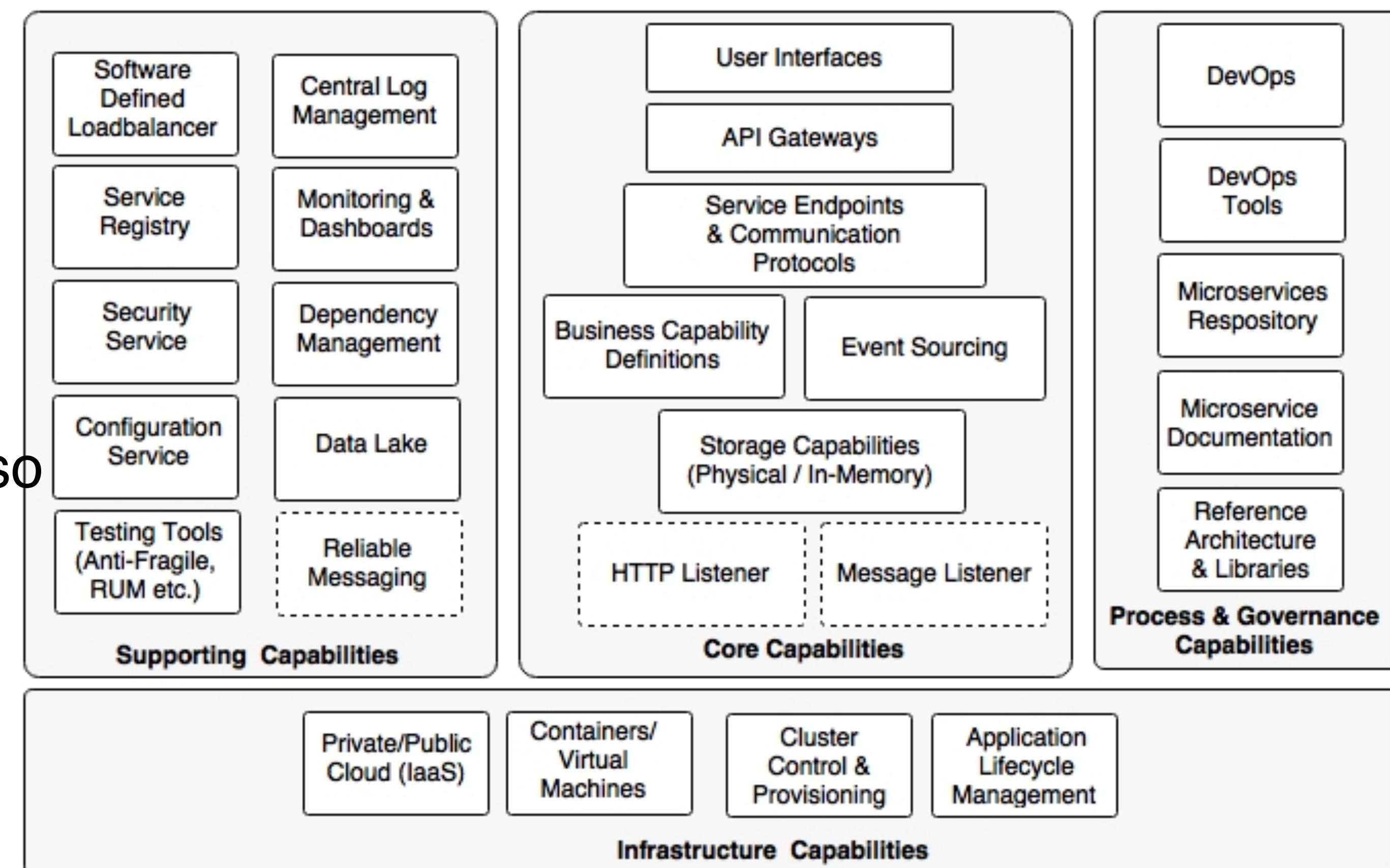
Infrastructure capabilities

- Cloud
 - traditional data center???
- Containers or virtual machines
 - Managing large physical machines is not cost effective
- Cluster control and provisioning
- Application lifecycle management



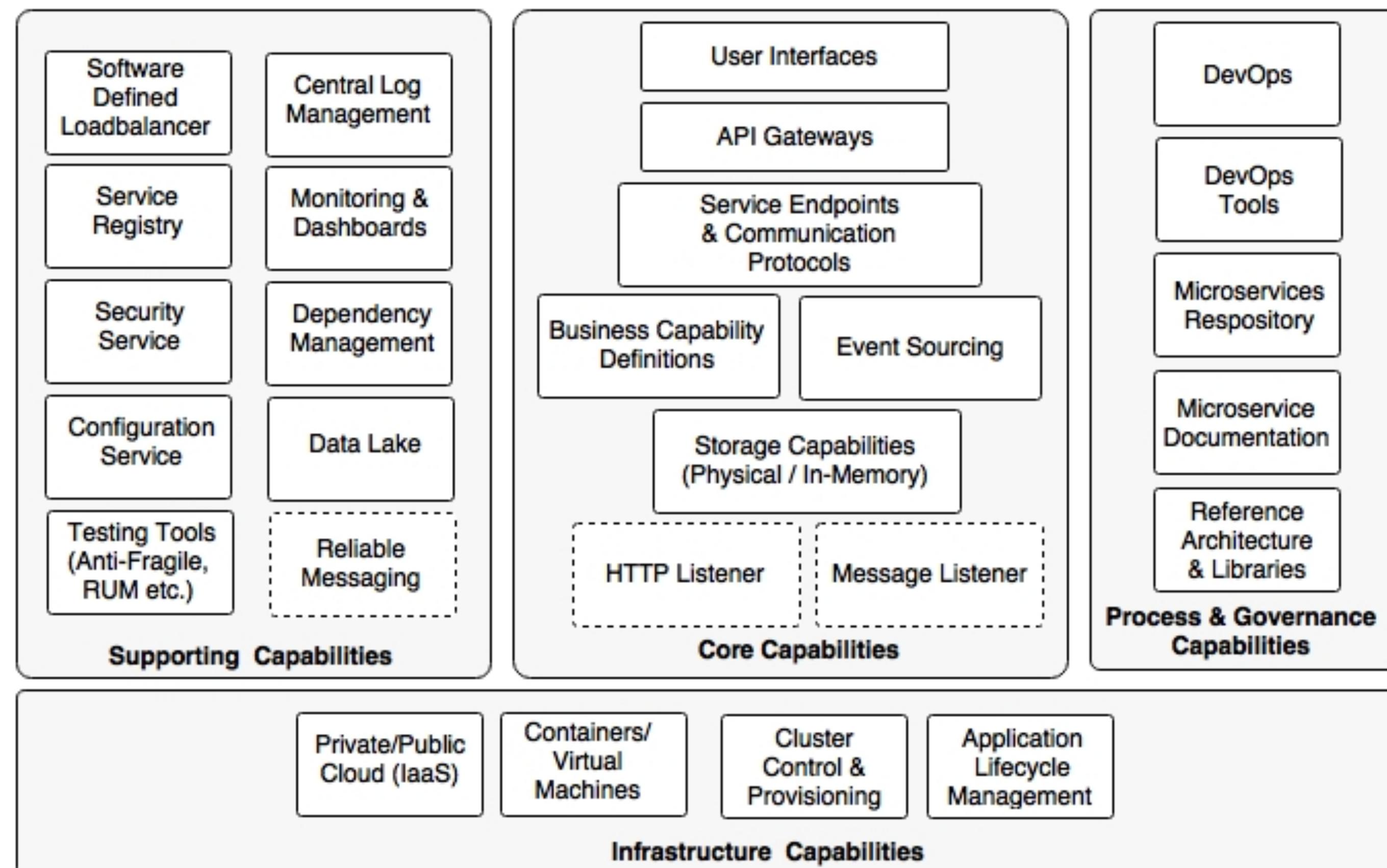
Supporting capabilities

- Software defined load balancer
- Central log management
- Service registry
- Security service
- Service configuration
- Testing tools (anti-fragile, RUM, and so on)
- Monitoring and dashboards
- Dependency and CI management
- Data lake
- Reliable messaging



Process and governance capabilities

- DevOps
- DevOps tools
- Microservices repository
- Microservices documentation
- Reference Architecture & Libraries



Homework 4

- 4.1. Service endpoint design consideration
- 4.2. Use of ESB and iPaaS with microservices
- 4.3. Microservices challenges - Data islands
- Submit in our Slack team, [http://
if1007-2018-1.slack.com](http://if1007-2018-1.slack.com), a MD file (**LOGIN-HW4.md**) containing your answer.
- Due **D+12** (Monday, 4/2), **23:59**.