Microservices architecture

Youssef Ghatas

Acknowledgement

- Some slides are adapted from: Ian Sommerville, “Engineering Software Products” 1st Edition

Agenda

- Intro to microservices
  - Monolith vs microservices
  - Cohesion and coupling.
- API Gateway
- How to architecture it
  - Decomposition Guideline
  - Communication & Coordination & Consistency
  - Orchestration and choreography
- Challenges
  - Failure Handling: Timeout & Circuit Breaker
- Restful Services
- Deployment

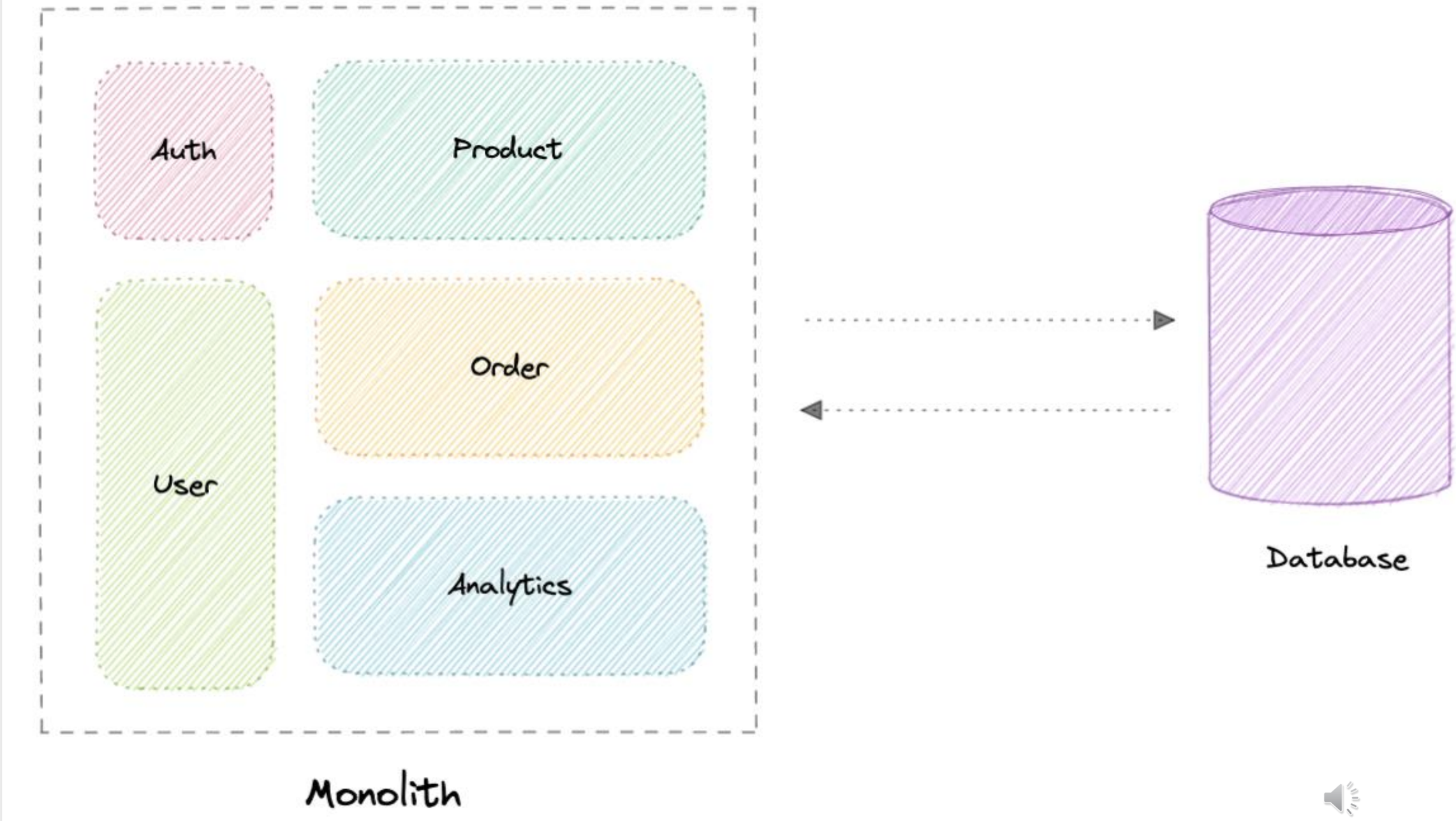
Software services

- A software service is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, **without side effects (each microservice has clear intended objective)**.
  - The service is accessed through its published interface and all details of the service implementation are hidden.
  - Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.
- As there is **no local state**, services can be dynamically reallocated from one virtual server to another and replicated across several servers.

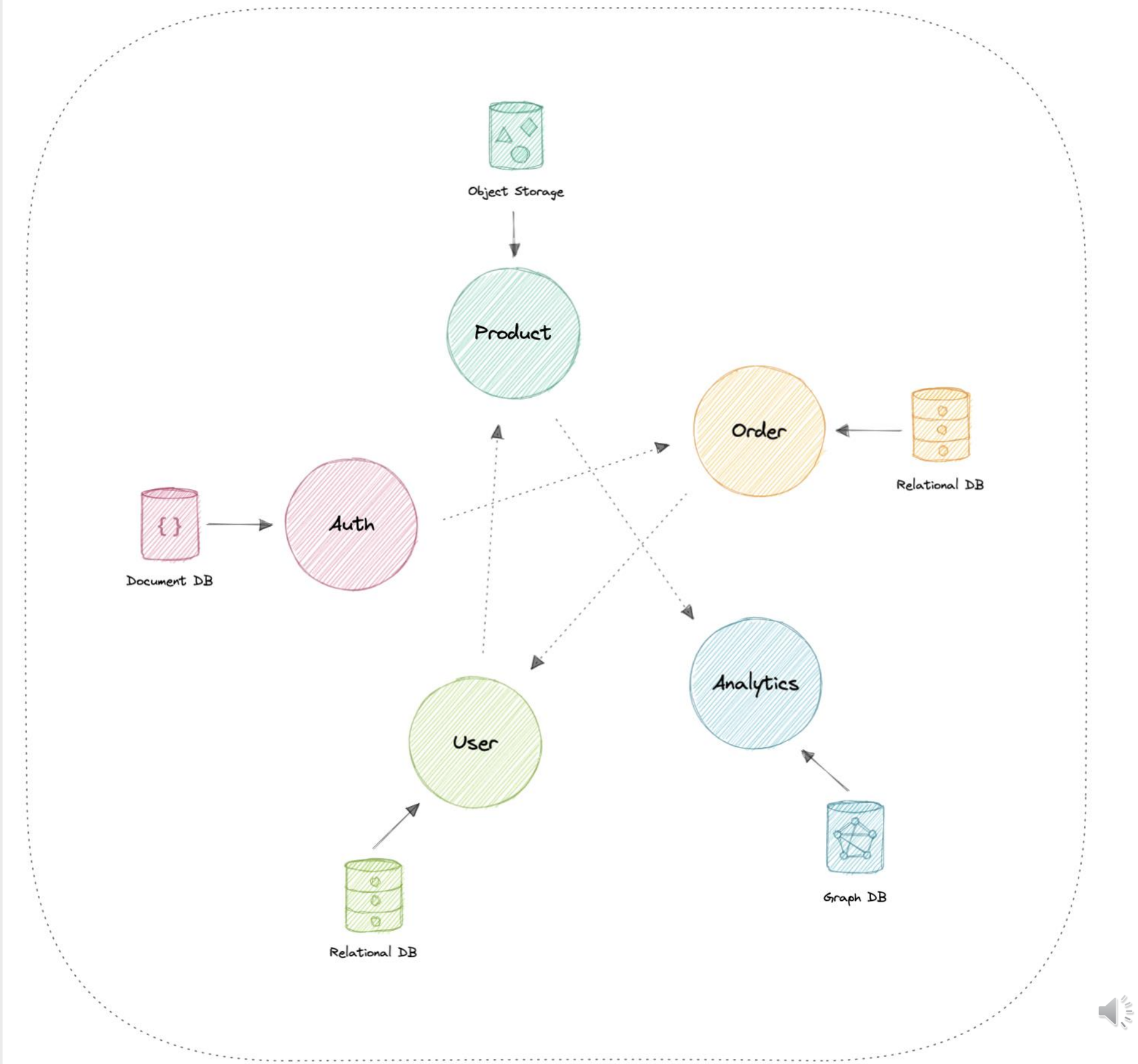
Microservices

- Microservices are **small-scale**, **stateless**, services that have a single responsibility. They are combined to create applications.
- They are **completely independent** with their own database and UI management code.
- Software products that use microservices have a ***microservices architecture***.

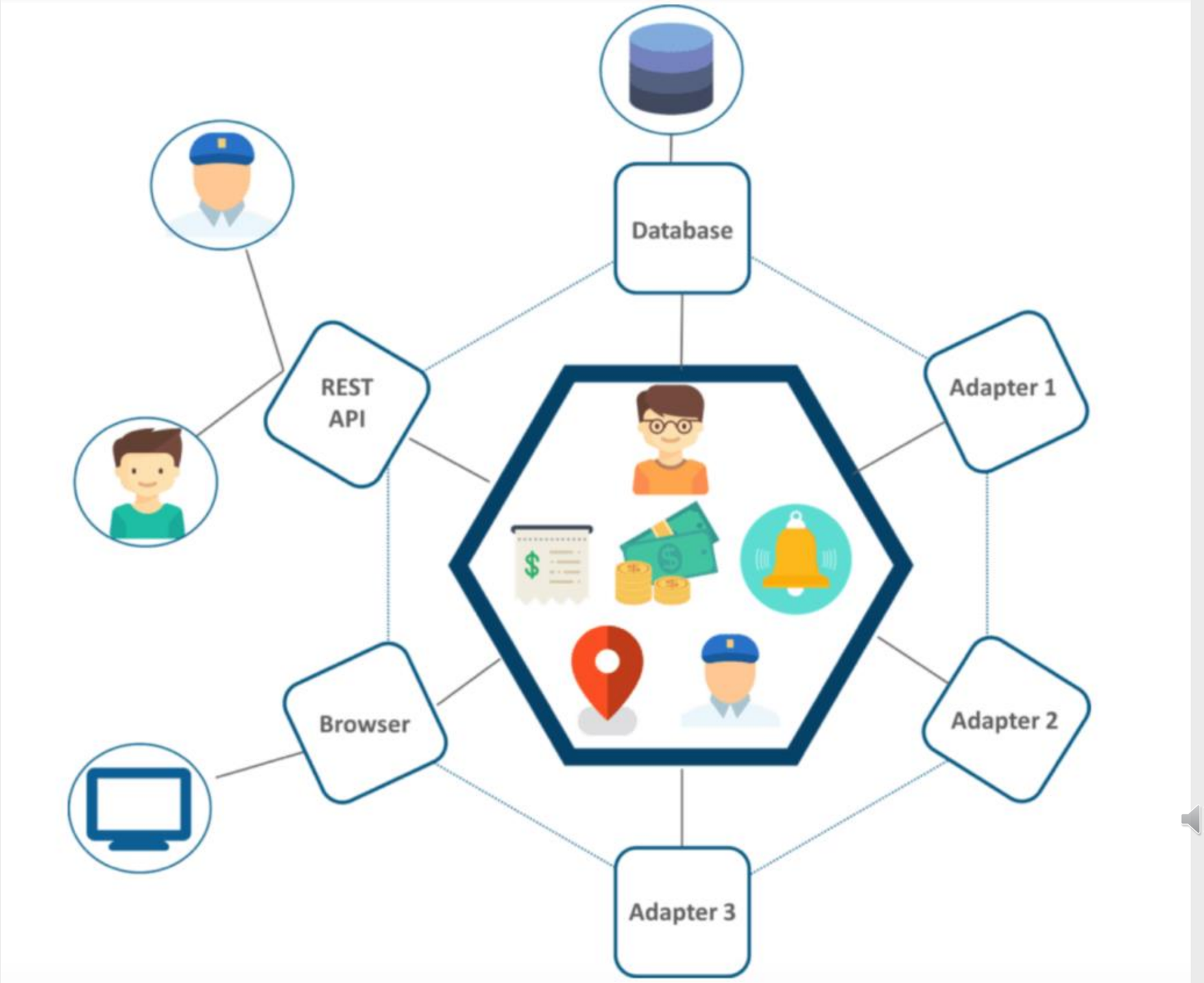
Monolith vs. Microservices



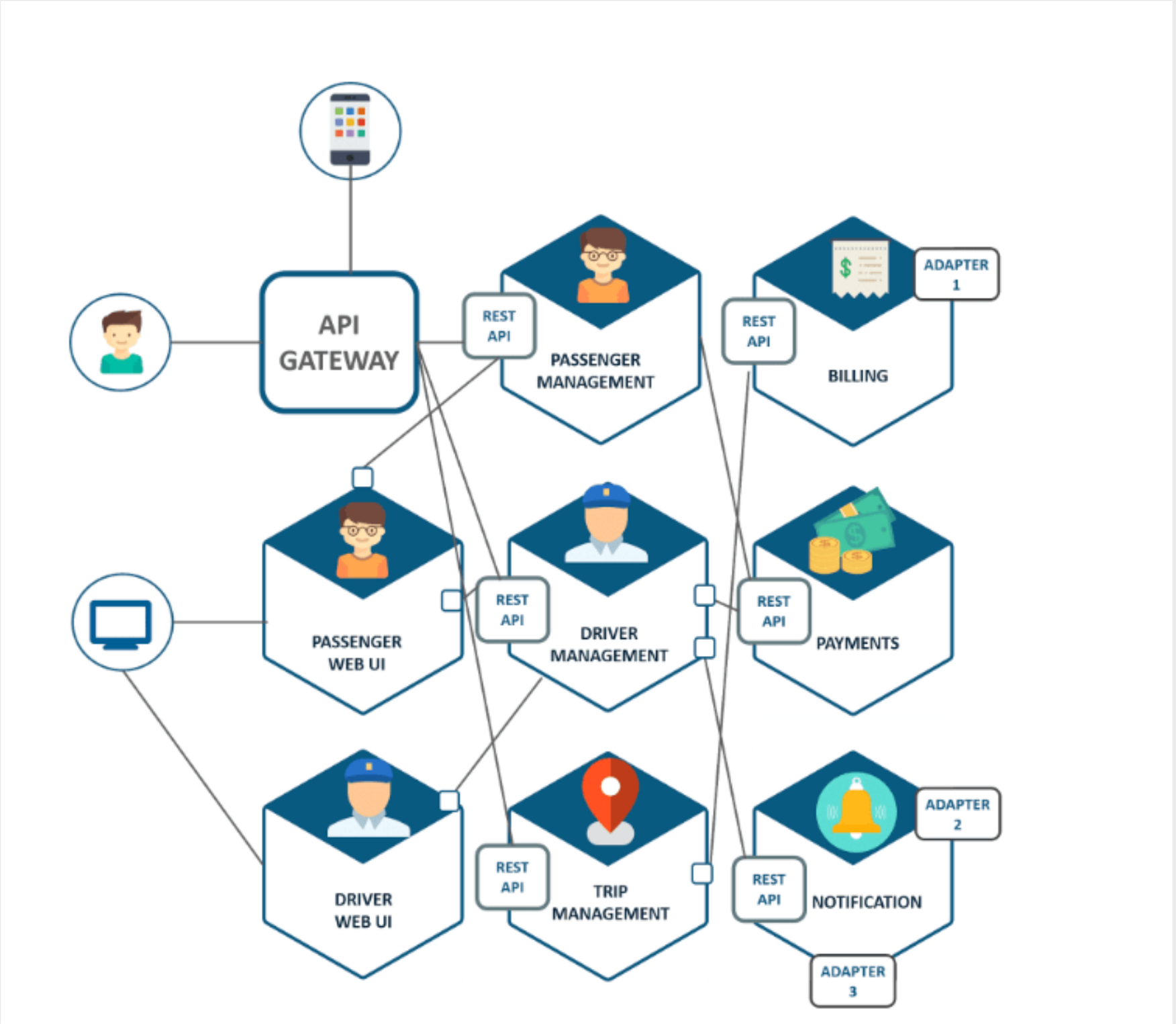
https://github.com/karanpratapsingh/system-design#monoliths-and-microservices


Object Storage
Product
Order
Relational DB
Auth
Document DB
User
Analytics
Relational DB
Graph DB
Microservices

Uber: From monolith to Micro-services



Uber: From monolith to Micro-services



https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a

Characteristics of microservices

Self-contained
Microservices do not have external dependencies. They manage their own data and implement their own user interface.

Lightweight
Microservices communicate using lightweight protocols, so that service communication overheads are low.

Implementation-independent
Microservices may be implemented using different programming languages and may use different technologies (e.g. different types of database) in their implementation.

Independently deployable
Each microservice runs in its own process and is independently deployable, using automated systems.

Business-oriented
Microservices should implement business capabilities and needs, rather than simply provide a technical service.

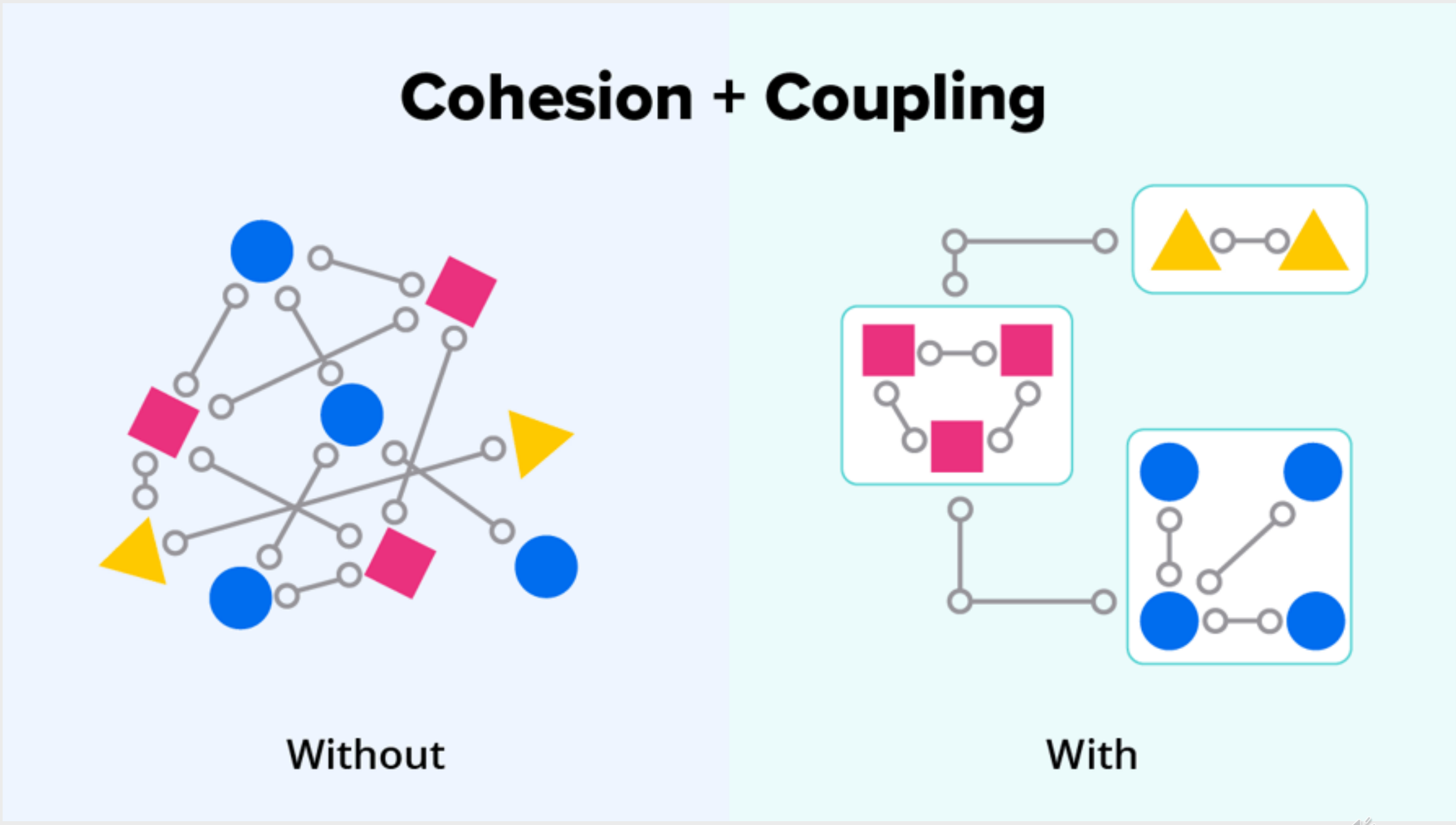
Microservice communication

- Microservices communicate by exchanging messages.
- A message that is sent between services includes some **data required to deliver the requested service**.
- Services return a response to service request messages.
  - An authentication service may send a message to a login service that includes the name input by the user.
  - The response may be a token associated with a valid user name or might be an error saying that there is no registered user.

Microservice characteristics

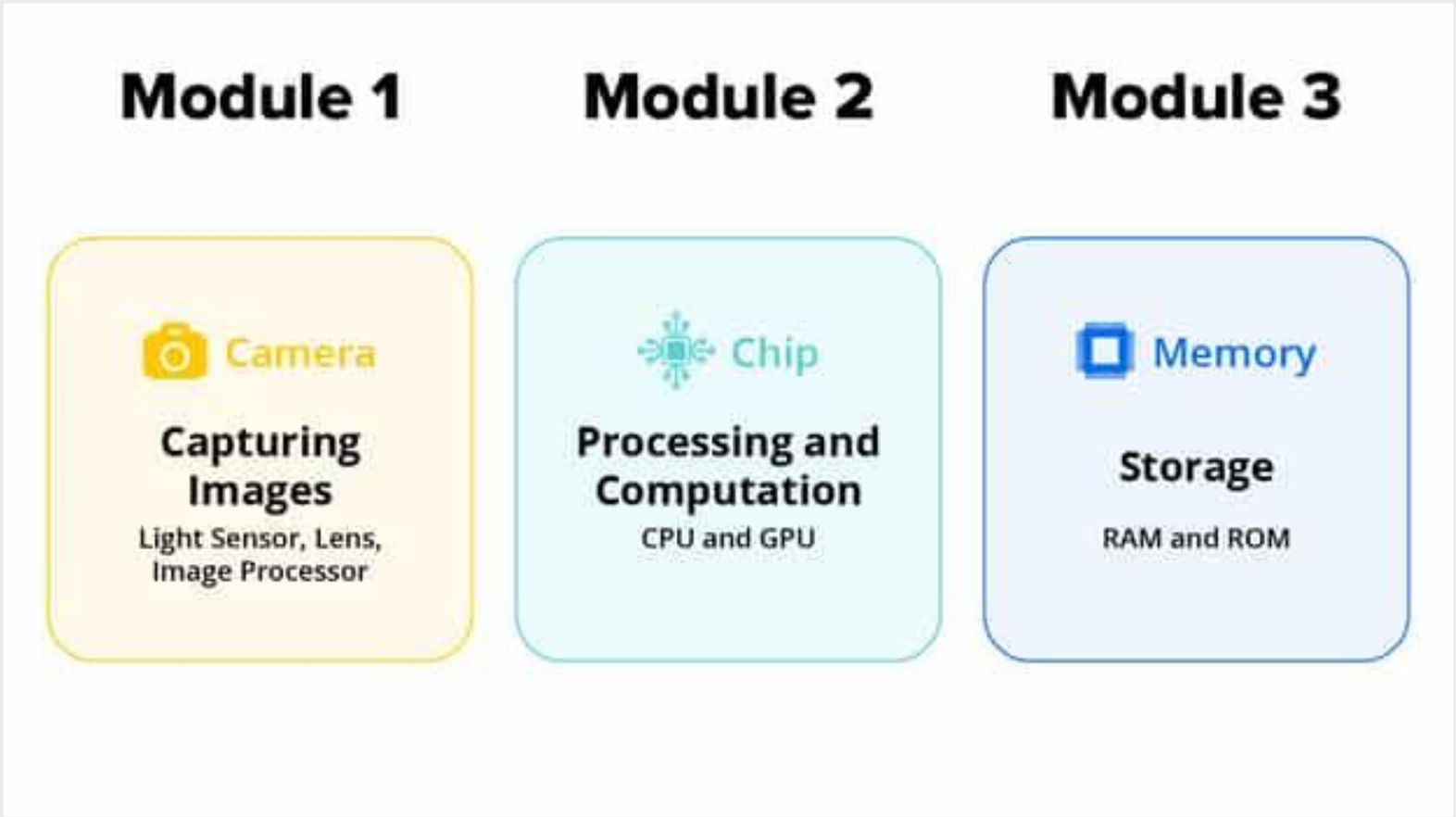
- A well-designed microservice should have high cohesion and low coupling.
  - **Cohesion** is a measure of the number of relationships that parts of a component have with each other. **High cohesion means that all of the parts that are needed to deliver the component's functionality are included in the component**.
  - **Coupling** is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.
- Each microservice should have a single responsibility i.e. it should do one thing only and it should do it well.
  - However, 'one thing only' is difficult to define in a way that's applicable to all services.
  - Responsibility does not always mean a single, functional activity.

Cohesion and Coupling characteristics



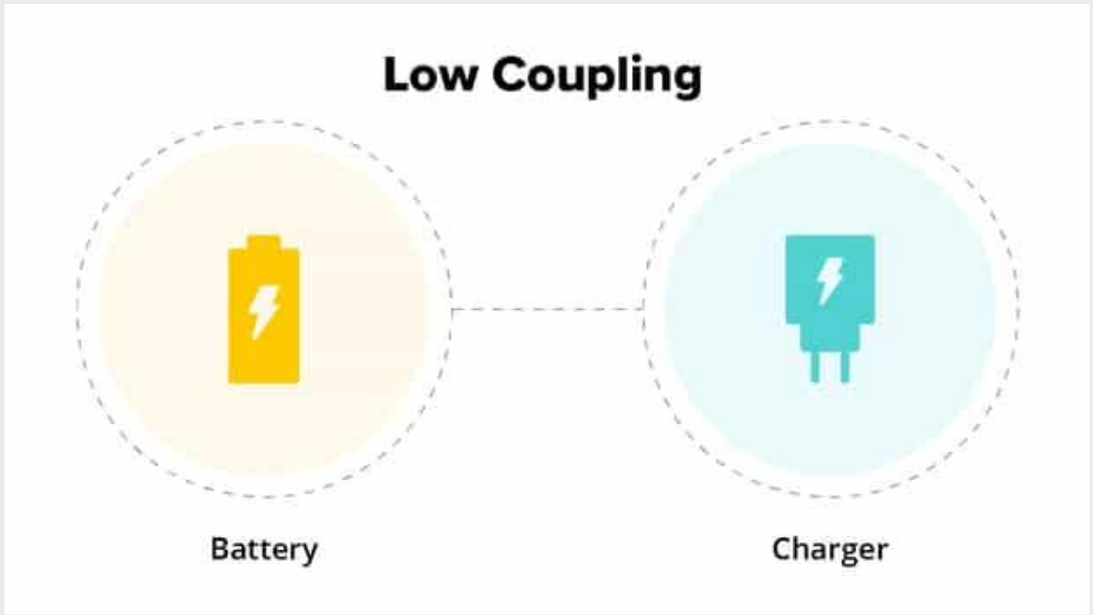
https://www.haptik.ai/tech/why-product-development-and-design-needs-cohesion-coupling

High Cohesion

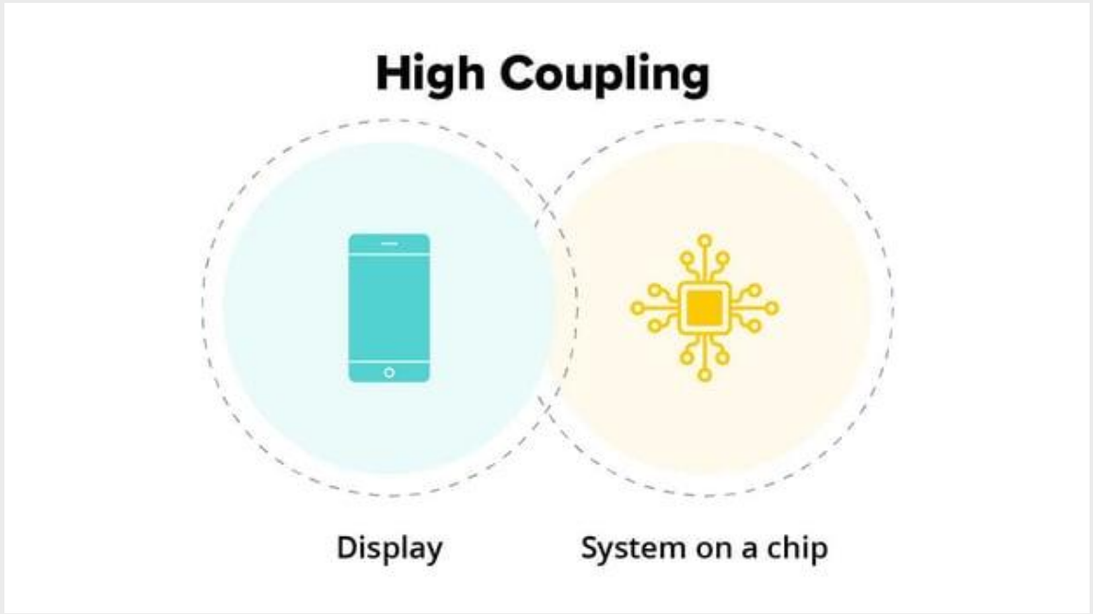


https://www.haptik.ai/tech/why-product-development-and-design-needs-cohesion-coupling

Coupling



Battery can work without charger
And
Battery or charger can be replaced or updated easily



Display cannot work without chip
And
If either is updated, the other needs update

https://www.haptik.ai/tech/why-product-development-and-design-needs-cohesion-coupling

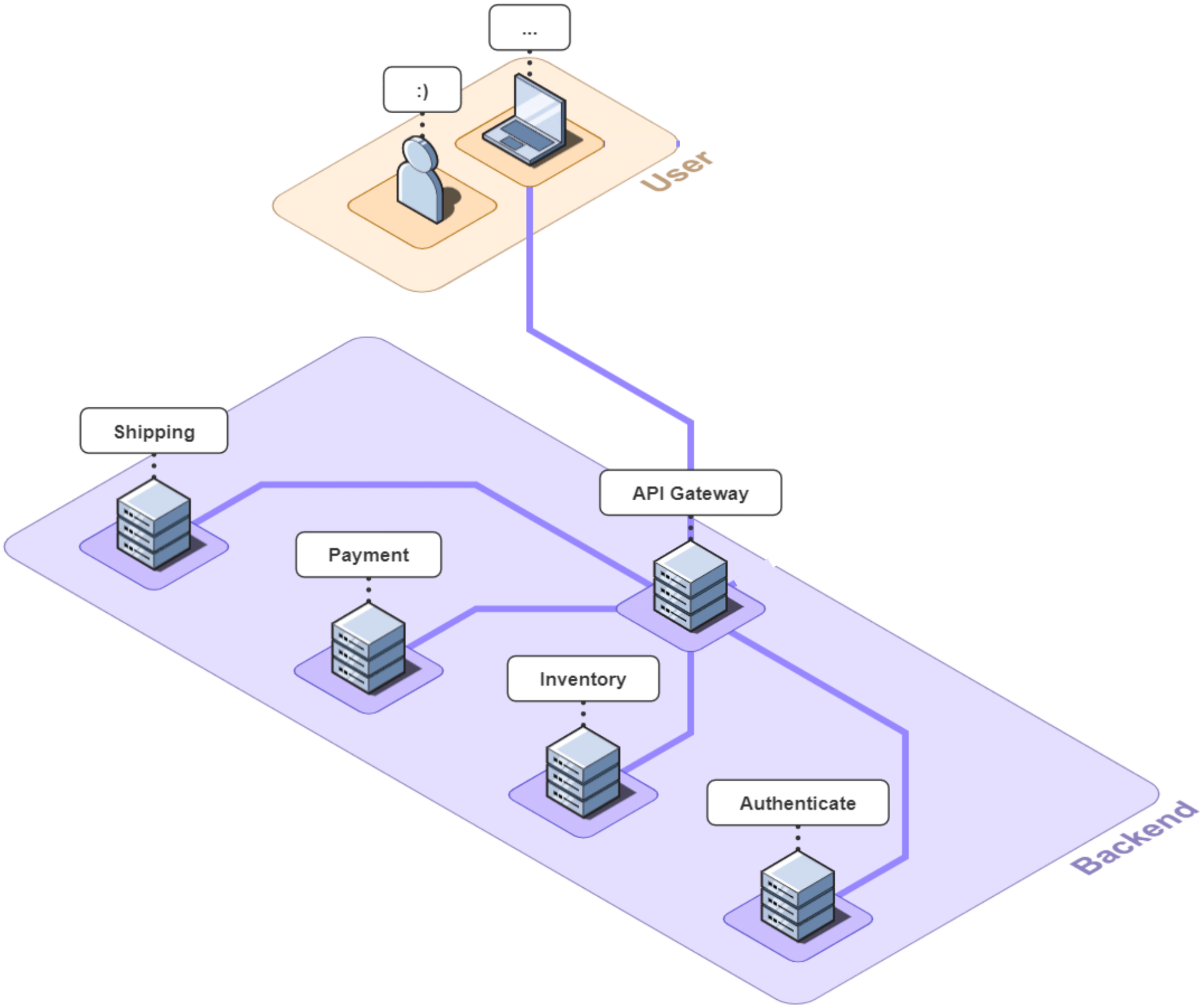
TEAM EXERCISE

- Different Presentation

Benefits of microservices architecture

- Microservices are self-contained and run in separate processes.
- In cloud-based systems, each microservice may be deployed in its own container. This means a **microservice can be stopped and restarted without affecting other parts of the system**.
- If the demand on a service increases, **service replicas can be quickly created and deployed**. These do not require a more powerful server so '**scaling-out' is, typically, much cheaper than 'scaling up**'.

API Gateway


...
:)
User
Shipping
API Gateway
Payment
Inventory
Authenticate
Backend

API Gateway

- Main Functionalities:
  - Gateway!
  - Rate limiter
  - Error Handling
  - Cache
- Either Commercial (third party)
- Or customized

API Gateway Customized options: Reverse Proxy

- Reverse Proxy in Nginx (Pros & Cons)?

```
http
{
    server
    {
        listen 80;
        server_name api-gateway.com;

        location /service1/
        {
            proxy_pass http://service1-api.com;
        }

        location /service2/
        {
            proxy_pass http://service2-api.com;
        }
    }
}
```

API Gateway Customized options: Fully Customized (Approach 1)

Fully Customized in Node.js

```
const express = require('express’)
const httpProxy = require('express-http-proxy’)
const app = express()
const userServiceProxy = httpProxy('https://user-service’)
// Authentication
app.use((req, res, next) => {
   // TODO: my authentication logic next()
})
// Proxy request
app.get('/users/:userId', (req, res, next) => {
   userServiceProxy(req, res, next)
})
```

API Gateway Customized options: Fully Customized (Approach 2)

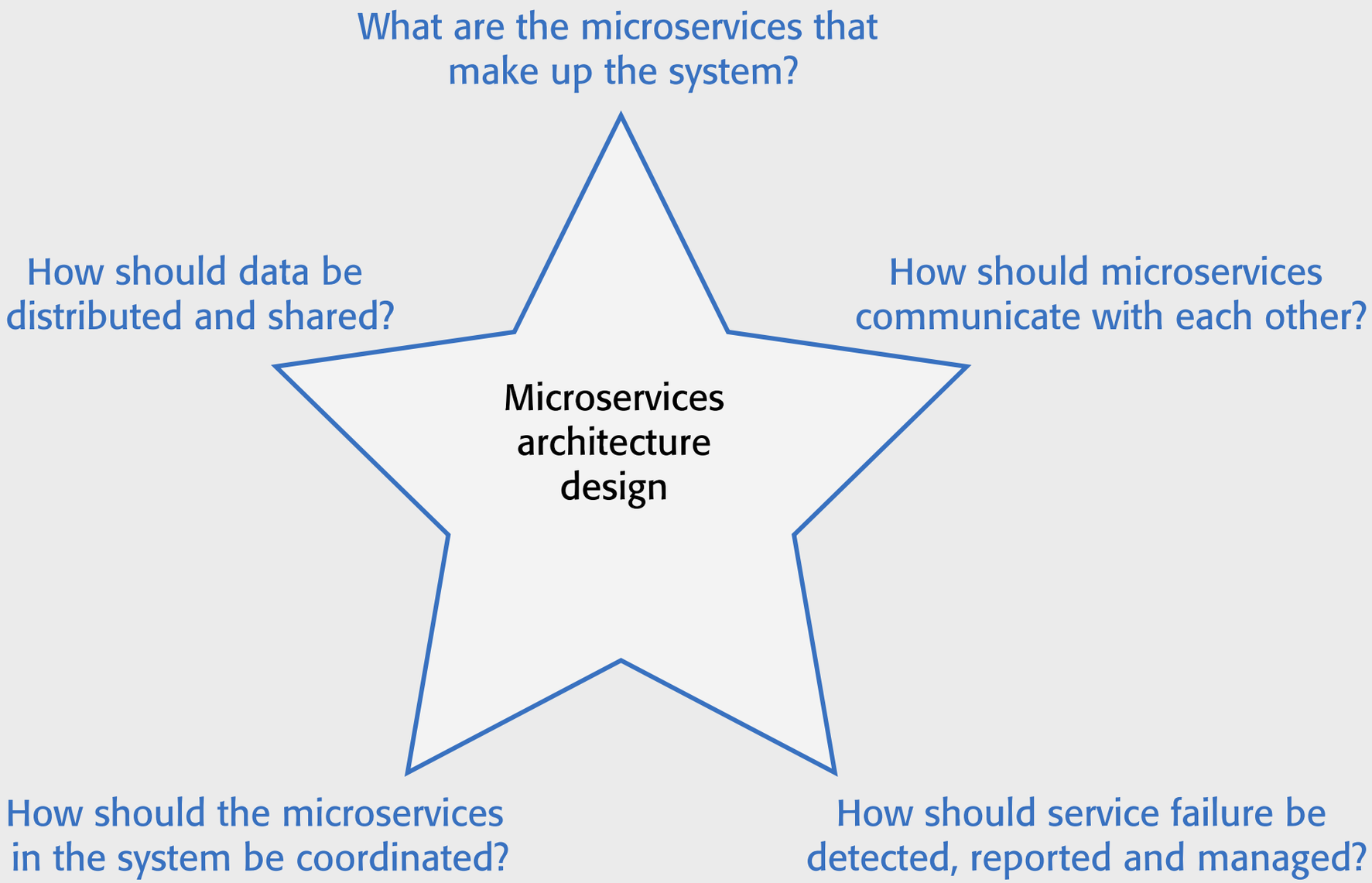
Fully Customized in Node.js

```
const express = require('express')
const request = require('request-promise-native')
const app = express()

// Resolve: GET /users/me
app.get('/users/me', async (req, res) => {
    const userId = req.session.userId
    const uri = `https://user-service/users/${userId}`
    const user = await request(uri)
    res.json(user)
})
```

Big Problem??!

Microservices architecture - key design questions



Not easy!!

But there are guidelines

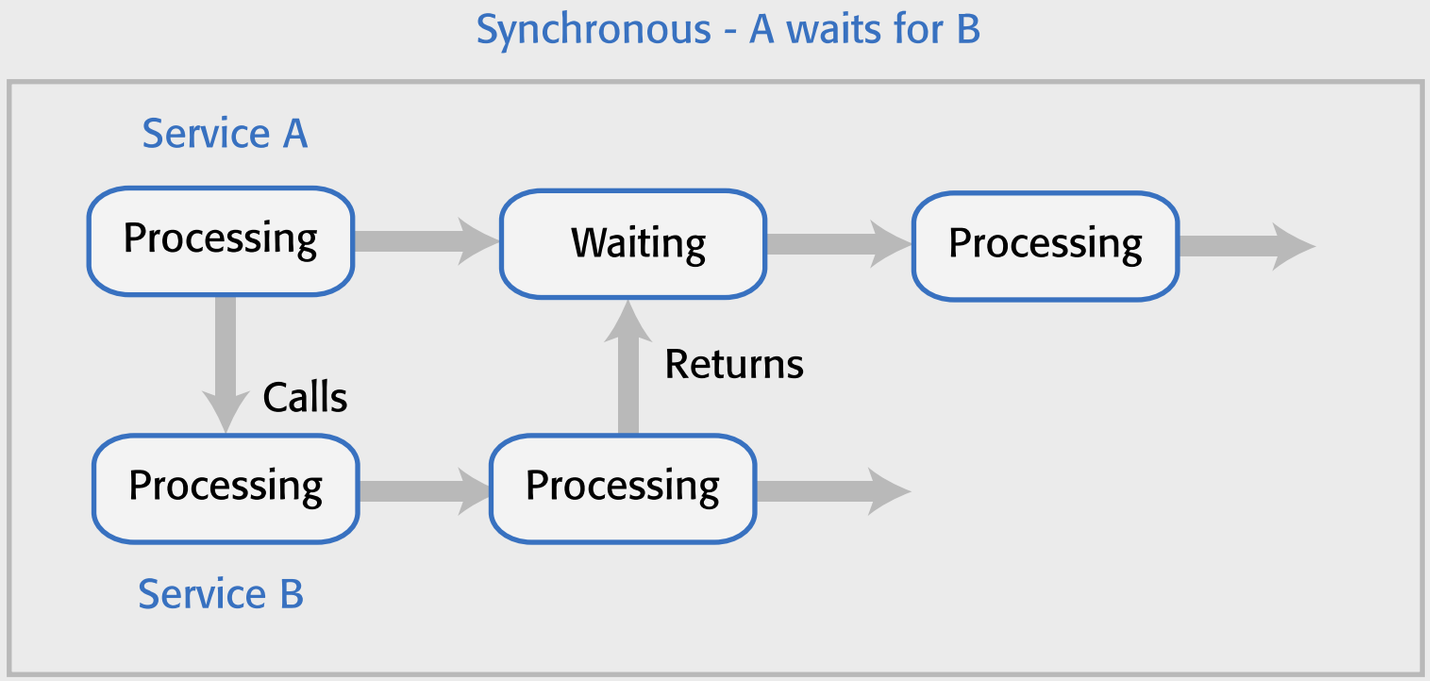
How to arch: Decomposition guidelines

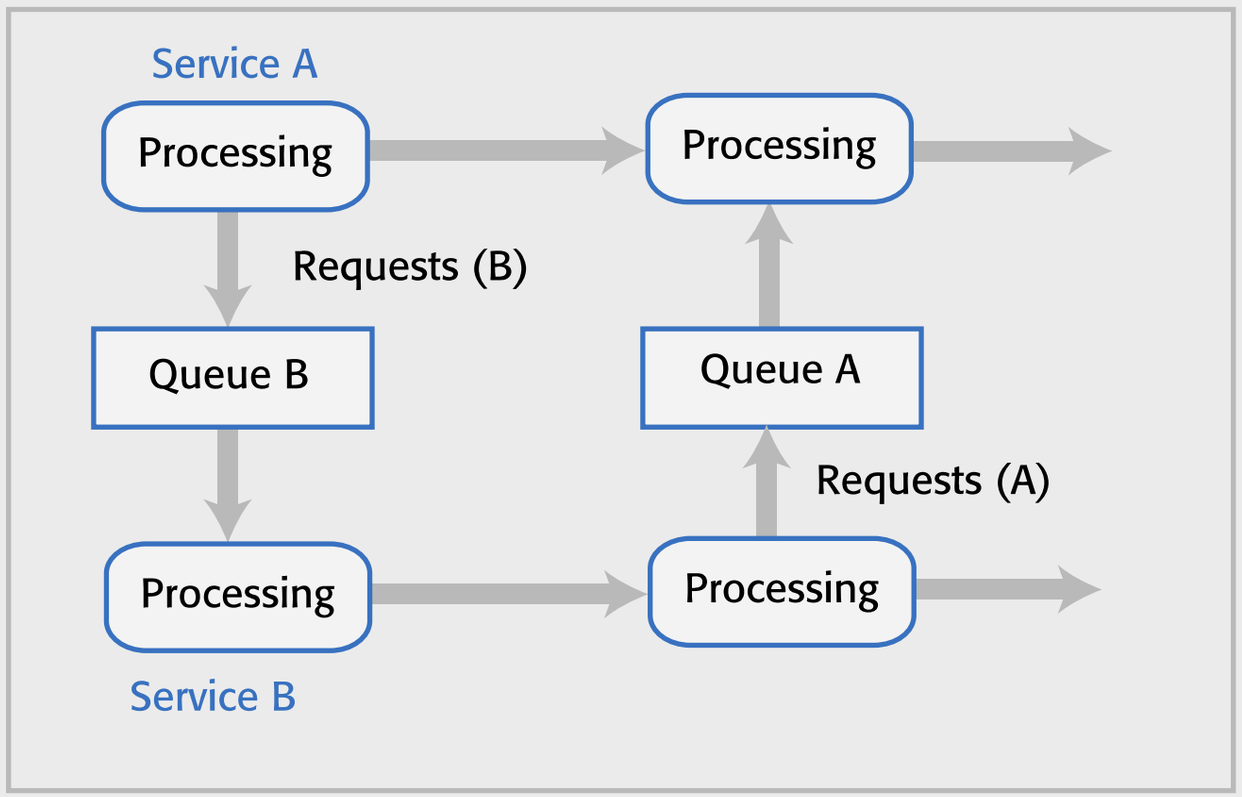
- **Balance fine-grain functionality and system performance**
  - Single function services vs. Multiple function services.
- Follow the '**common closure principle**'
  - Elements of a system that are **likely to be changed at the same time** should be located within the same service. Most new and changed requirements should therefore only affect a single service.
- **Associate services with business capabilities**
  - A business capability is a **discrete area of business functionality that is the responsibility of an individual or a group**. You should identify the services that are required to support each business capability.
- Design services so that they **only have access to the data that they need**
  - If there is an overlap between the data used by different services, you need a mechanism to propagate data changes to all services using the same data.

Service communications

- Services communicate by exchanging messages that include information about the originator of the message, as well as the data that is the input to or output from the request.
- When you are designing a microservices architecture, you have to establish a standard for communications that all microservices should follow. Some of the key decisions that you have to make are
  - should service interaction be synchronous or asynchronous?
  - should services communicate directly or via message broker middleware?
  - what protocol should be used for messages exchanged between services?

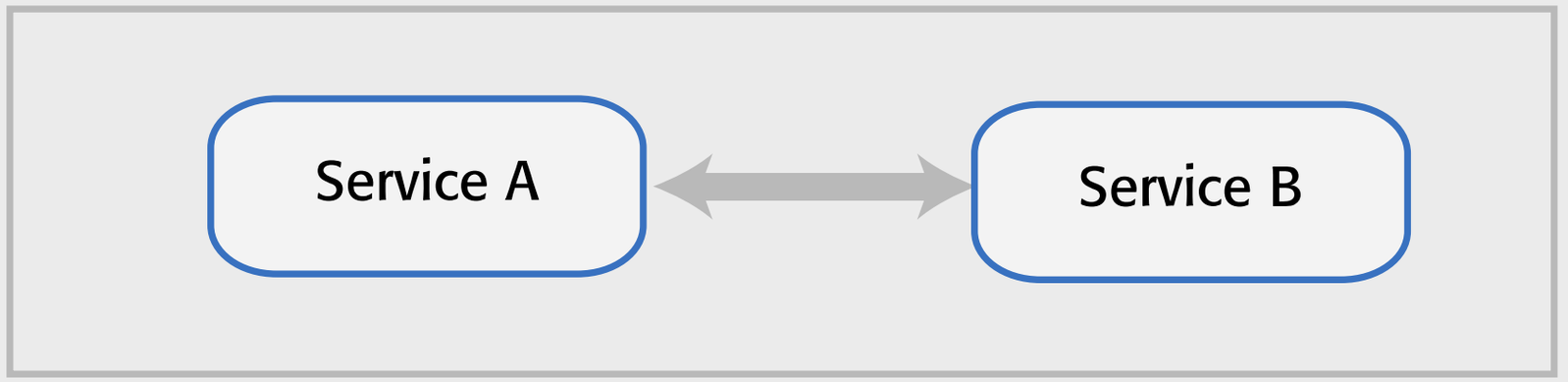
Synchronous and asynchronous microservice interaction





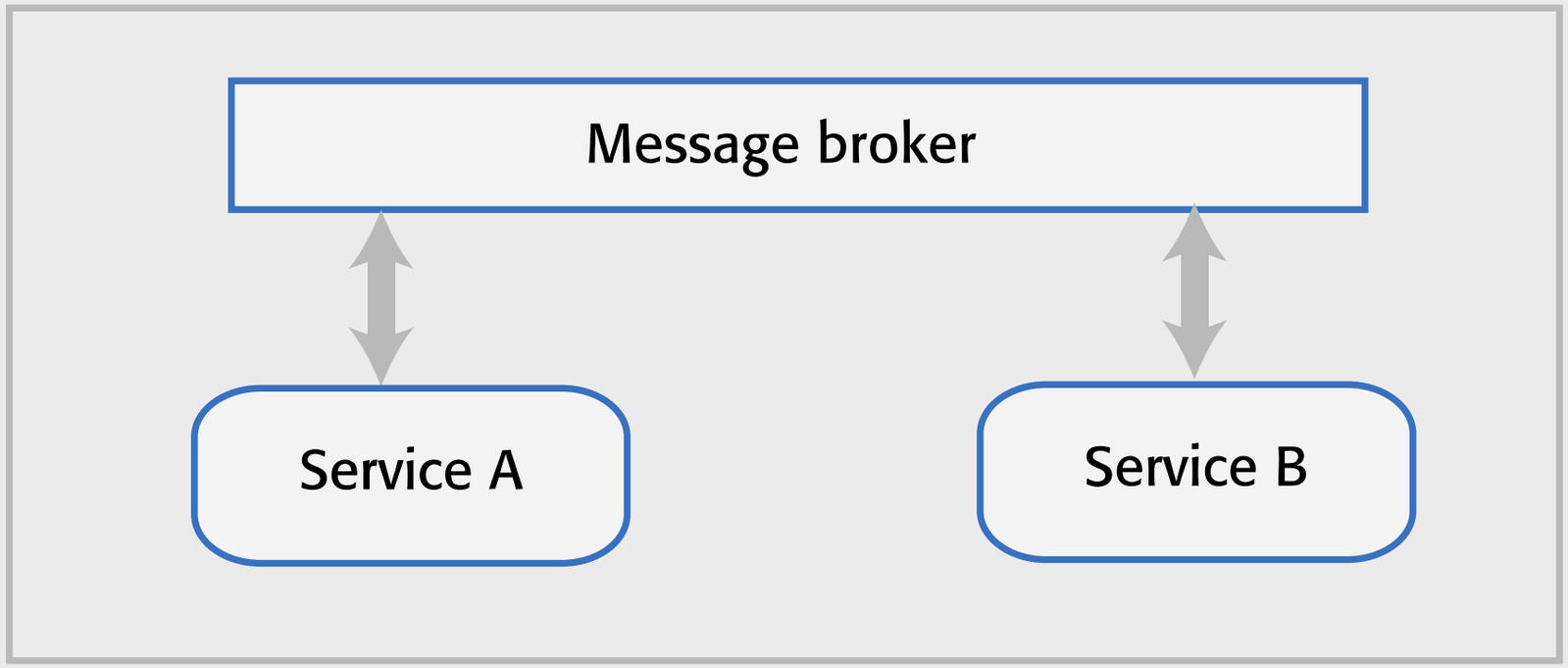
Direct and indirect service communication

Direct communication - A and B send messages to each other

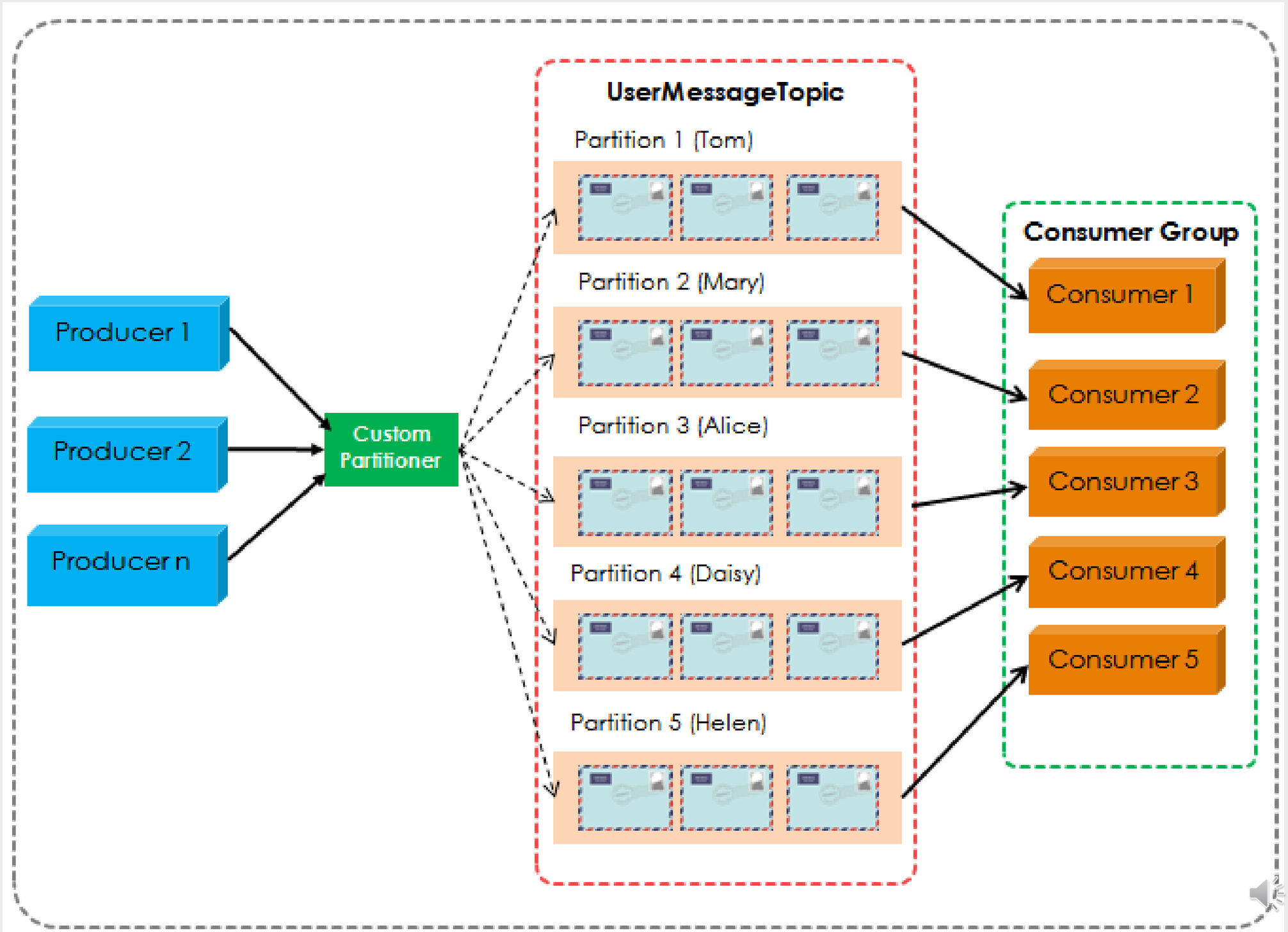


Indirect communication - A and B communicate through a message broker

Commercial Examples:



Example Kafka

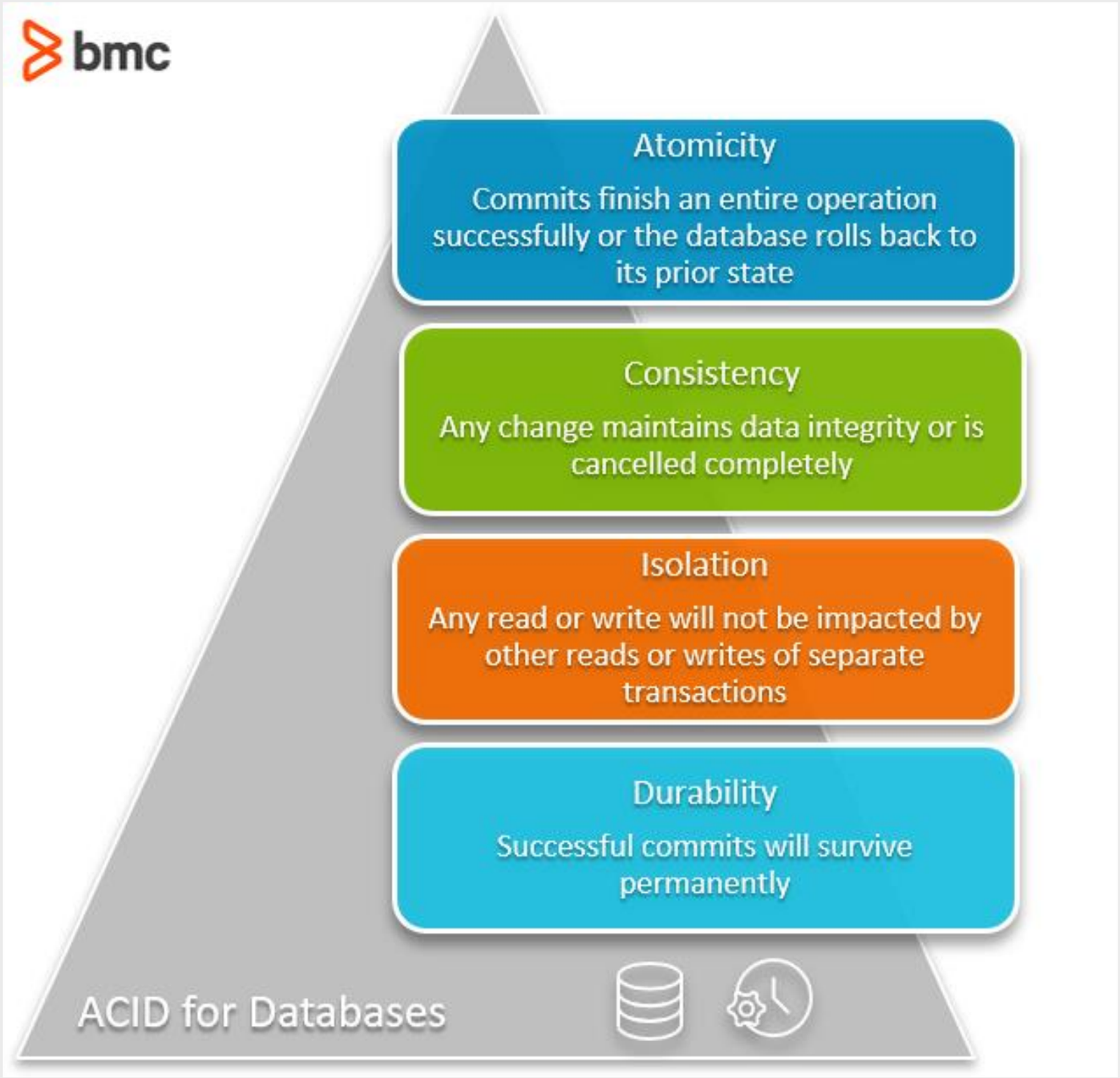


https://howtoprogram.xyz/2016/06/04/write-apache-kafka-custom-partitioner/

Microservice data design

Quick Review: ACID

Not always (easily) feasible in Micro-services System



https://www.bmc.com/blogs/acid-atomic-consistent-isolated-durable/

Microservice data design

- You should isolate data within each system service with **as little data sharing as possible**.
- If data sharing is unavoidable, you should design microservices so that most sharing is ‘read-only’, with a **minimal number of services responsible for data updates**.
- If services are **replicated** in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Why do we replicate database?

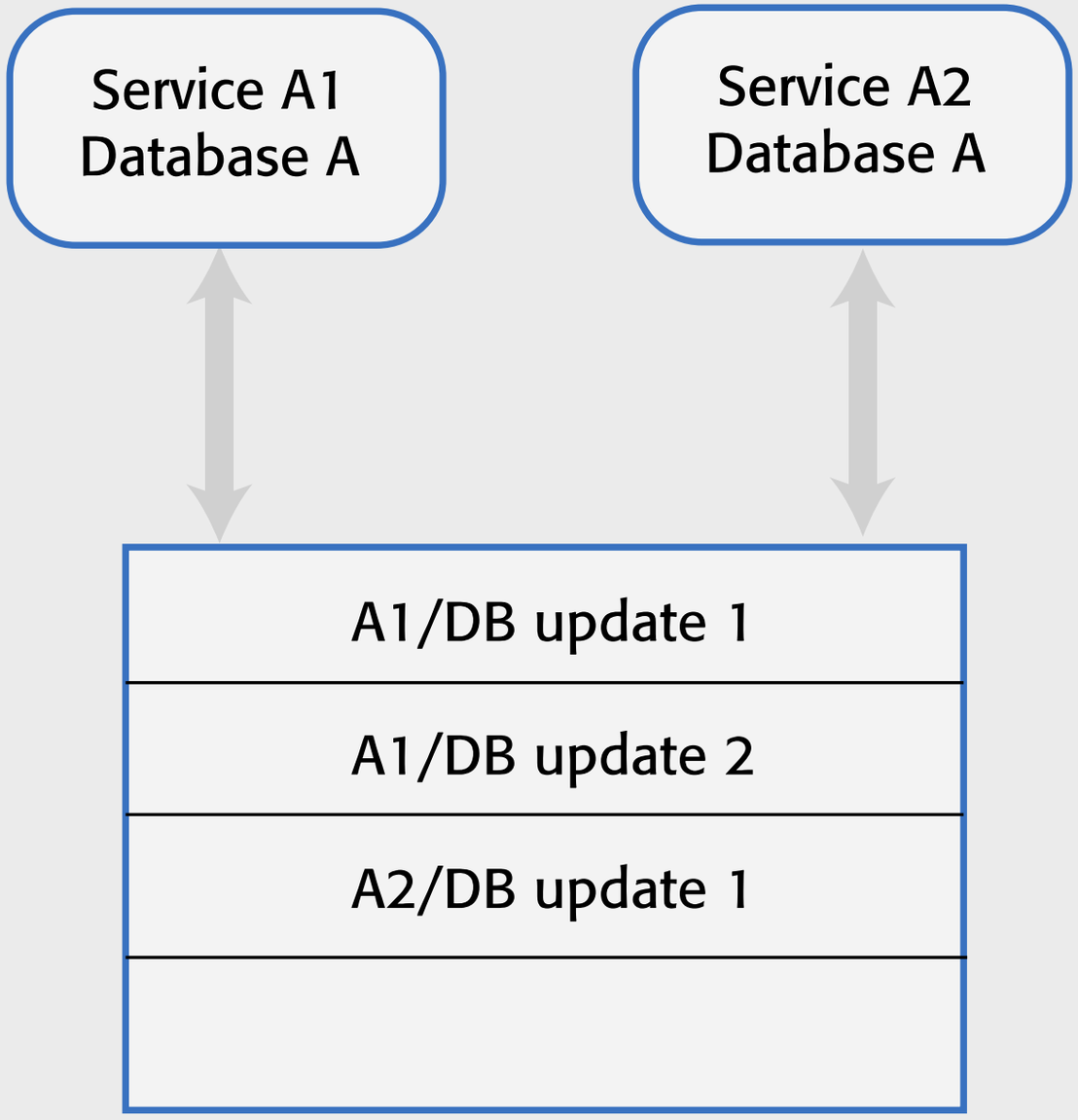
Inconsistency management

- An ACID transaction bundles a set of data updates into a single unit so that either all updates are completed or none of them are. **ACID transactions are impractical** in a microservices architecture.
- The databases used by different microservices or microservice replicas need **NOT** be completely consistent all of the time.
- Dependent data inconsistency
  - The actions or failures of one service can cause the data managed by another service to become inconsistent.
    - Ex: Payment failed, but the order service never knew!
- Replica inconsistency
  - There are several replicas of the same service that are executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases ‘eventually consistent’ so that all replicas are working on the same data.
    - Ex: Service 1 changed user data in db replica 1 and service 2 changed the same data in db replica 2, we need to make sure EVENTUALLY they are the same.

Eventual consistency

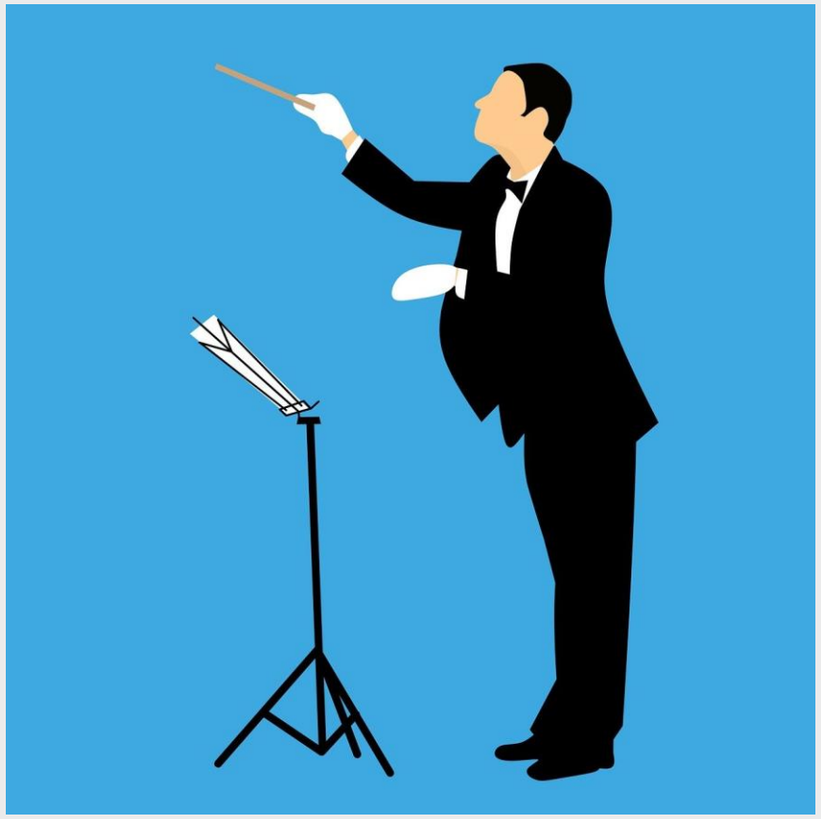
- Eventual consistency is a situation where the system guarantees that the databases will eventually become consistent.
- You can implement eventual consistency by maintaining a transaction log.
- When a database change is made, this is recorded on a ‘pending updates’ log.
- Other service instances look at this log, update their own database and indicate that they have made the change.

Using a pending transaction log



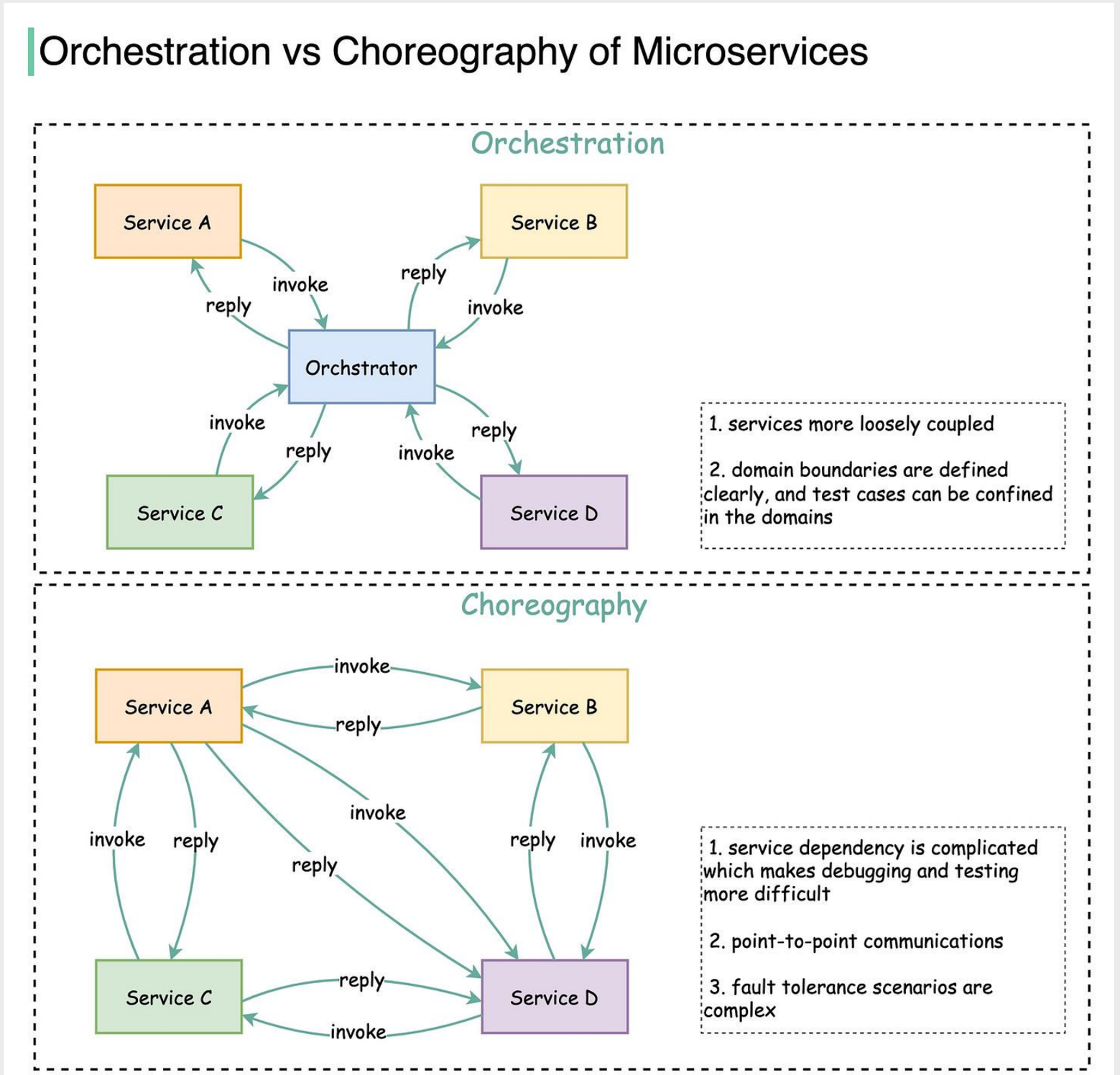
How do we coordinate service? Flow of work!

Workflow implementation methods: Orchestration and choreography



https://pixabay.com/illustrations/maestro-conductor-orchestra-music-3020019/

Workflow implementation methods: Orchestration and choreography



https://blog.bytebytego.com/p/orchestration-and-choreography

Failures in Micro-services

Failure types in a microservices system

Internal service failure

These are conditions that are **detected by the service and can be reported to the service client in an error message**. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.

External service failure

These failures have an external cause, which affects the availability of a service. **Failure may cause the service to become unresponsive and actions have to be taken to restart the service**. ***Ex: problem in another service, or connectivity problem to other service.***

Service performance failure

The performance of the service **degrades to an unacceptable level**. This may be due to a heavy load or an internal problem with the service. **External service monitoring** can be used to detect performance failures and unresponsive services.

To detect failures: Timeouts and circuit breakers

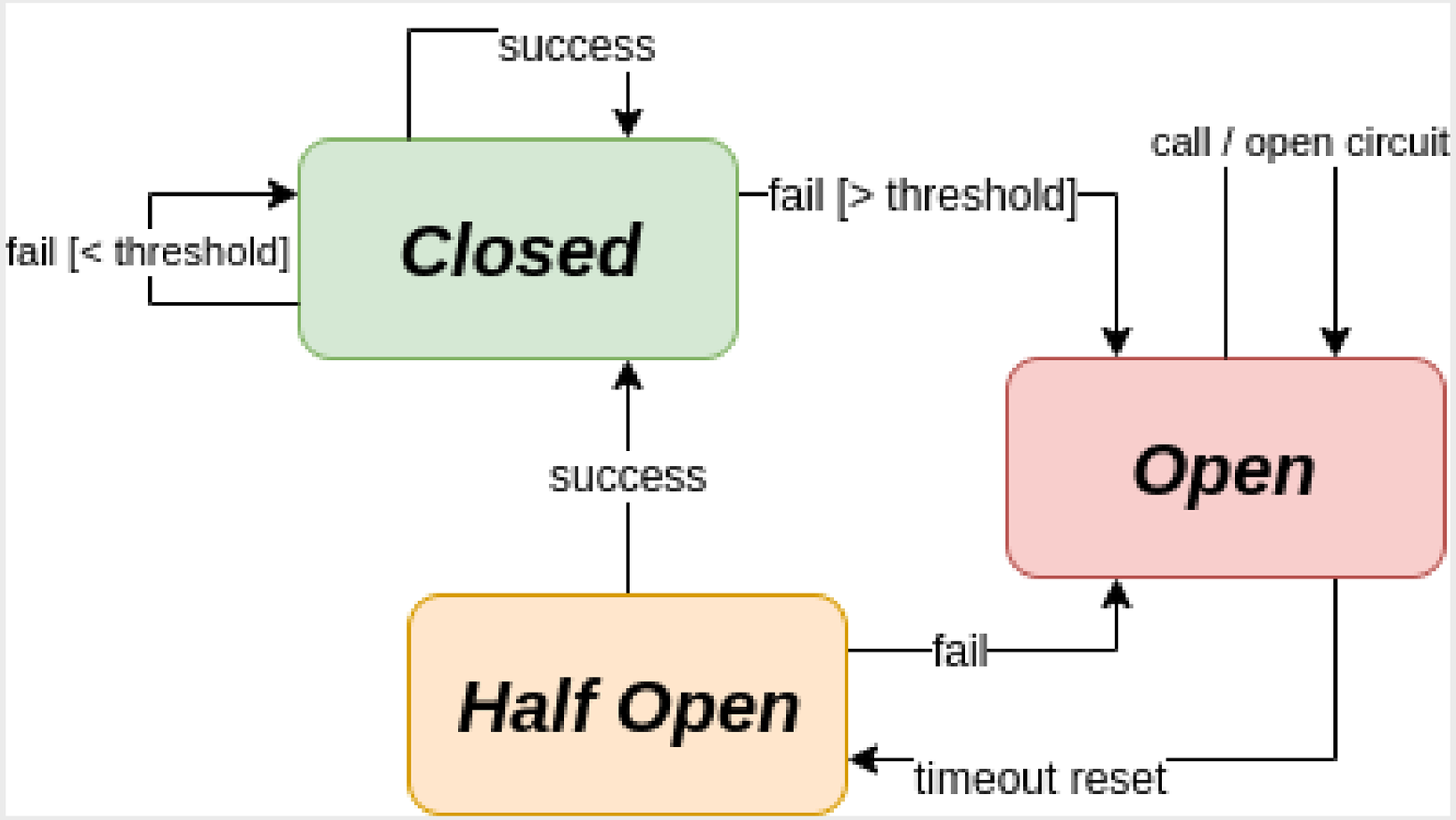
- A timeout is a counter that this associated with the service requests and starts running when the request is made.

- Once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly.

- The problem with the timeout approach is **that every service call to a 'failed service' is delayed by the timeout value so the whole system slows down**.

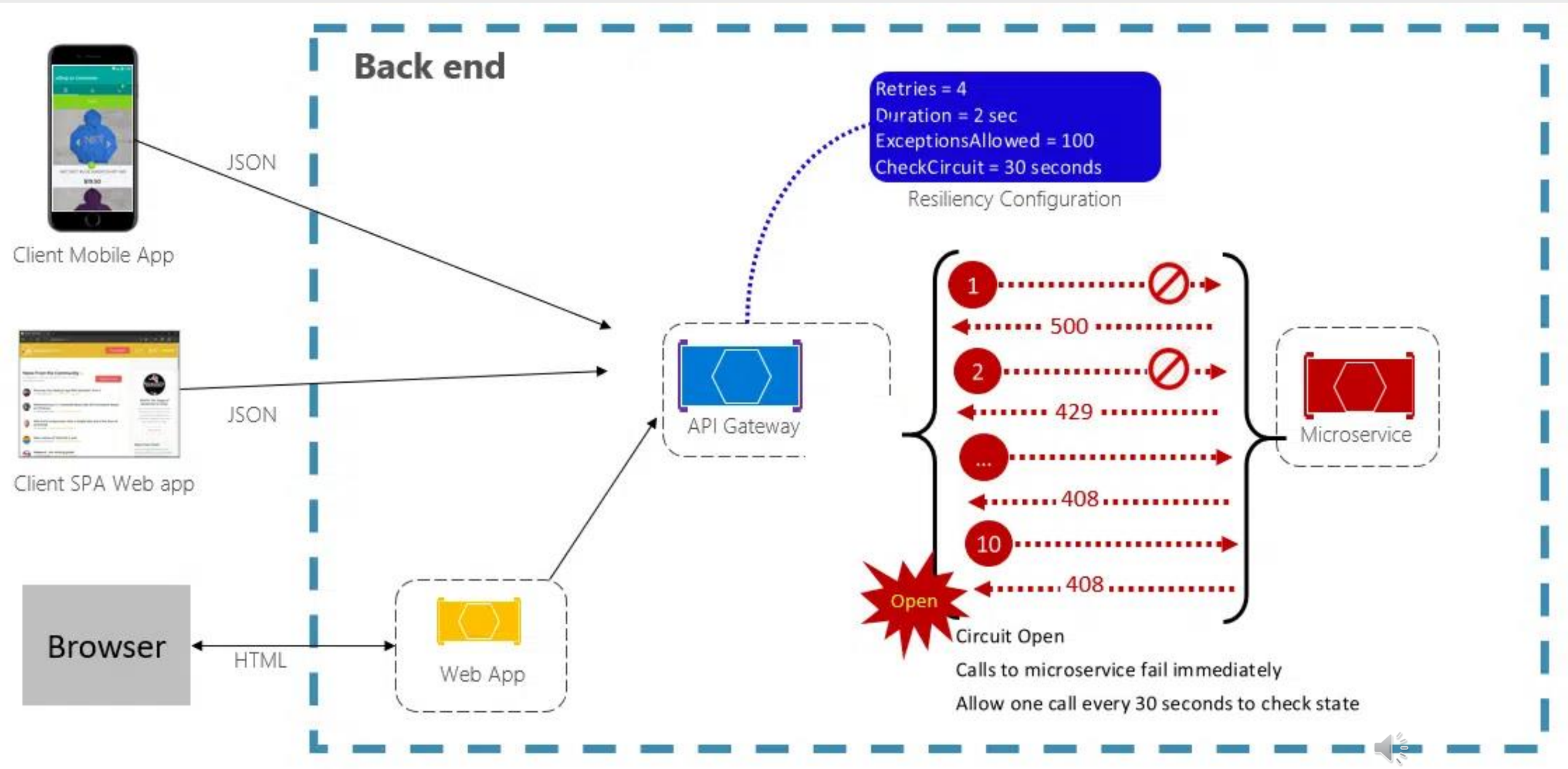
- Instead of using timeouts explicitly when a service call is made, we can use a circuit breaker. Like an electrical circuit breaker, this **immediately denies access to a failed service without the delays associated with timeouts**.

Circuit Breaker



Note: Analogy to real life scenario:

Circuit Breaker



RESTful services [Self Study]

- Already known during the project!

RESTful services

- The REST (REpresentational State Transfer) architectural style is based on the idea of transferring representations of digital resources from a server to a client.
  - You can think of a resource as any chunk of data such as credit card details, an individual's medical record, a magazine or newspaper, a library catalogue, and so on.
  - Resources are accessed via their unique URI and RESTful services operate on these resources.
- This is the fundamental approach used in the web where the resource is a page to be displayed in the user's browser.
  - An HTML representation is generated by the server in response to an HTTP GET request and is transferred to the client for display by a browser or a special-purpose app.

RESTful service principles

Use HTTP verbs

The basic methods defined in the HTTP protocol (GET, PUT, POST, DELETE) must be used to access the operations made available by the service.

Stateless services

Services must never maintain internal state. As I have already explained, microservices are stateless so fit with this principle.

URI addressable

All resources must have a URI, with a hierarchical structure, that is used to access sub-resources.

Use XML or JSON

Resources should normally be represented in JSON or XML or both. Other representations, such as audio and video representations, may be used if appropriate.

RESTful service operations

Create
Implemented using HTTP POST, which creates the resource with the given URI. If the resource has already been created, an error is returned.

Read
Implemented using HTTP GET, which reads the resource and returns its value. GET operations should never update a resource so that successive GET operations with no intervening PUT operations always return the same value.

Update
Implemented using HTTP PUT, which modifies an existing resource. PUT should not be used for resource creation.

Delete
Implemented using HTTP DELETE, which makes the resource inaccessible using the specified URI. The resource may or may not be physically deleted.

Road information system

- Imagine a system that maintains information about incidents, such as traffic delays, roadworks and accidents on a national road network. This system can be accessed via a browser using the URL:
  - https://trafficinfo.net/incidents/
- Users can query the system to discover incidents on the roads on which they are planning to travel.
- When implemented as a RESTful web service, you need to design the resource structure so that incidents are organized hierarchically.
  - For example, incidents may be recorded according to the road identifier (e.g. A90), the location (e.g. stonehaven), the carriageway direction (e.g. north) and an incident number (e.g. 1). Therefore, each incident can be accessed using its URI:
  - https://trafficinfo.net/incidents/A90/stonehaven/north/1

Incident description

Incident ID: A90N17061714391

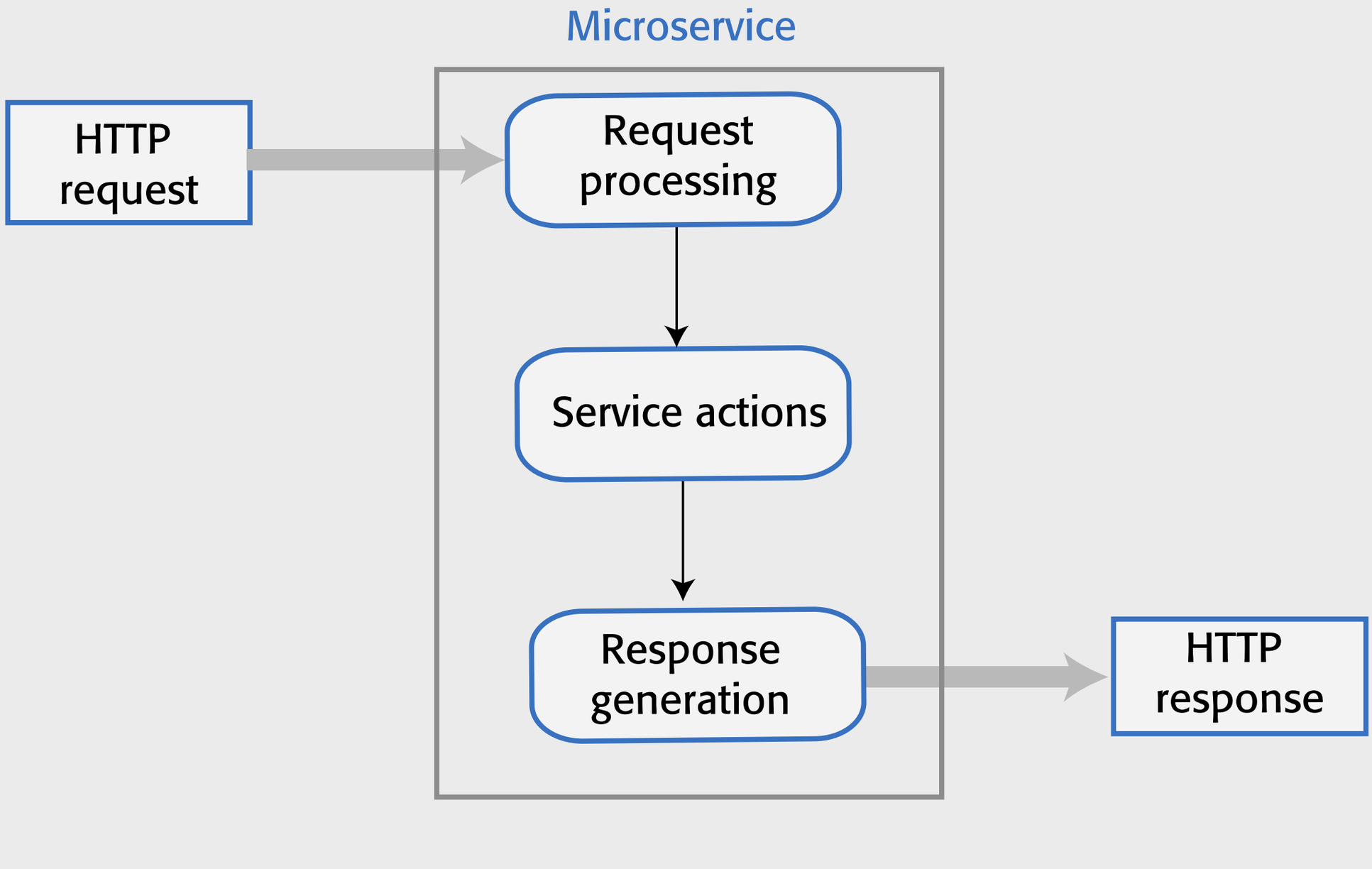
Date: 17 June 2017

Time reported: 1439

Severity: Significant

Description: Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes

HTTP request and response processing



request and response message organization

REQUEST

[HTTP verb]	[URI]	[HTTP version]
[Request header]		
[Request body]		

RESPONSE

[HTTP version]	[Response code]
[Response header]	
[Response body]	

JSON

{
id: “A90N17061714391”,
“date”: “20170617”,
“time”: “1437”,
“road_id”: “A90”,
“place”: “Stonehaven”,
“direction”: “north”,
“severity”: “significant”,
“description”: “Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.”
}

XML and JSON descriptions

XML

<id>
A90N17061714391
</id>
<date>
20170617
</date>
<time>
1437
</time>
…
<description>Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.
</description>

A GET request and the associated response

REQUEST

GET	incidents/A90/stonehaven/	HTTP/1.1

Host: trafficinfo.net
...
Accept: text/json, text/xml, text/plain
Content-Length: 0

RESPONSE

HTTP/1.1	200

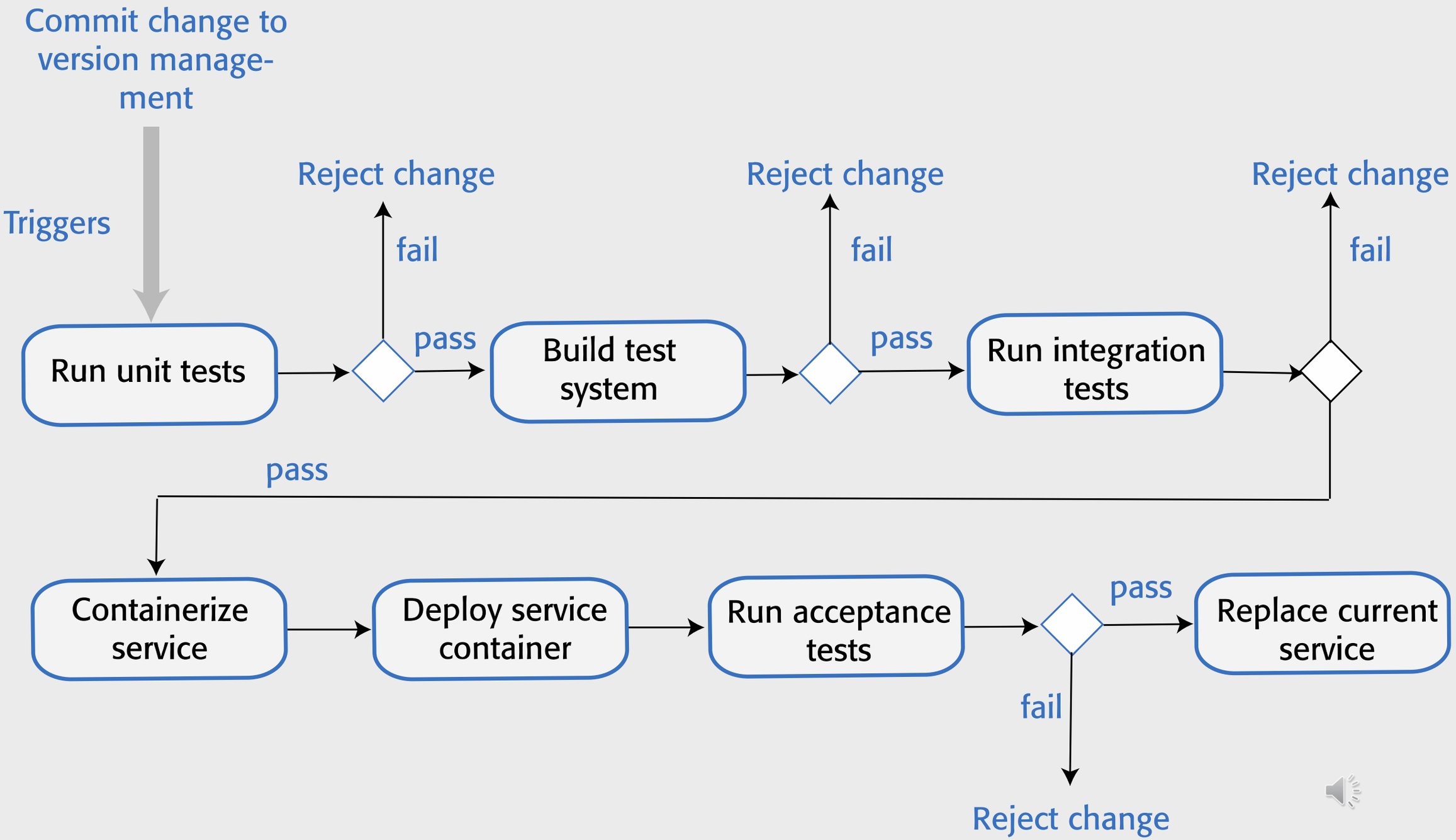
...
Content-Length: 461
Content-Type: text/json

{
"number": "A90N17061714391",
"date": "20170617",
"time": "1437",
"road_id": "A90",
"place": "Stonehaven",
"direction": "north",
"severity": "significant",
"description": "Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes."
}
{
"number": "A90S17061713001",
"date": "20170617",
"time": "1300",
"road_id": "A90",
"place": "Stonehaven",
"direction": "south",
"severity": "minor",
"description": "Grass cutting on verge. Minor delays"
}

Service deployment

- After a system has been developed and delivered, it has to be deployed on servers, monitored for problems and updated as new versions become available.
- When a system is composed of tens or even hundreds of microservices, deployment of the system is more complex than for monolithic systems.
- The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is no ‘standard’ deployment configuration for all services.
- It is now normal practice for microservice development teams to be responsible for deployment and service management as well as software development and to use continuous deployment.
- Continuous deployment means that as soon as a change to a service has been made and validated, the modified service is redeployed.

A continuous deployment pipeline



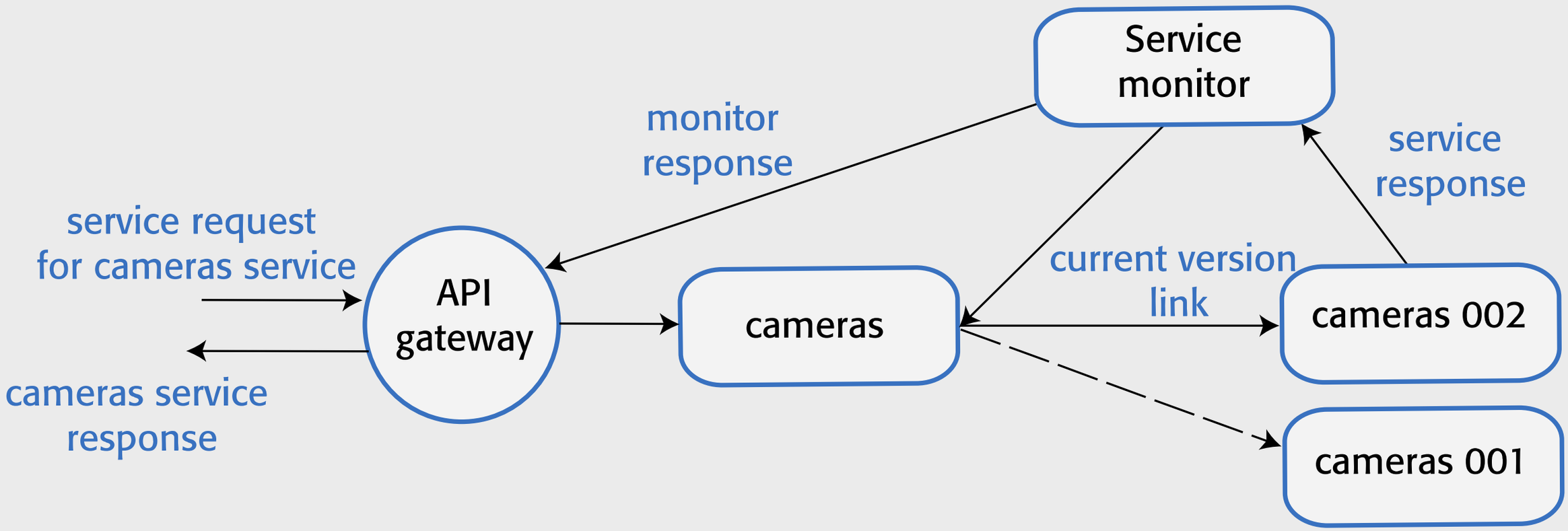
Simple (unrealistic) Pipeline for nodejs CD

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {                         git 'https://github.com/your/repository.git'}
        }
        stage('Build') {
            steps { sh 'npm install'
                sh 'npm run build'}
        }
        stage('Unit Test') {
            steps {                 sh 'npm test'           }
        }
        stage('Integration Test') {
            steps {                                 sh 'npm run integration-test'
        }
    }
    stage('Containerization') {
        steps {
            // Build and push Docker image
            script {
                docker.build('your-docker-image:latest')
                docker.withRegistry('https://your-docker-registry/', 'docker-credentials') {
                    docker.image('your-docker-image:latest').push()
                }
            }
        }
    }
```

```
    stage('Acceptance Test') {
        steps {
            // Run acceptance tests against the deployed Docker container
            sh 'npm run acceptance-test'
        }
    }
    stage('Replace Service') {
        steps {
            // Deploy the new Docker container and replace the currently running service
            script {
            }
        }
    }
}
    post {
        success {
            // Notify or perform additional actions on pipeline success
            echo 'Pipeline succeeded!'
        }
        failure {
            // Notify or perform additional actions on pipeline failure
            echo 'Pipeline failed!'
        }
    }
}
```

Versioned services



Final Questions: When to use Micro-services? Is it a “one size fits all” solution

Key points 1

- A microservice is an independent and self-contained software component that runs in its own process and communicates with other microservices using lightweight protocols.
- Microservices in a system can be implemented using different programming languages and database technologies.
- Microservices have a single responsibility and should be designed so that they can be easily changed without having to change other microservices in the system.
- Microservices architecture is an architectural style in which the system is constructed from communicating microservices. It is well-suited to cloud based systems where each microservice can run in its own container.
- The two most important responsibilities of architects of a microservices system are to decide how to structure the system into microservices and to decide how microservices should communicate and be coordinated.

Key points 2

- Communication and coordination decisions include deciding on microservice communication protocols, data sharing, whether services should be centrally coordinated, and failure management.
- The RESTful architectural style is widely used in microservice-based systems. Services are designed so that the HTTP verbs, GET, POST, PUT and DELETE, map onto the service operations.
- The RESTful style is based on digital resources that, in a microservices architecture, may be represented using XML or, more commonly, JSON.
- Continuous deployment is a process where new versions of a service are put into production as soon as a service change has been made. It is a completely automated process that relies on automated testing to check that the new version is of ‘production quality’.
- If continuous deployment is used, you may need to maintain multiple versions of deployed services so that you can switch to an older version if problems are discovered in a newly-deployed service.