L02 System Design Requirements and Software Architectures

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Today's learning goals

- **Part I:** Requirements engineering
  - Requirement types (functional/non-functional)
  - Stages in requirements engineering
  - Non-functional requirements in the cloud
- **Part II:** Software architectures in the cloud
  - Overview
  - Client-server architecture
    - Communication layers (REST and gRPC)
    - Serialization and deserialization of structured data using Protbuf
  - Monolithic architecture
  - Microservice architecture
    - Strangler pattern: From monoliths to microservices

Outline

- **Part I: Requirements engineering**
  - Requirement types (functional/non-functional)
  - Stages in requirements engineering
  - Non-functional requirements in the cloud

- **Part II:** Software architectures in the cloud

Why requirements matter in software engineering?

die Anforderung, -en

Effective gathering of requirements is critical to the success of system development projects, as it ensures that the **final product meets the needs and expectations** of stakeholders and is **delivered on time and within budget**

Requirements

Features that the system **must have** in order to be **accepted** by the client

Constraints that the system **must satisfy** in order to be **accepted** by the client

They describe a **user's view** of the system

Describe **what**, not **how**

Requirements	Not requirements
• Functionality • User interaction • Error handling • Environmental conditions	• System design • Implementation technology • Development methodology

Requirement types

- **Functional requirements:**
  - Describe the **specific tasks and functions** that a system or product must perform
  - Typically expressed in terms of **use cases or user stories**, and describe the features and functionalities of a system or product

- **Non-functional requirements:**
  - Describe the **characteristics or qualities** that the system or product must possess to meet the desired level of performance, usability, and reliability
  - Typically expressed in terms of **quality attributes**, such as system's performance, reliability, security, maintainability, etc.

Important!

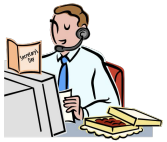
Both **functional and non-functional requirements are essential** to the success of a software project, as they help to ensure that the system **meets the needs and expectations** of its intended users

Requirements engineering

- **Requirements engineering** involves various activities such as **gathering and documenting** the requirements from stakeholders, **analyzing the requirements** to **identify potential conflicts** or **missing requirements**, **prioritizing** the requirements, and **validating** the requirements to ensure that they are **complete, consistent, and feasible**

- The **ultimate goal** is to create
  - a **clear and concise set of requirements** that accurately reflects the needs of the stakeholders
  - **provides a solid foundation for the development** of a high-quality software product

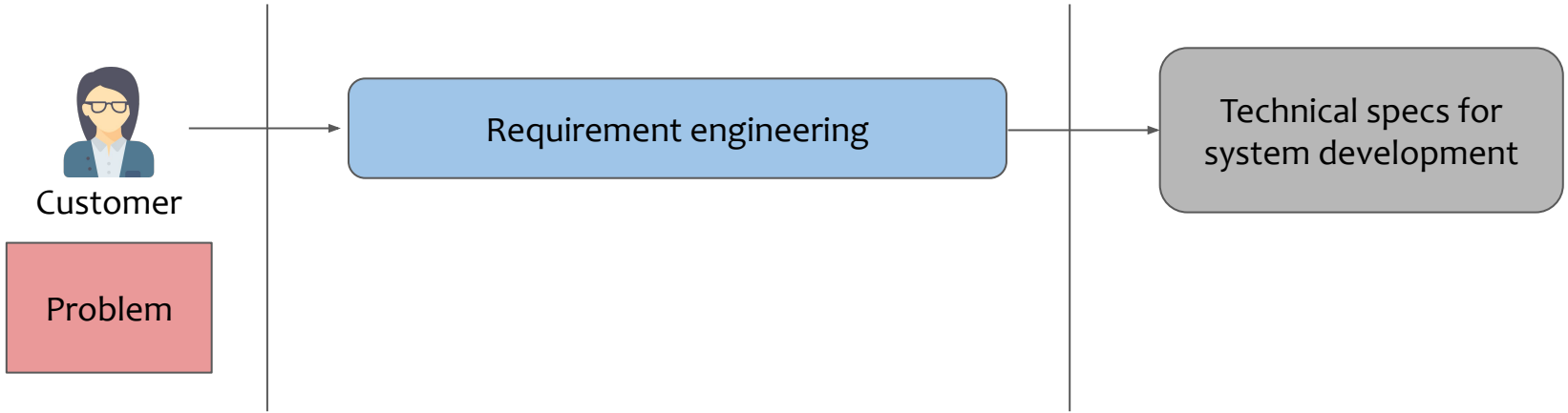

Client


Users

Software product

Requirements engineering

- **Requirements engineering** is the process of **eliciting**, **analyzing**, **documenting**, **validating**, and **managing** the requirements for a software system



Stages in requirement engineering

Customer

Problem

Requirement engineering

Elicitation

Analysis

Specification

Validation

Management

Maintenance

Technical specs for system development

May involve several iterations

Stages in requirement engineering

General Information

Tech-related Information

1. **Elicitation: Gathering requirements from stakeholders** through interviews, surveys, workshops, and other techniques
2. **Analysis: Analyzing and prioritizing requirements**, identifying dependencies, and resolving conflicts
3. **Specification: Documenting requirements** in a clear and concise manner, often using tech specs (e.g., SRS) or standard notations (e.g., UML)
4. **Validation:** Ensuring the **requirements are complete, consistent, and correct**, and that they meet the needs of stakeholders
5. **Management: Tracking changes to requirements**, communicating changes to stakeholders, and ensuring that requirements are met throughout the software or system development life cycle
6. **Maintenance ("aka long-term management"): Managing changes to requirements over time**, ensuring that they remain relevant and up-to-date

Case study: Let’s go back to our Google Maps example

Product vision:

Our overarching vision is to create a user-friendly, reliable, and feature-rich mapping and navigation application that helps users explore the world around them. This includes core functionalities like **viewing maps**, searching for locations, **getting directions**, and potentially incorporating features like **real-time traffic** updates, points of interest, and offline map capabilities.

#1: Elicitation

- The primary goal of elicitation is **to identify the requirements** that must be met by the system in order to satisfy the needs of its users and stakeholders

- There are **many techniques** that can be used to elicit requirements, including interviews, questionnaires, surveys, observation, and focus groups

- Once the requirements have been elicited, they are typically documented in a **requirements document** or a similar artifact

#1: Elicitation – Example

Goal: Discover user & stakeholder needs

Example activities:

- **User interviews** ("What's important in a map app?")
- **Surveys** ("Rank the following features in order of importance (Map View, Search, Directions, Offline Maps, Real-time Traffic, POIs).")
- **Brainstorming** (core features: map view, search, directions...)
- **Initial user stories** ("As a user, I want to search...")

#2: Analysis

- Analysis involves **examining the requirements** to understand their meaning, assess their feasibility, and identify any potential conflicts or inconsistencies

- During analysis, the **requirements are typically organized and categorized** in a structured manner, such as by **grouping related requirements** together or by creating **use cases** that describe how the system should behave in different scenarios

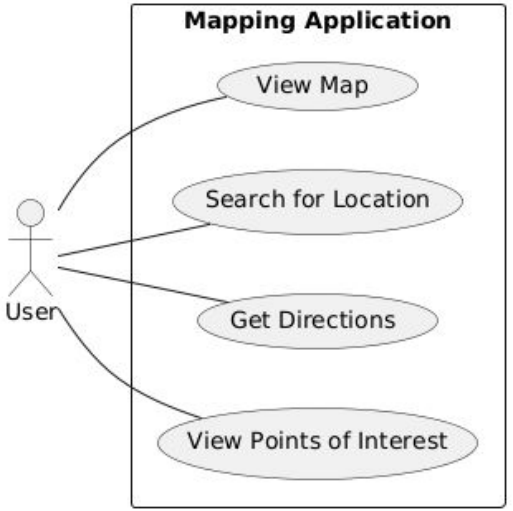
- The analysis stage also involves **determining whether the requirements are achievable** within the constraints of the project, such as the budget, schedule, and available resources

#2: Analysis – Example

Goal: Refine & structure gathered information

Example activities:

- **Clarify and combine similar requests** ("I need to find places," "I want to look up addresses")
- **Categorize** (Functional: search; Non-Functional: user-friendly)
- **Prioritize** (e.g., core features first)
- **Initial modeling via a use-case UML diagram** (e.g., basic use case for "Search")



#3: Specification

- Requirements have been elicited and analyzed are **formally documented and communicated to stakeholders**
  - The purpose of specification is to create a clear, concise, and unambiguous description of the requirements

- Specifications also captures **any constraints or limitations** that must be taken into account, and any specific requirements related to data or interfaces

- There are many ways to **document requirements**, such as
  - UML use-case diagrams
  - User stories
  - Functional/non-functional requirements
  - System specifications

#3: Specification – Example

Goal: Formally document the analyzed requirements

- **Software Requirement Specifications (SRS)**
  - Containing Functional Requirements **(FR)** and Non-functional Requirements **(NFR)**

Example:

FR001: Search for location

- **Description:** The system shall allow users to search for locations
- **Priority:** High
- **Details:**
  - Supports address and Point of Interest (POI) search
  - Displays matching locations on the map
- **Acceptance Criteria:**
  - Relevant search results appear within 2 seconds of query
  - Selecting a result centers the map on the correct location

#4: Validation

- Verification and validation are critical to ensuring that the requirements for a system are accurate, complete, and consistent

Before the system is built

- **Verification** refers to the process of checking that the requirements have been correctly captured, and that they accurately reflect the needs/expectations of the stakeholders
  - By verifying the requirements before the system is built, potential issues can be identified and corrected early in the process
  - Verification can be done by reviewing the requirements document with stakeholders, or checking with any relevant industry standards or regulations
- **Validation,** on the other hand, refers to the process of ensuring that the system meets the requirements that have been specified
  - Validation ensures that the system behaves as expected and meets the needs of its users
  - Validation can be carried out through a range of methods, such as testing, simulations, prototypes

#4: Validation – Example

Goal: Ensure documented requirements meet stakeholder needs

Example activities:

- **Stakeholder reviews of the specifications** (“Present SRS details to users, product owners, etc.”)
- **Prototype feedback** (basic map & search, e.g., “Try finding a coffee shop.")
- **Early test case ideas** (“How do we test if the map loads in under 2 seconds?", "What searches will verify relevant results?")

#5: Management

- Management in requirements engineering is concerned with managing the requirements throughout the software development lifecycle
  - **Prioritizing and organizing requirements:** Determining which requirements are most important or critical, and ensuring that they are given appropriate attention/resources
  - **Tracking changes to the requirements:** Maintaining a record of any changes to the requirements, as it is necessary to ensure that all stakeholders are aware of any changes to the requirements, and to prevent misunderstandings/miscommunications
  - **Ensuring that the requirements remain valid and up-to-date:** Periodic reviews of the requirements to ensure that they are still relevant and accurate, and updating them as necessary to reflect any changes in stakeholder needs or project goals

#5: Management – Example

Goal: Manage changes throughout development

Example activities:

- **Change request process** (“Users want to filter restaurants by their average star rating”)
- **Tracking links between requirements & development work** (“Linking the "Search" requirement to its design, code, and tests ensures everything aligns”)
- **Version control of requirements** (“Tracking different SRS versions helps manage the addition”)
- **Communicating changes to the team**

#6: Maintenance

- **Maintenance in requirements engineering** is concerned with managing the requirements of a software system after it has been deployed and is in use
  - During this stage, the requirements are reviewed, updated, and modified as necessary to reflect changes in the system's environment or the needs of its users
- **Some reasons for maintenance**
  - Systems are **operational over long periods of time**, and over that time, requirements change
  - Ensuring that the system continues to **function properly and efficiently**, e.g., bug fixing, hardware updates, performance updates
  - System remains **secure and compliant** with any relevant regulations or standards

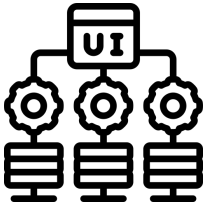
Outline

- **Part I: Requirements engineering**
  - Requirement types (functional/non-functional)
  - Stages in requirements engineering
  - **Non-functional requirements in the cloud**

- **Part II:** Software architectures in the cloud

Non-functional requirements

#1: Performance



#2: Scalability

#3: Reliability

#4: Availability

#5: Security

#6: Maintainability

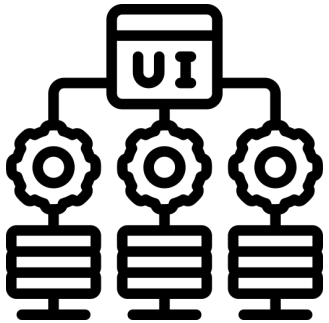
#7: Deployability

#1: Performance

- **The need for performance**
  - The specification of a computer system typically includes explicit (or implicit) performance goals
- **Performance metrics:**
  - **Latency:** The time interval between a user's request and the system response
  - **Throughput:** Number of work units done (or requests served) per time unit
  - **Utilization:** The percentage of capacity used to serve a given workload of requests
- **Service level agreements (SLAs):**
  - An SLA is an agreement between provider (cloud software) and client (or users) about measurable metrics, e.g., performance metrics

#2: Scalability

- **Scalability:**
  - Measures the ability of software systems to adapt to the increasing workloads, e.g., serving millions of active users!
- **Scalability via concurrency:**
  - **Vertical scaling (scale-up!):** Exploiting parallelism in multicores OR adding more resources on a single machine
  - **Horizontal scaling (scale-out!):** Exploiting distributed architectures by adding more machines in the system
- **Elasticity:**
  - The ability of software systems to expand/shrink the usage of computing resource with increasing/decreasing workload



#3: Reliability

- **Reliability:** Applications are prone to hardware and software failures (bugs) in the cloud
- **Fault-tolerance** is the property that enables a system to continue operating properly in the event of the failure
- **Reliability metrics:** common failure metrics that get measured and tracked for any system
  - **Mean time between failures (MTBF):** The average operational time between one device failure or system breakdown and the next
  - **Mean time to failure (MTTF):** The average time a device or system is expected to function before it fails (usually for not repairable devices)
  - **Mean time to repair (MTTR):** The average time to repair and restore a failed system

between each device

for a specific device

#4: Availability

- **High availability** specifies a design that aims to minimize the downtime of a system or service
  - The main objective of high availability is to keep these systems and services continuously available
- **Availability metrics:**
  - We measure high availability through the percentage of time that a service is guaranteed to be online and available for use in a year
  - For e.g., usually they are referred to as "9s"
    - **99.99% (four nines):** the four nines class accepts a maximum downtime of 52.6 minutes (less than an hour) per year
    - **99.999% (five nines):** the five nines class tolerates a maximum downtime of 5.26 minutes (few minutes) per year

#5: Security

- **Security:** Software deployed in the cloud is vulnerable to security vulnerabilities as the underlying computing infrastructure is untrusted (or shared by multiple tenants)
  - **Secure systems** deal with securing **computing**, **network** and **storage**

- **Security properties (CIA properties):**
  - **Confidentiality** refers to protecting information from unauthorized access
  - **Integrity** means data are trustworthy, complete and have not been accidentally altered or modified by an unauthorized user
  - **Availability** means data are accessible when you need them

#6: Maintainability

- **Maintainable software** allows us to quickly and easily:
  - Fix a bug
  - Add new features
  - Improve usability
  - Increase performance
  - etc.
- **Design tips** for maintainable software
  - **Modular design** to easily extend/modify system components
  - **Version control** for proper software/code management
  - **Quality management:** Refactoring, comments, code reviews, etc.
  - **Continuous integration testing/deployment** for ensuring the extensions are stable/bug free

#7: Deployability

- **Deployability** of a software system is the ease with which it can be taken from development to production
  - Incorporating (hardware and software) dependencies
  - Software updates (patches)

- **Design tips** for deployability
  - **Build and release management**
  - **Packaging dependencies** for deployment, e.g., containers
  - **DevOps pipeline:** continuous integration and deployment

Outline

- ~~**Part I:** Requirements engineering~~

- **Part II: Software architectures in the cloud**
  - Overview
  - Client-server architecture
    - Communication layers (REST and gRPC)
    - Serialization and deserialization of structured data using Protbuf
  - Monolithic architecture
  - Microservice architecture
    - Strangler pattern: From monoliths to microservices

Software architectures

The ***software architecture*** defines:

- The **fundamental structure** of a software system
- Defines the **system's components**, their **relationships**, and how they interact
- Provides **a framework for development, deployment, and maintenance**

This will impact:

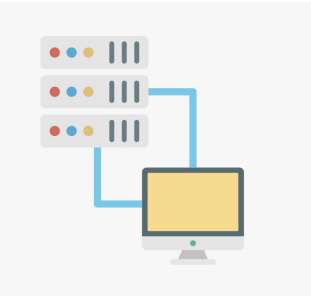
- How you can ***meet your requirements***
- How you ***structure your development***
- How you ***deploy your software***

Software architectures

Key Concepts:

- **Components:** Building blocks of the system
  - E.g., modules, services
- **Connectors:** Define how components interact
  - E.g., APIs, messaging
- **Styles (high-level):** A set of architectural constraints
  - E.g., client-server, layered, monolithic, microservices
- **Patterns (specific low-level):** Reusable solutions to common architectural problems
  - E.g., Model-View-Controller (MVC), Facade, Strangler, etc.

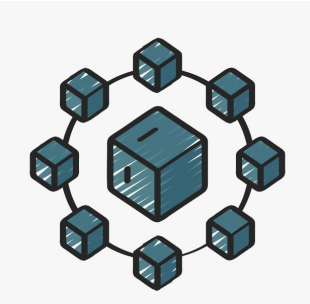
Software architectures



Client-server

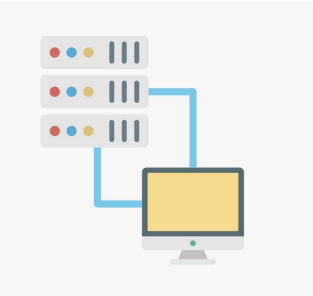
Three-tier

Monolithic



Microservices

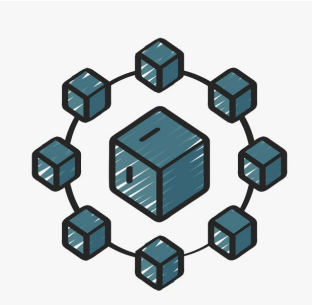
Software architectures



Client-server

Three-tier

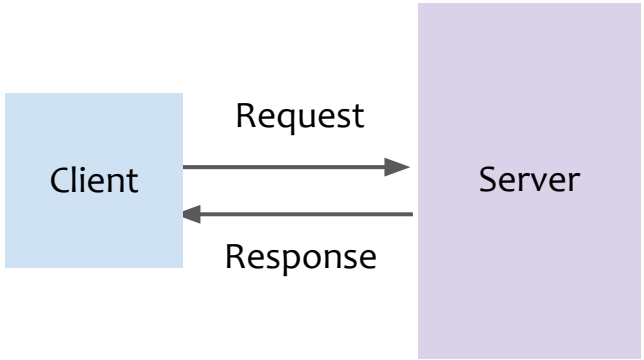
Monolithic



Microservices

Client-server architecture

- A **distributed system** model
- Divides tasks between two main components:
  - **Clients:** Request services
  - **Servers:** Provide services
- Clients and servers **communicate over a network**
- Based on the **request-response** protocol



der Server (no change)
der Kunde, -n • customer

der Klient, -en
die Klientin, -nen
} client

Key components

- **Client:** ***Initiates requests*** for services or resources
  - Responsible for user interface and interaction
  - Examples: Web browsers, mobile apps, desktop applications
- **Server:** ***Listens, processes, and provides services*** for client requests
  - Manages resources, data, and security
  - Examples: Web servers, database servers, file servers
- **Network:** The communication channel between clients and servers
  - Can be a local network (LAN) or a wide area network (WAN) like the Internet
- **Communication protocols:** Define the rules and formats for communication
  - Examples: REST, gRPC

How it works?

- Client ***sends a request*** to the server
- The server ***receives and interprets*** the request
- The server ***processes the request***, which may involve:
  - Retrieving data from a database or accessing a file
  - Performing a computation
- The server ***sends a response*** to the client
- The client ***receives the response and displays*** the results to the user

Example: Credit card transaction payment

Problem statement: A customer uses ***a client application*** to initiate a credit card payment. The application securely ***sends transaction details*** to a payment server, which ***validates the card, authorizes the transaction***, ***records it***, and ***sends a confirmation to the client*** for display to the customer.

Client-server communication: Two approaches

- **REST (Representational State Transfer)**
  - **HTTP-based,** flexible and widely adopted
  - Uses **HTTP requests/responses** for communication
  - Data is commonly formatted **as JSON**
  - **Stateless communication** where each request is independent

- **gRPC (gRPC Remote Procedure Call)**
  - **RPC-based,** uses HTTP/2 as its transport protocol
  - Efficient and structured framework for **high-performance communication**
  - Data is serialized using **Protocol Buffers**
  - **Stateless communication** where each request is independent

gRPC

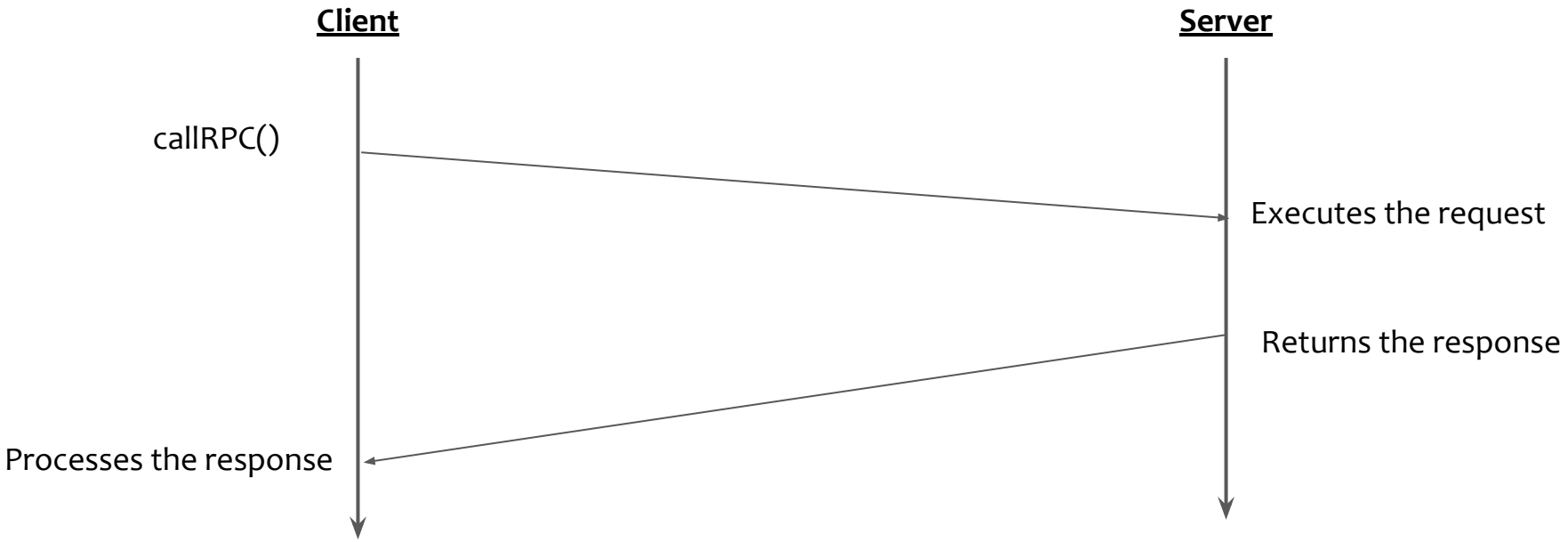
- A high-performance and widely-deployed RPC framework
- Platform independent
- Applicable in almost all distributed computing use-cases:
  - Devices, mobile applications and browsers to backend cloud/data-center services.

Remote procedure calls (RPCs)

- Extends the notion of **local procedure calls**
  - Allow two process to communicate (local or remote via network)
  - Residing in different address spaces
  - Presumes the existence of a low-level transport protocol (TCP/IP or UDP)

- How does a **RPC work**?
  - A client invokes an RPC, similar to a function call
  - Arguments are passed to the remote procedure
  - The caller waits for a response to be returned

RPC in action

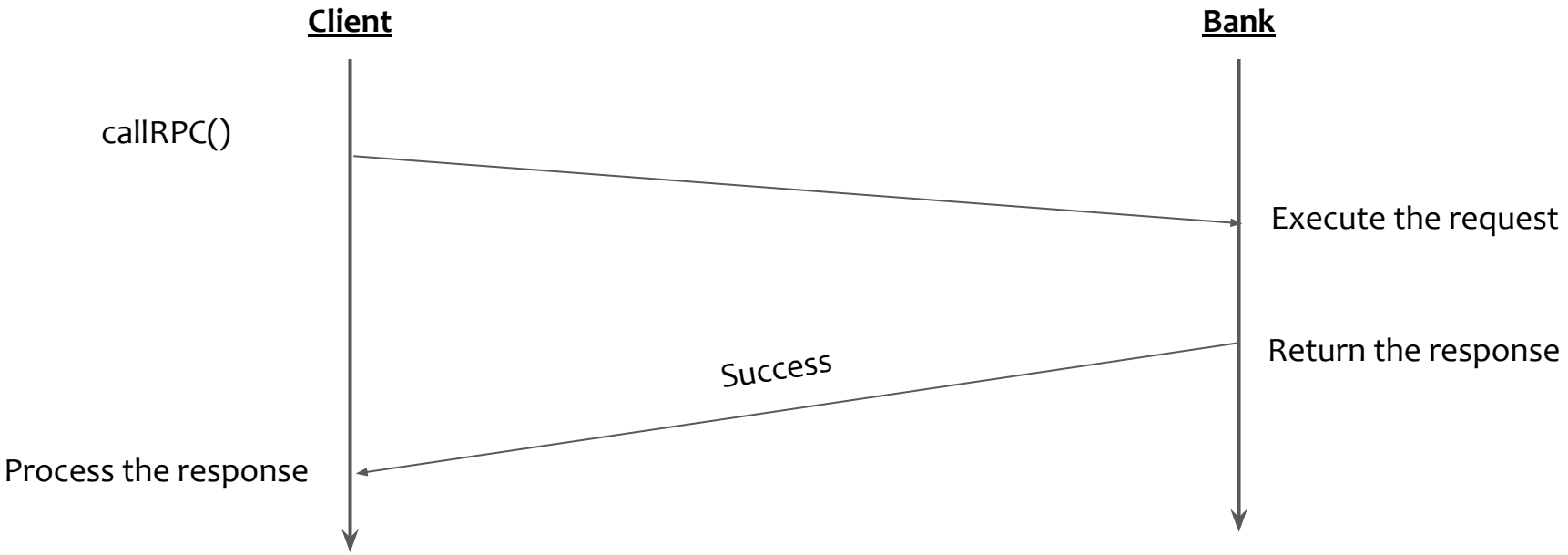


An application programmer:

1. Develops the client program
2. Develops the server program
3. Specifies the protocol for client-server communication

charge 3.99 GBP to credit card “123..”

How gRPC works?



A shopping cart example

// Online shop handling customer's card details

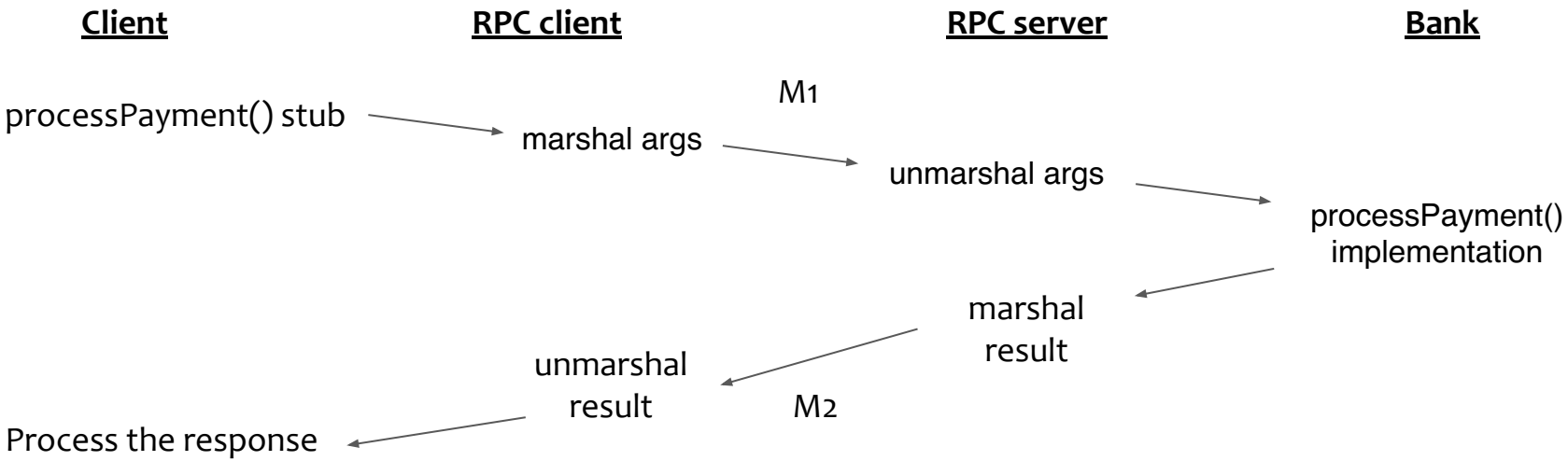
```
Card card = new Card();
card.setCardNumber("123…");
card.setExpiryDate("10/2024");
card.setCVC("123");

Result result = paymentsService.processPayment(card, 3.99, Currency.GBP);


if (result.isSuccess()) {
        fulfilOrder();
}
```

gRPC in action

marshaling is the process of transforming the memory representation of an object into a data format suitable for storage or transmission, especially between different runtimes.



$m_1 =$

```
{
  "request": "processPayment",
  "card": {
    "number": "1234567887654321",
    "expiryDate": "10/2024",
    "CVC": "123"
  },
  "amount": 3.99,
  "currency": "GBP"
}
```

$m_2 =$

```
{
  "result": "success",
  "id": "XP61hHw2Rvo"
}
```

Data communication format

- The RPC framework needs to convert datatypes such that the caller's arguments are understood by the code being called, and vica-versa for the return values

- *Interface Definition Language (IDL)* provides language-independent type signatures of the functions that are being made available over RPC

- From the IDL, software developers can then automatically generate marshalling/unmarshalling code and RPC stubs for the respective programming languages of each service and its clients

Google's protocol buffers

- Protocol buffers provide a serialization format for packets of typed, structured data

- Specify the message format in a language-neutral, platform-neutral, extensible format for serialization/deserialization structured data (.proto files)

- The proto compiler is invoked on .proto files to generate code in various programming languages to manipulate the corresponding protocol buffer, i.e., to serialize and parse the whole structure to and from raw bytes

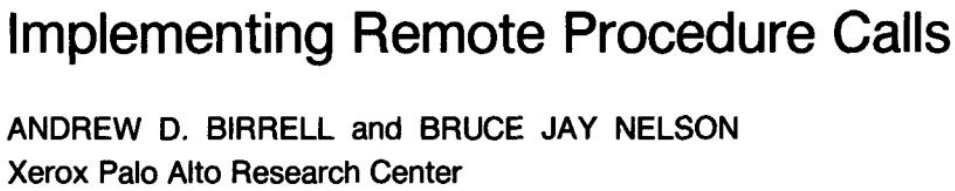
```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

Common error conditions

- Network data loss resulting in retransmission
  - Incorrect operations when the data is received multiple times
- Server process crashes during RPC operation
  - Before completing its task → Client retries the request
  - After completing its task and before responding → Recovery for a consistent state
- Client process crashes before receiving response
  - Client is restarted, server discards response data

References

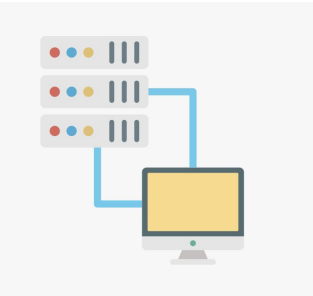
- gRPC: https://grpc.io/
- Protocol buffers: https://developers.google.com/protocol-buffers
- Implementing Remote Procedure Calls
  - https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf



Implementing Remote Procedure Calls

ANDREW D. BIRRELL and BRUCE JAY NELSON
Xerox Palo Alto Research Center

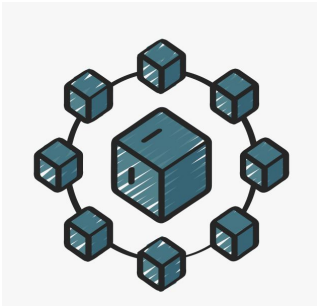
Software architectures



Client-server

Three-tier

Monolithic



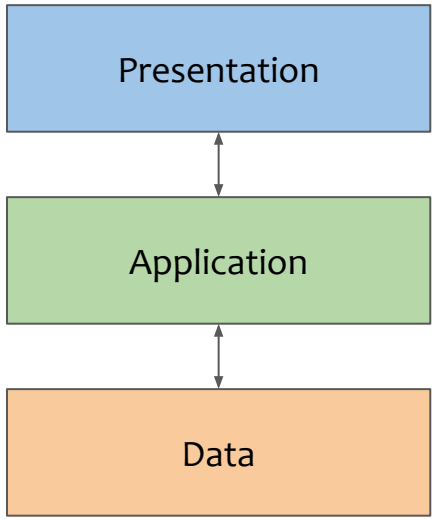
Microservices

Three-tier architecture

A classic architecture with 3 main components (*tiers*):

- Presentation
- Application
- Data

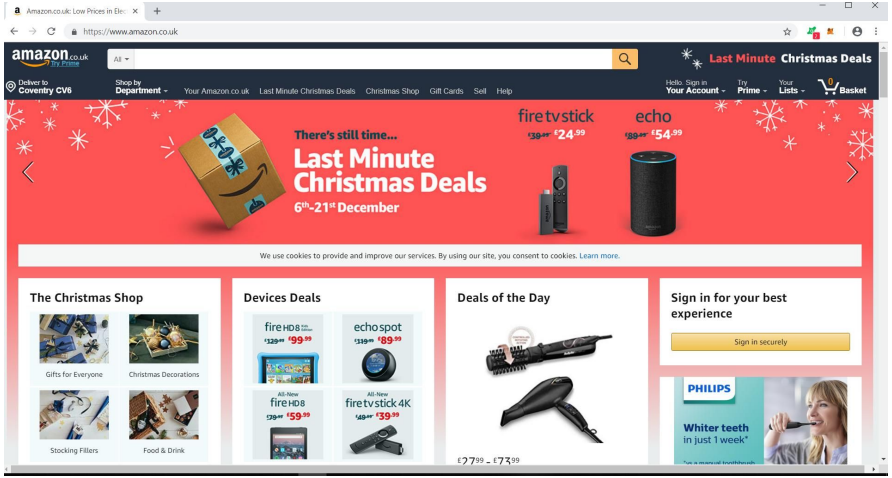
All interactions happen through the application tier

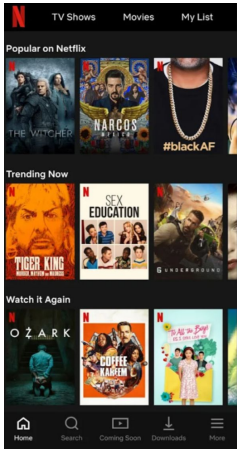


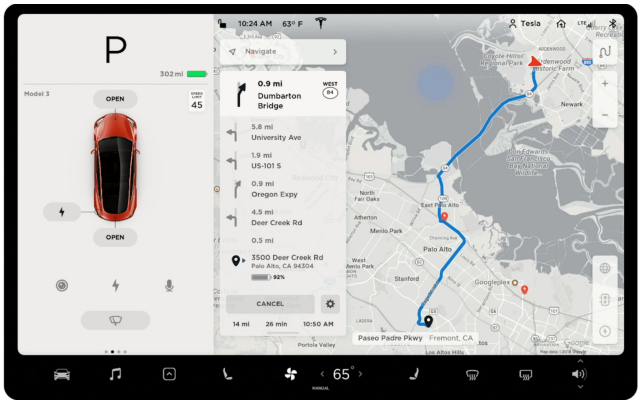
Three-tier architecture: Presentation

The presentation tier:

- Serves the UI to users
- Manages the interactions between the users and the software



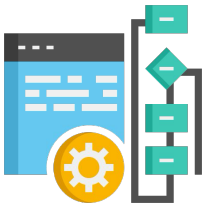




Three-tier architecture: Application

The application tier:

- Contains the logic of the application
- Processes the data from/to the user through the presentation tier



Recommendation algorithm

Video processing

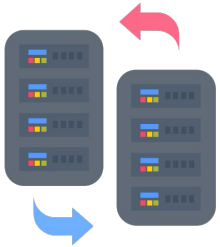
Route calculation

Three-tier architecture: Data

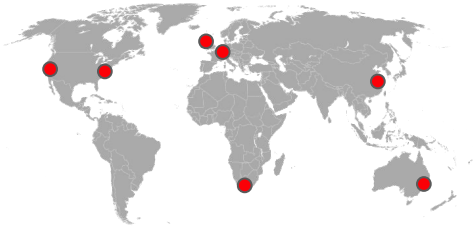
The data tier:

- Stores the information needed by the software
- Services the data to the application layer

Database



Data safety



Geo replication

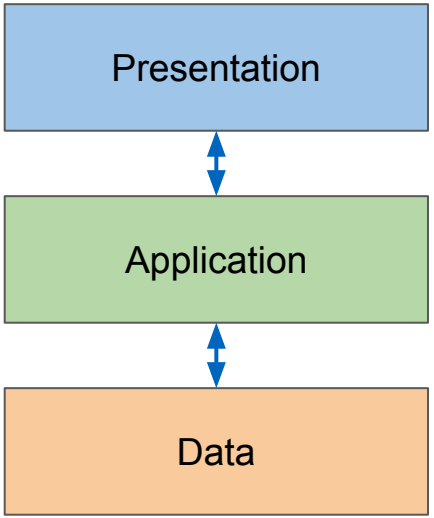
Three-tier architecture: Communication

Tiers need to communicate data to function properly

Presentation and *data* tiers can only communicate with the *application* tier

The most widely used communication technologies are:

- REST
- Remote Procedure Calls



REST

REpresentational **S**tate **T**ransfer is a set of constraints for communication:

- *Client-server:* client and server are independent
- *Stateless:* the server does not maintain a state between requests
- *Cacheable:* data can be cached anywhere (client, server, on the network)
- *Uniform interface:* the server interface should be usable by an arbitrary client, without knowledge of the internal server representation
- *Layered systems:* clients can transparently communicate with the server through other layers (proxy, load balancer)

REST methods

REST methods manipulate resources and are usually mapped to HTTP methods, usually formatted in HTML, XML or JSON:

- **GET:** retrieve a resource, *e.g., query the list of branches of a GitHub repository:*

```
curl -X GET https://api.github.com/repos/OWNER/REPO/branches
```

- **POST:** create a resource, *e.g., rename a branch in a GitHub repository:*

```
curl -X POST \
  https://api.github.com/repos/OWNER/REPO/branches/BRANCH/rename \
  -d '{"new_name":"my_renamed_branch"}'
```

- **PUT:** update a resource, *e.g., merge a pull request in a GitHub repository:*

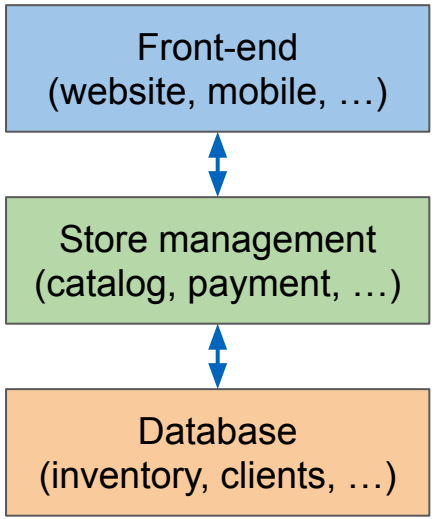
```
curl -X PUT \
  https://api.github.com/repos/OWNER/REPO/pulls/PULL_NUMBER/merge \
  -d '{"commit_title":"title","commit_message":"msg"}'
```

- **DELETE:** delete a resource, *e.g., delete a GitHub repository:*

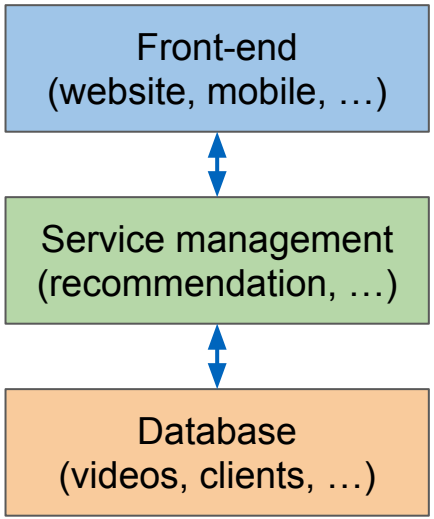
```
curl -X DELETE https://api.github.com/repos/OWNER/REPO
```

Three-tier architecture: Examples

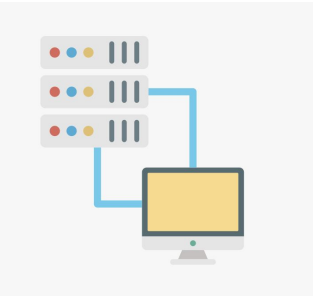
Online store



Streaming service



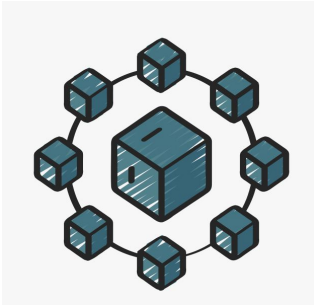
Software architectures



Client-server

Three-tier

Monolithic



Microservices

Quick warning

What follows might feel dense but no worries!

The goal is to give you an overview of

- Software architectures in the cloud
- Their benefits and challenges
- Some widely used techniques

You don’t need to memorize everything for the exercises/exams!

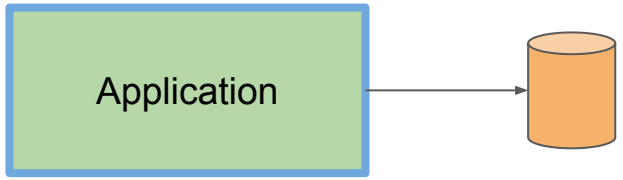
Software architectures in the cloud

Choice of architecture depends on:

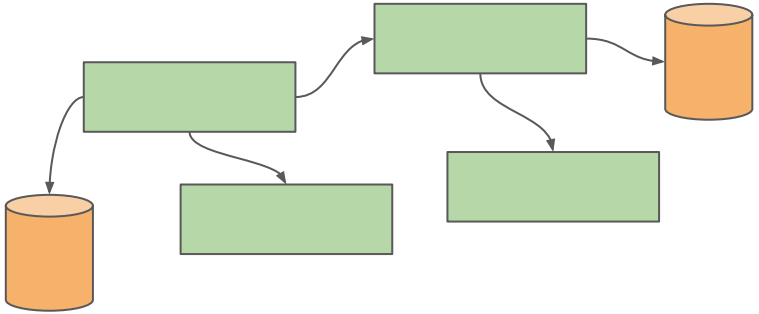
- Requirements
  - Functional
  - Non-functional
- Development process
  - Size of the software
  - Number of developers

Main architectures in the cloud:

- Monolithic



- Microservice-based

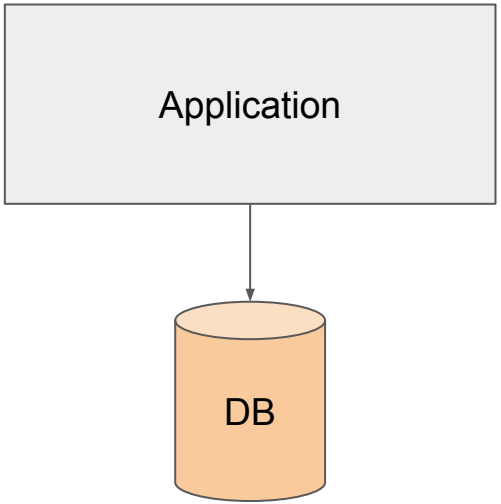


Monolithic architectures

All functionalities of the software are packaged and deployed together

- All tiers are tightly coupled into a single program
- Usually deployed as a packaged artifact

  e.g., JAR files for JAVA applications

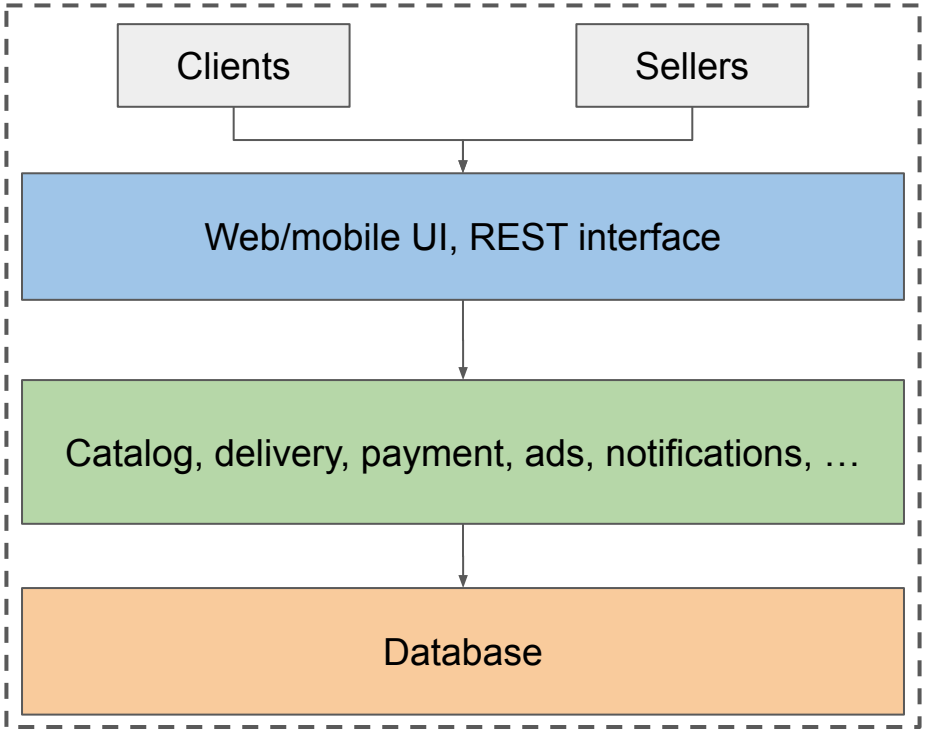


Two main types of monoliths:

- Single process
- Modular

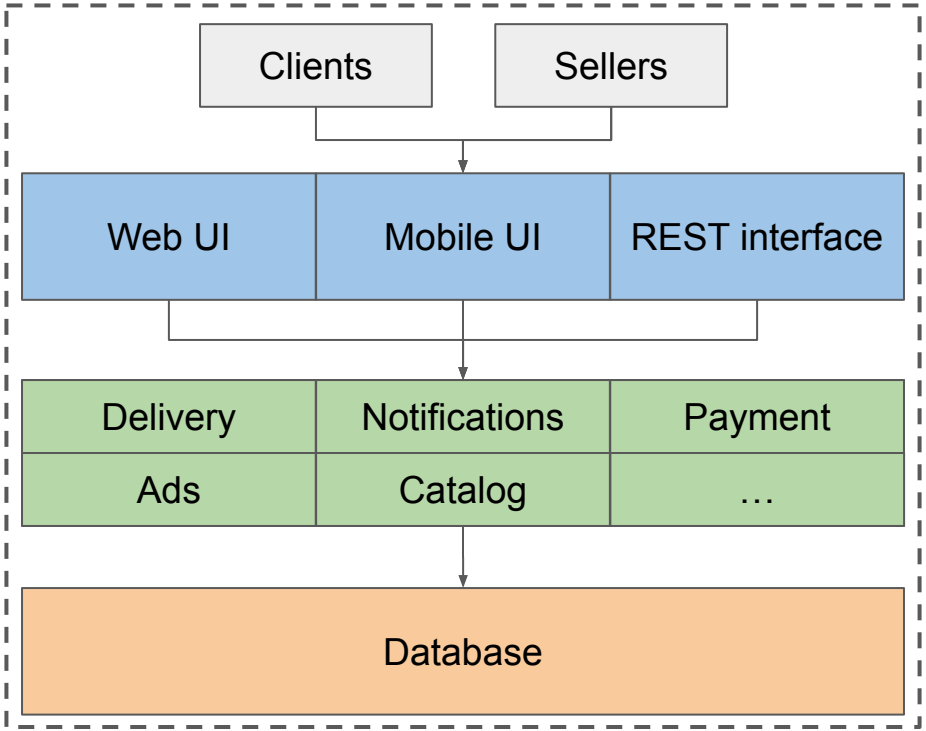
Single-process monolith

Everything is packaged as a single process, all the code is tightly coupled

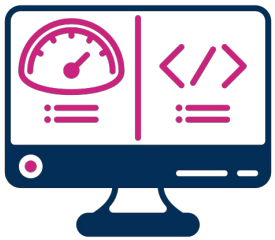


Modular monolith

Still a single package, but with independent modules



Benefits of monoliths



Development

Deployment

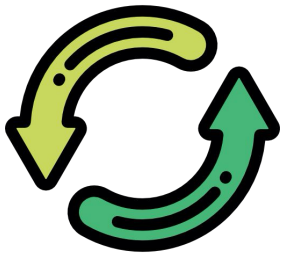
Performance

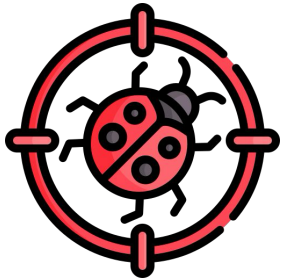
Developing monoliths

All components of the monolith share memory, code, libraries, runtime, …

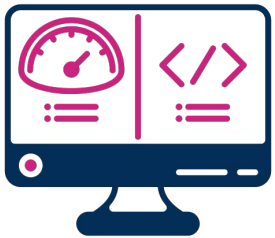
This simplifies

- Code reuse: code/libraries can be used across components
- Testing: end-to-end testing is possible in a straightforward way
- Debugging: introspection is easy (shared address space, code and runtime)
- Monitoring: unified logging system






Benefits of monoliths



Development

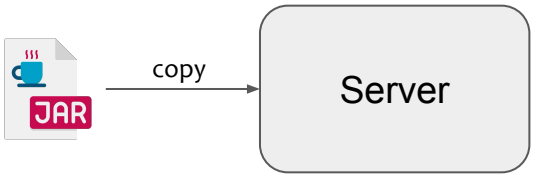
Deployment

Performance

Deploying monoliths

Simple deployment with a single package to deploy

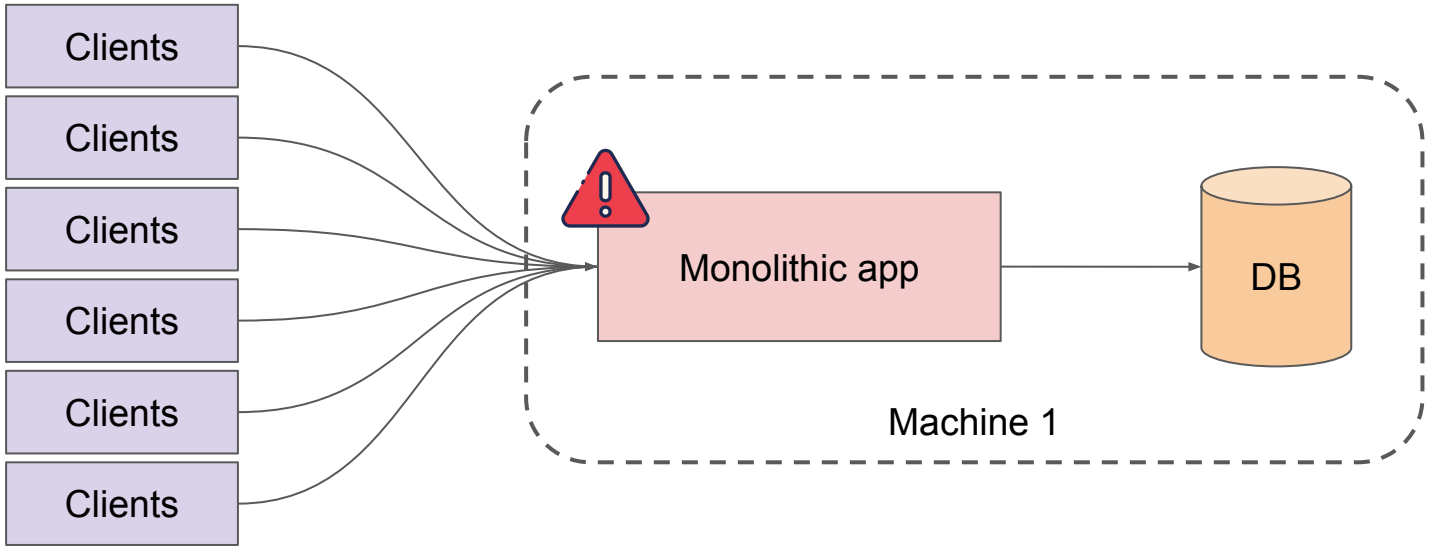
- Just copy the application package to a machine and run it
- No complicated network configuration
- Large choice of deployment models (baremetal, virtual machines, containers)



Handling contention

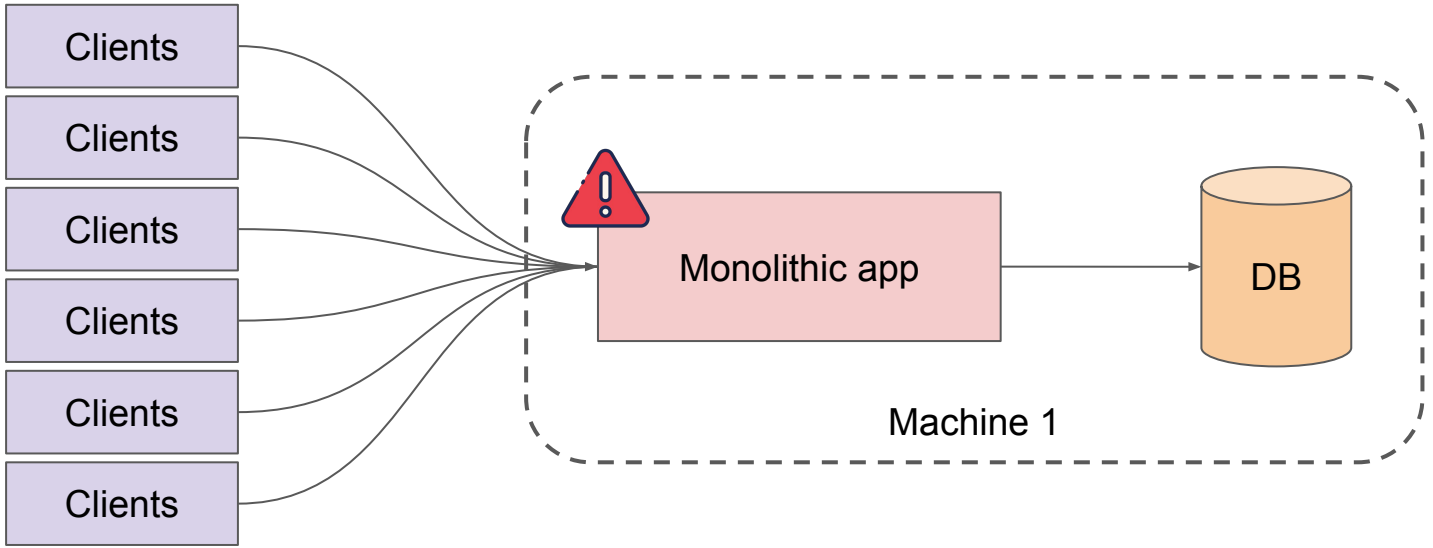
If you have too many clients, one machine might not be enough

- Performance deterioration
- More faults
- SLA violations



Scaling monoliths

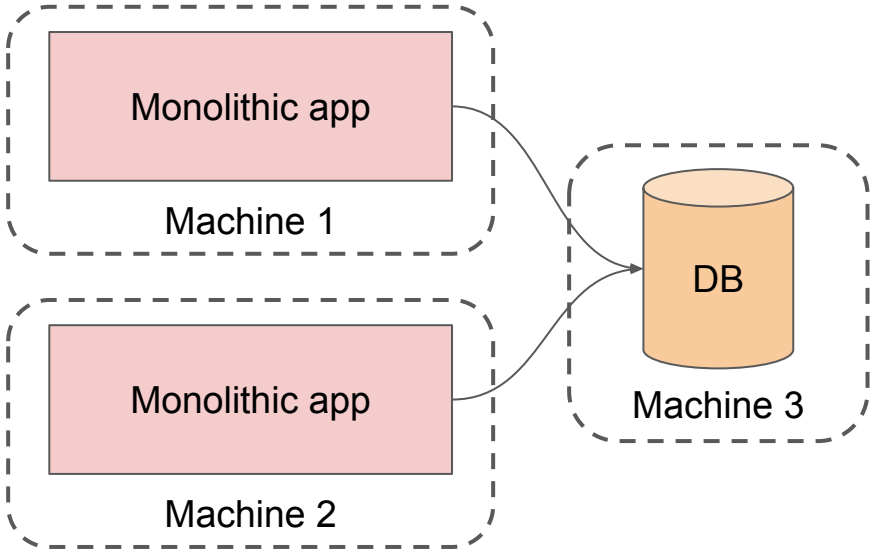
You can reduce contention by scaling out your application



Scaling monoliths

You can reduce contention by scaling out your application

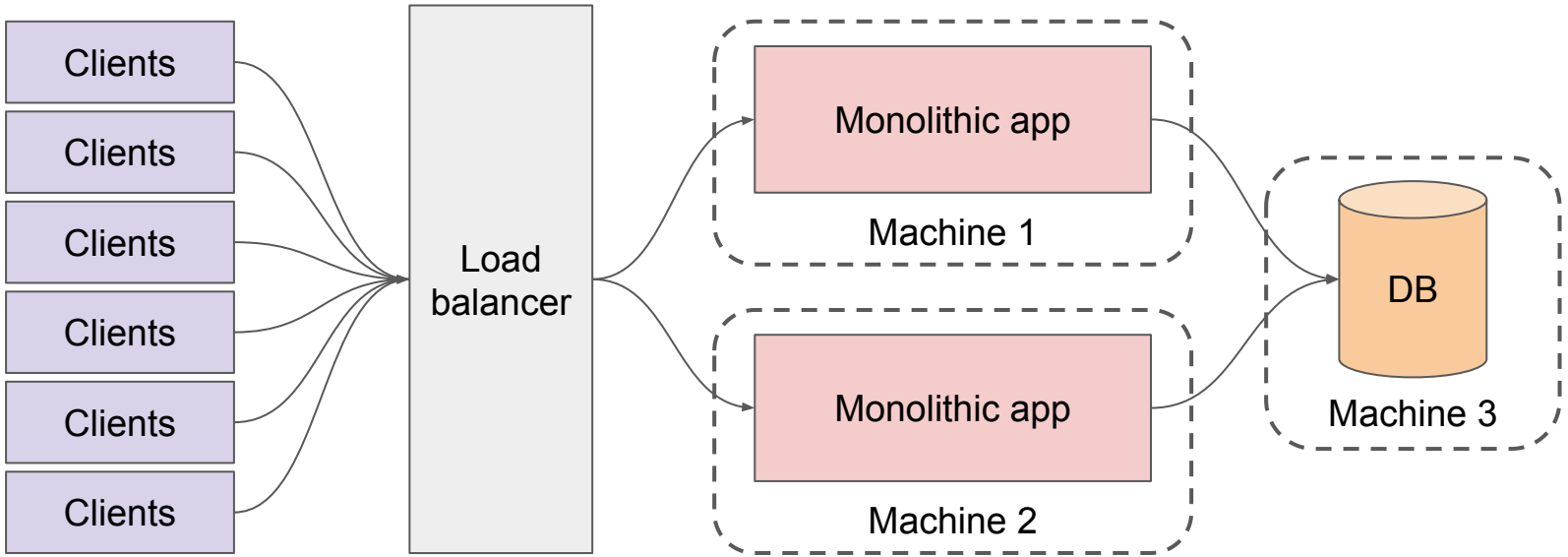
- Deploy multiple instances of the monolith (**monolith ≠ single instance**)



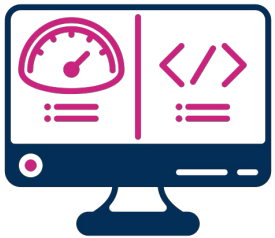
Scaling monoliths

You can reduce contention by scaling out your application

- Deploy multiple instances of the monolith (**monolith ≠ single instance**)
- Deploy a load balancer to route client requests



Benefits of monoliths



Development

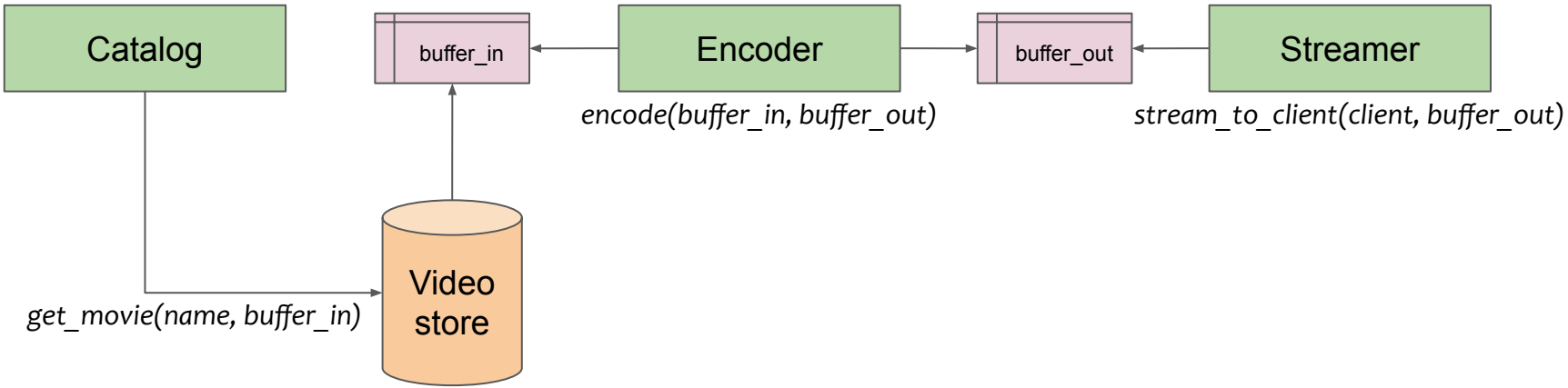
Deployment

Performance

Monolith performance

The components are *tightly coupled*, allowing performance gains through:

- Shared address space: using shared memory is easy
- Shared code: all interactions are cheap function calls

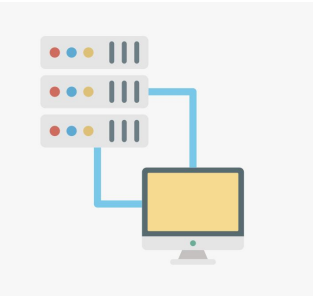


Drawbacks of monoliths

With application growth, the tight coupling of modules becomes an issue

- One **localized code change can impact** the whole application
  - Development team coordination becomes difficult
  - Testing is also impacted
- **Continuous integration and deployment** become heavier
  - The whole application must be redeployed every time
  - Extensive testing is required to check for regressions, even for small changes
- **Scaling is monolithic**
  - Everything is scaled in the same way, even if modules have different needs
  - Hardware choices/cost become difficult
- **Reliability**
  - A bug in one module will crash the whole application

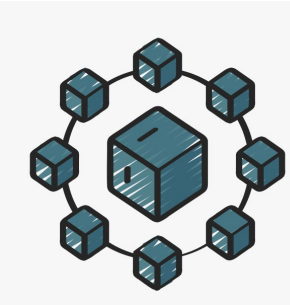
Software architectures



Client-server

Three-tier

Monolithic



Microservices

Microservice architecture

Monolithic architecture issues stem from the coupling of all components

Microservice architecture is a way of overcoming these issues by more strictly compartmentalising the components of the software

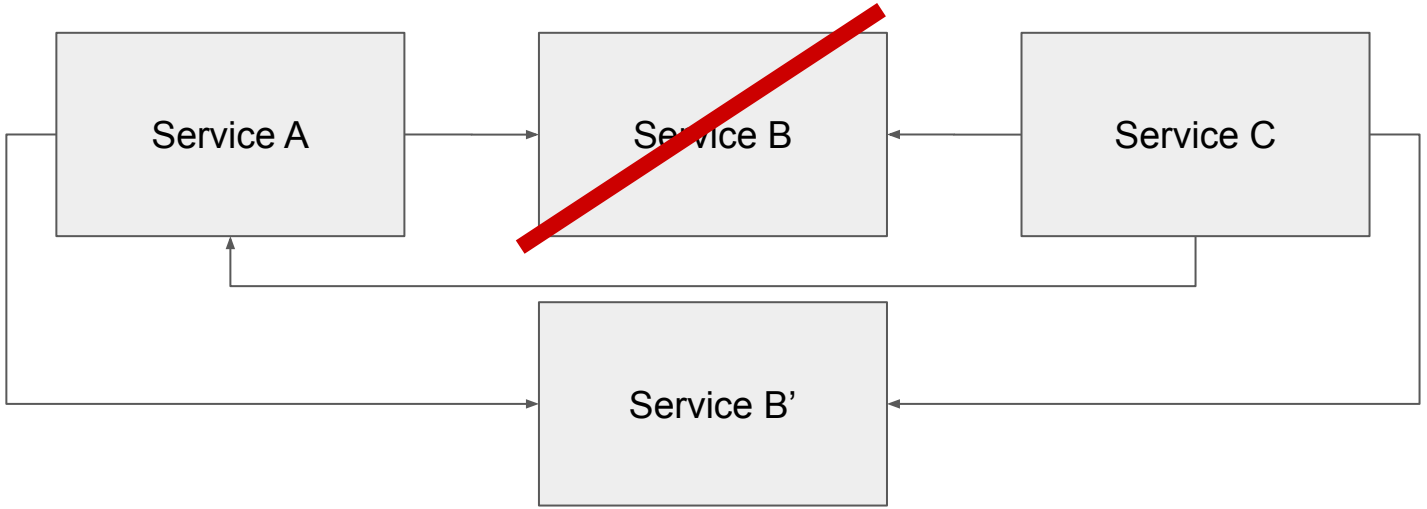
Key principles

- Loose coupling
- High functional cohesion

Loose coupling

Microservices are *loosely coupled*

Changing a microservice does not require to change anything else

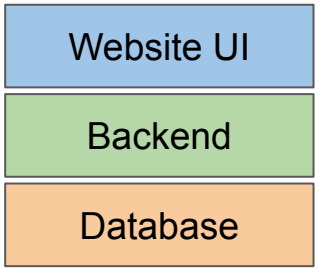


High functional cohesion

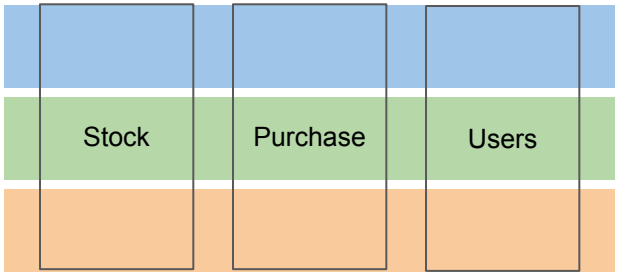
Microservices have *high function cohesion*

A microservice has a single, well-defined purpose

Example: An online store

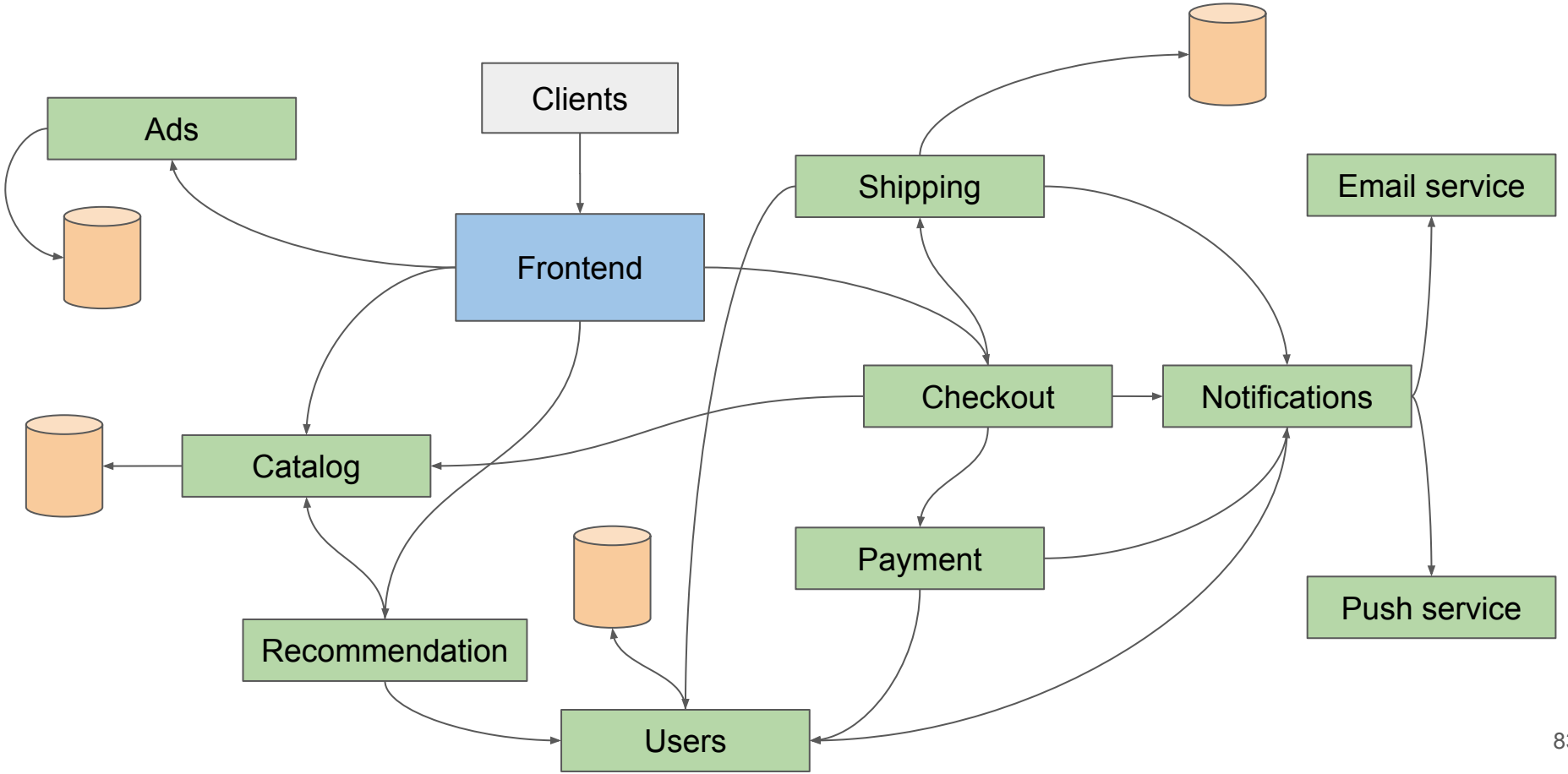


3-tier architecture



Microservice architecture

Example: An e-commerce application



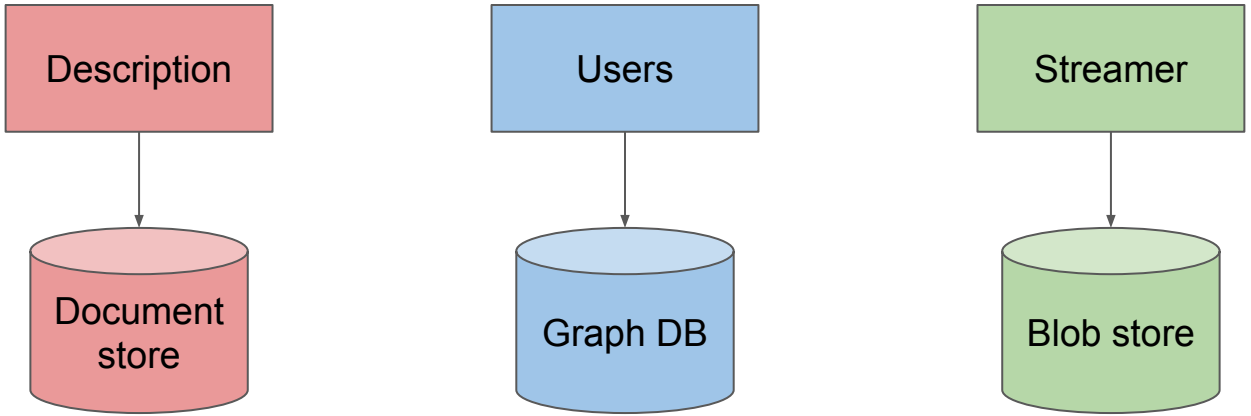
Advantages of microservices

- Technical heterogeneity
- Scalability
- Robustness
- Composability

Technological heterogeneity

Microservices allow using the best technology for each service, with little constraints

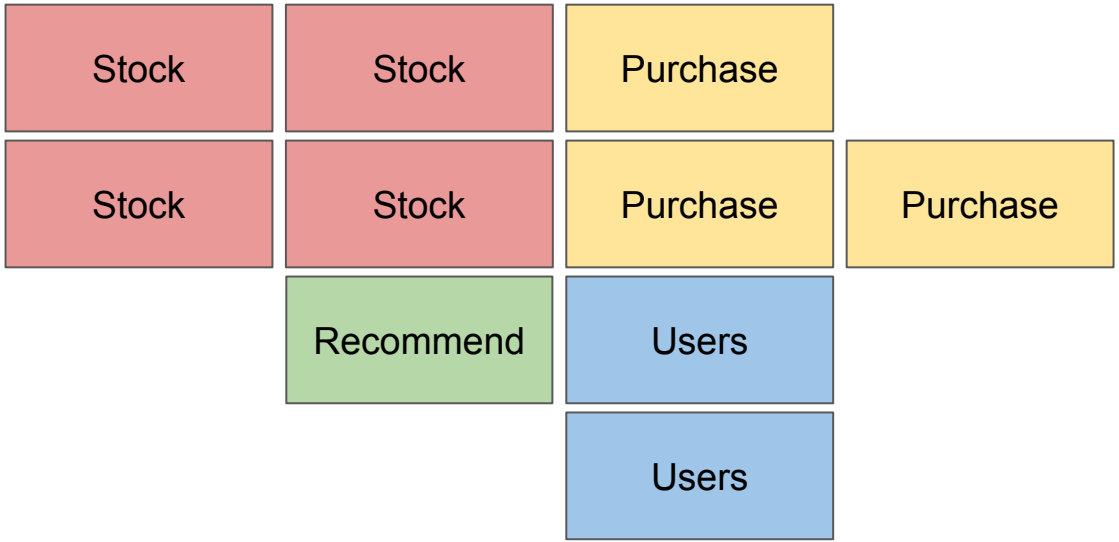
Example: use the best database type for each microservice



Scaling microservices

Scaling can be done at a finer grain than with monolithic architectures

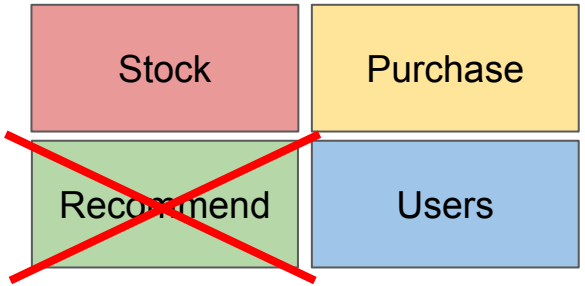
Example: scale each microservice depending on its specific requirements



Microservice robustness

Contained *failure points*, allows application to continue with partial functionality

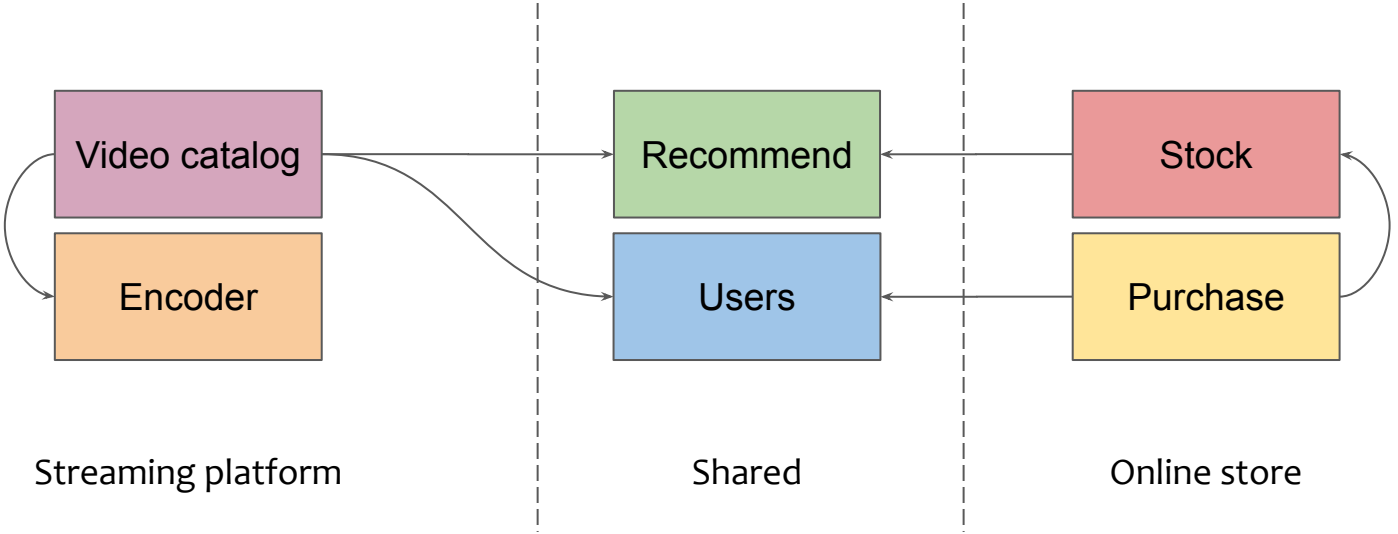
Example: if the recommendation service fails, an online store can still continue to work for purchases and previous order management



Microservice composability

Some microservices can be shared by multiple applications

Example: A company offering a streaming platform and an online store



Microservice pain points

- Development process
- Interservice communication
- Distributing data

Development process

The increased complexity of the software architecture introduces pain points for the teams developing the software

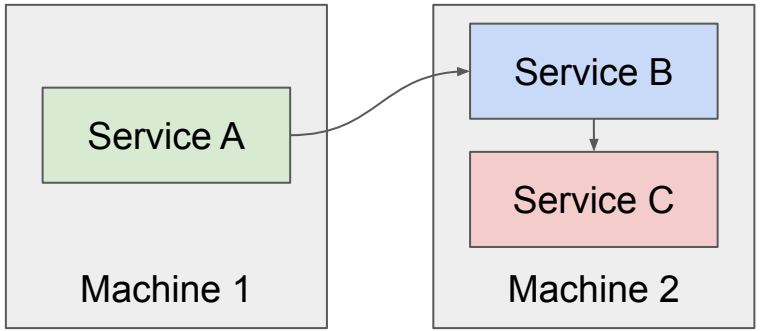
- Large number of services interact in unpredictable ways
- Need for a “real” deployment to actually test
- Monitoring and debugging become more distributed, with less deterministic behaviors, and more complex interactions
- Downside of technological heterogeneity: more varied expertise is required

Interservice communication

All communication can go through the network, creating two side effects:

Latency:

- Any API call can go to a different machine
- Network stack overhead
- Physical network overhead



Interservice communication

All communication can go through the network, creating two side effects:

Latency:

- Any API call can go to a different machine
- Network stack overhead
- Physical network overhead

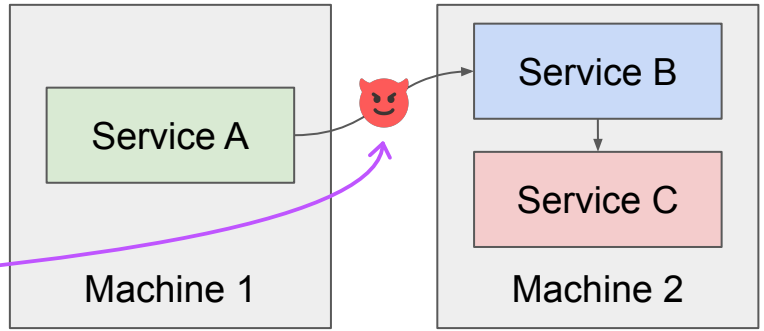
Security:

- Vulnerable to new vector of attacks

  e.g., man-in-the-middle

- Need secure communication protocols

  encryption, authentication, …

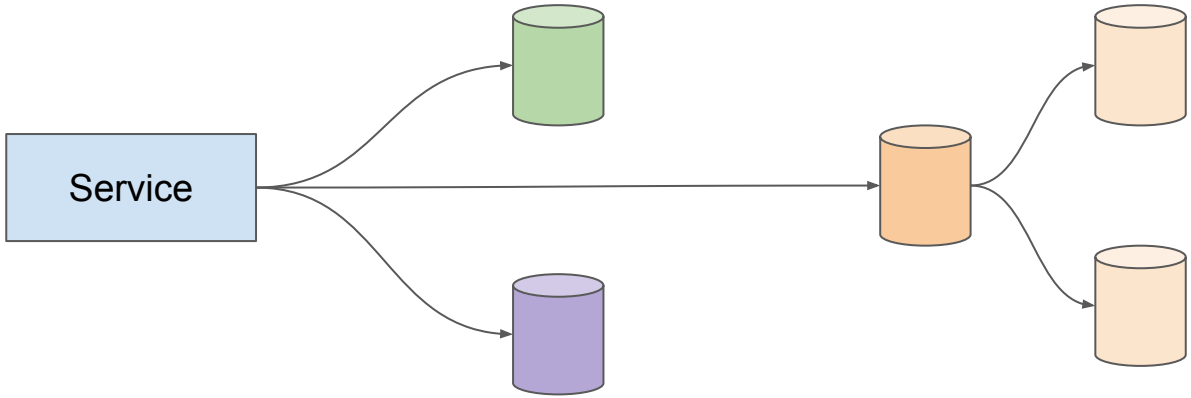


Distributing data

To enforce loose coupling, we split the databases so that each service owns their data

This can create a performance overhead:

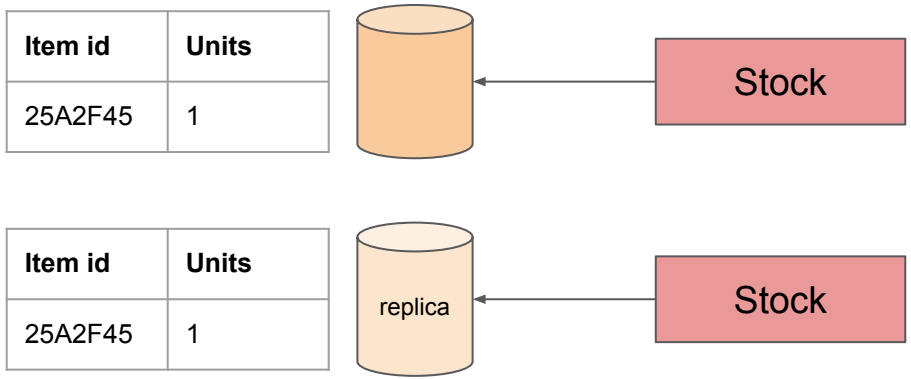
- A query can access multiple databases located on different machines
- Additional API calls needed to enforce consistency between databases



Data layer: Consistency

With a single database, multiple operations can be done atomically

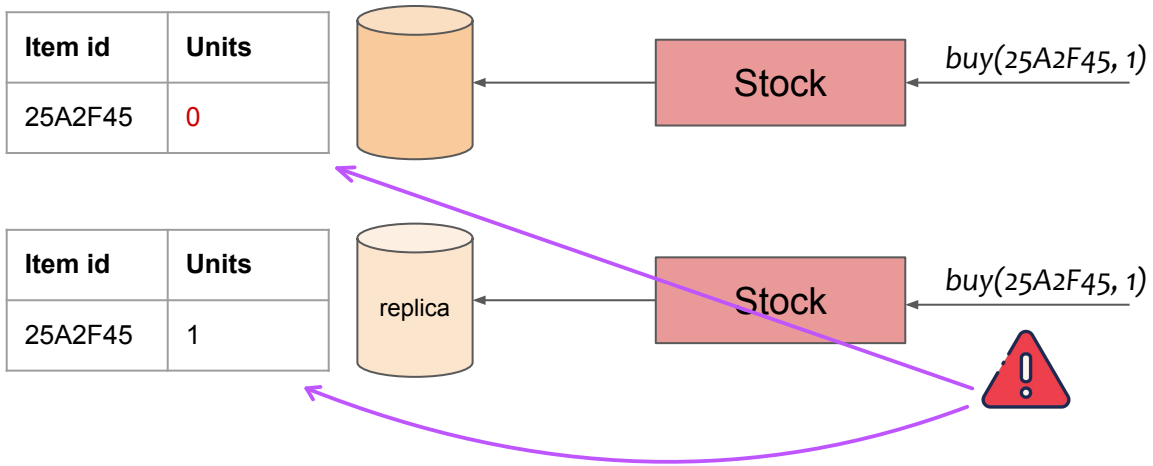
With distributed databases, this becomes difficult, and different versions of the database can coexist at the same time



Data layer: Consistency

With a single database, multiple operations can be done atomically

With distributed databases, this becomes difficult, and different versions of the database can coexist at the same time

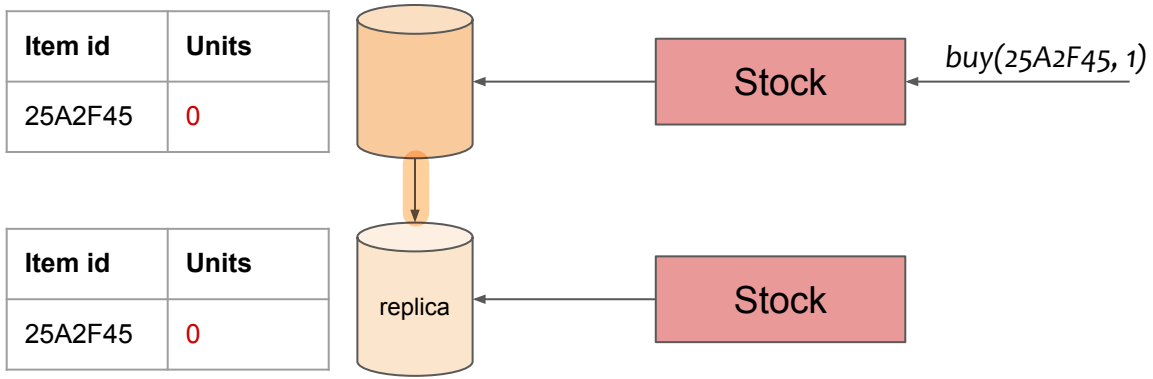


Data layer: Consistency

With a single database, multiple operations can be done atomically

With distributed databases, this becomes difficult, and different versions of the database can coexist at the same time

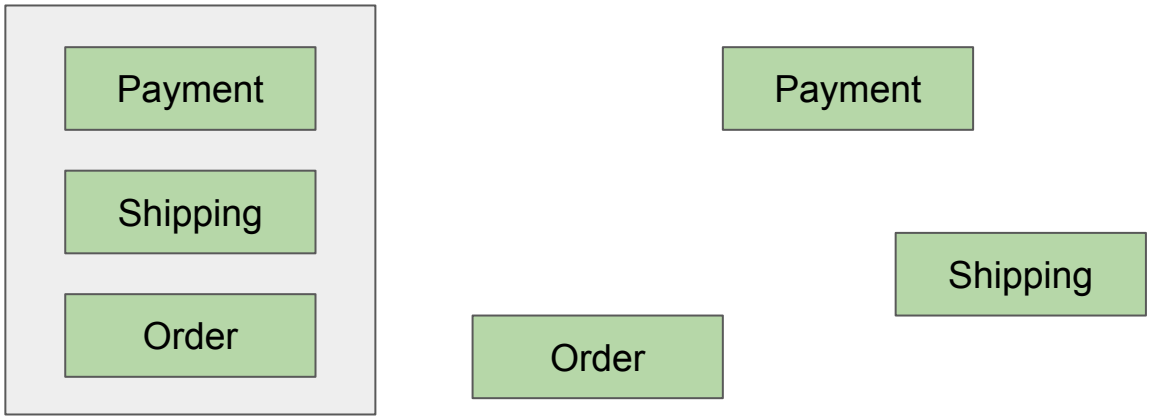
Consistency must be **enforced** or **tolerated** by the system



From a monolith to microservices

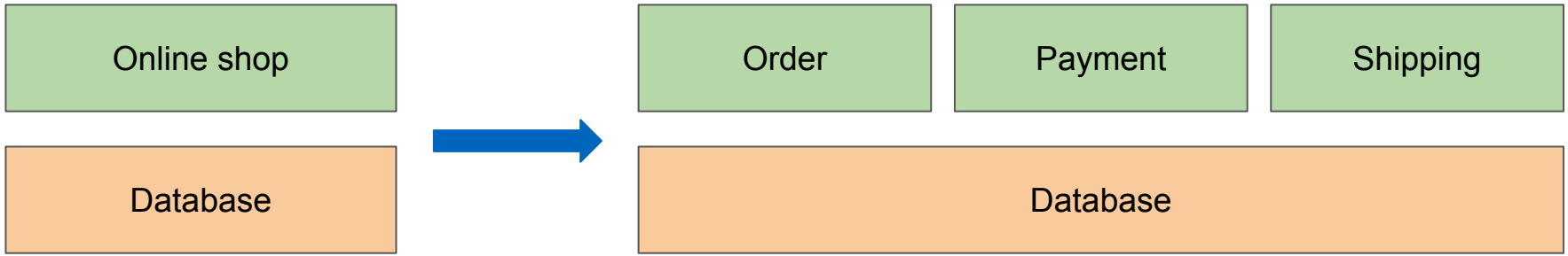
Refactoring a monolithic application into microservices can be tedious

An incremental refactor is recommended, using the **Strangler Pattern**



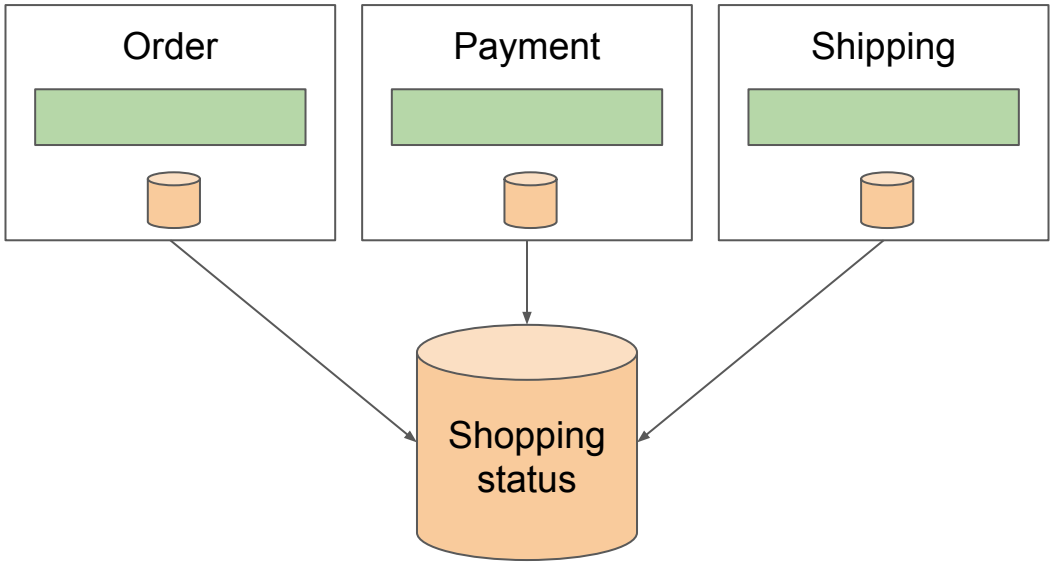
Achieving loose coupling

- Use Domain-Driven Design to isolate cohesive functions into microservices



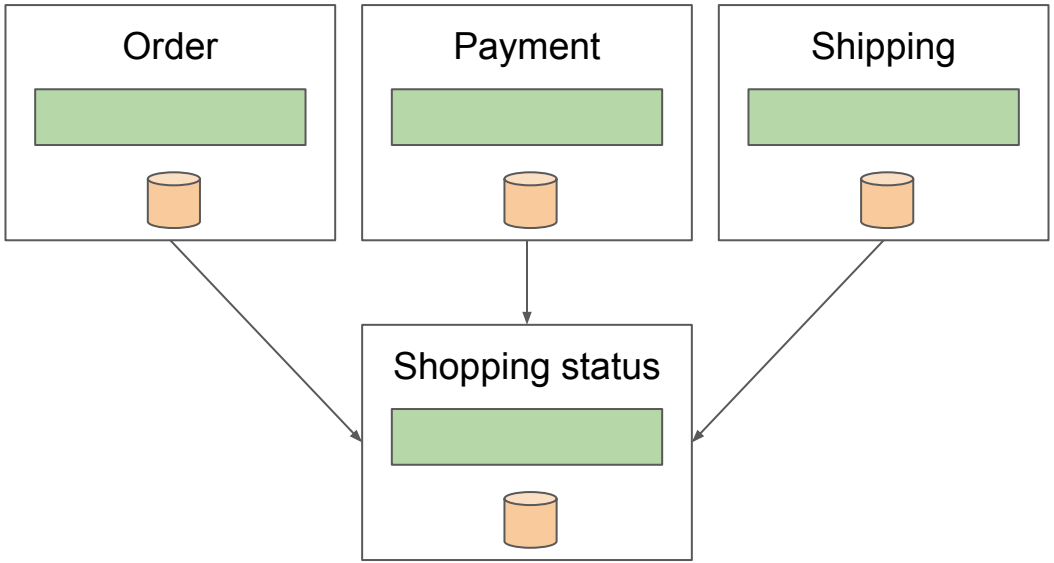
Achieving loose coupling

- Use Domain-Driven Design to isolate cohesive functions into microservices
- Split the databases so that each service owns its data



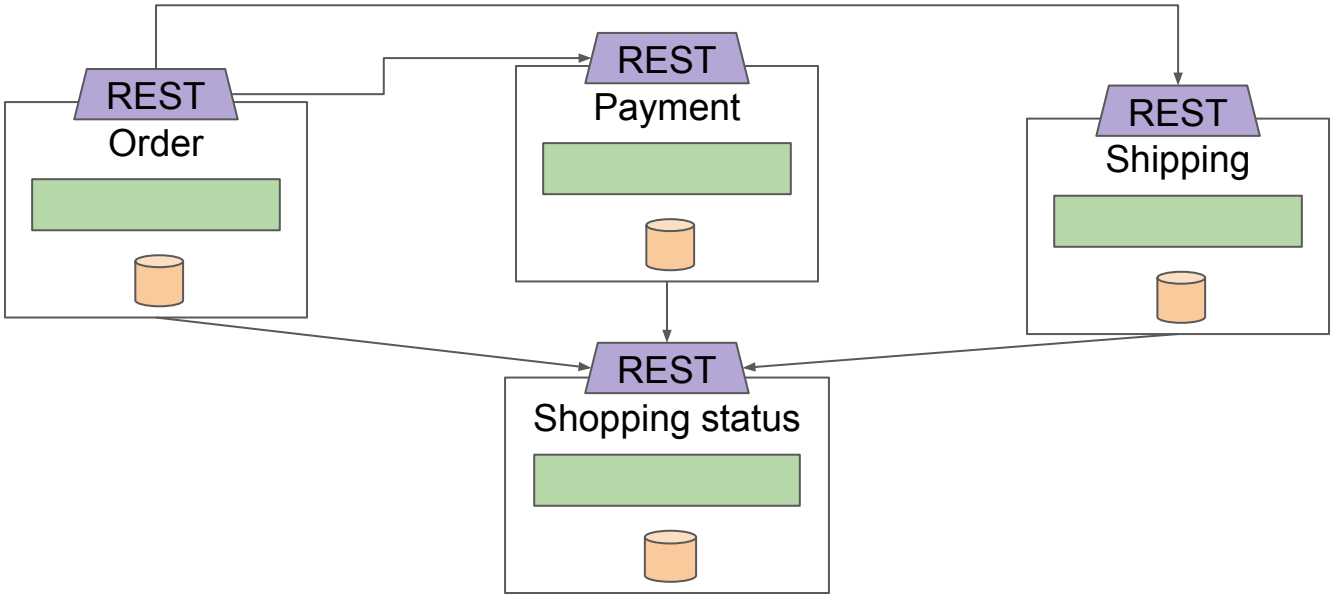
Achieving loose coupling

- Use Domain-Driven Design to isolate cohesive functions into microservices
- Split the databases so that each service owns its data
- Extract shared data as its own service



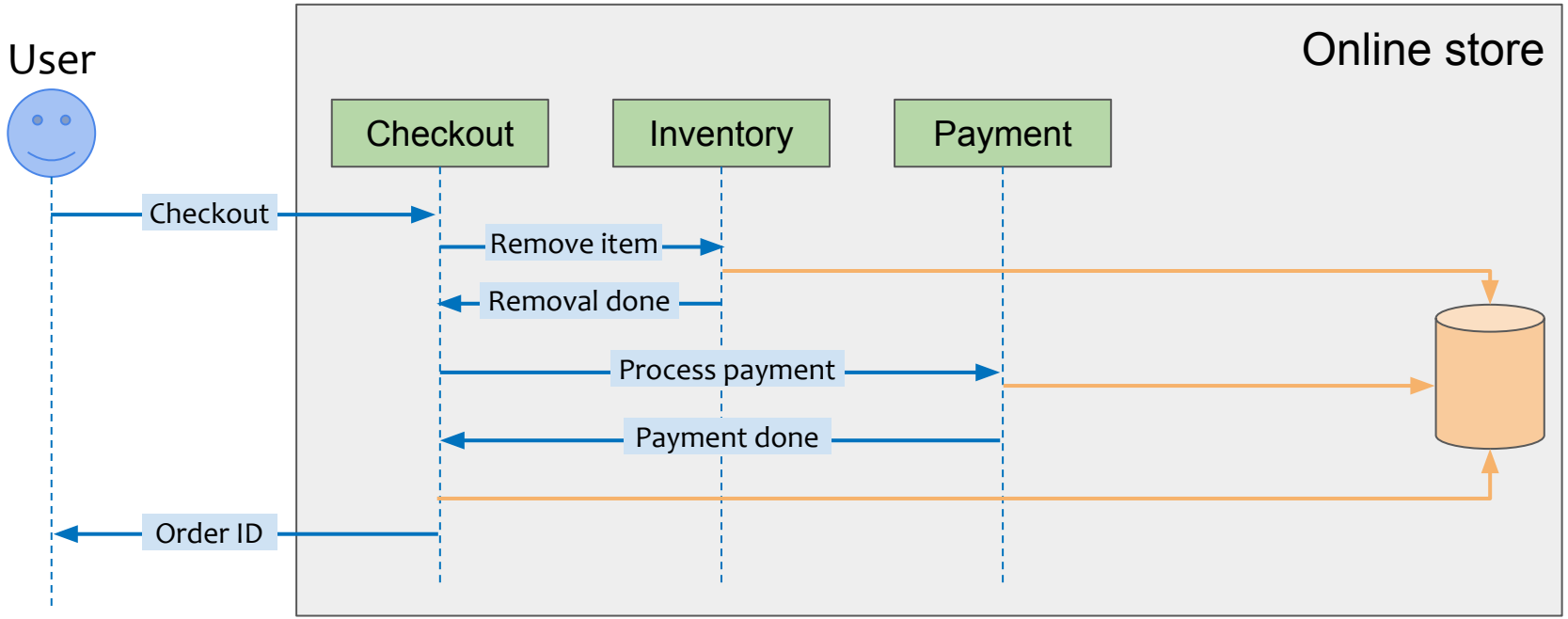
Achieving loose coupling

- Use Domain-Driven Design to isolate cohesive functions into microservices
- Split the databases so that each service owns its data
- Extract shared data as its own service
- Provide an API to the “outside world”, making each service a black box



Refactoring communication

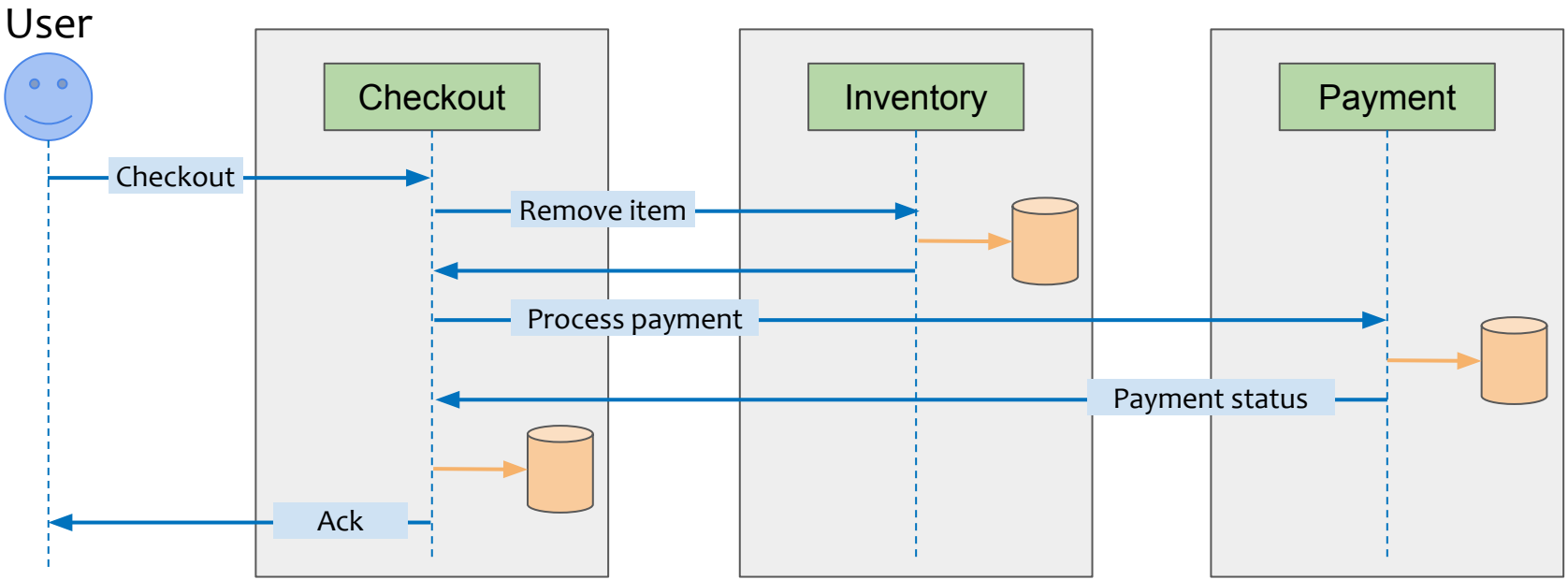
In a monolith, transactions are atomic



Refactoring communication (2)

With microservices, transactions become distributed, leading to:

- Consistency issues (multiple databases)
- Increased latency (more network interactions)



Ensuring consistency

Two commonly used patterns:

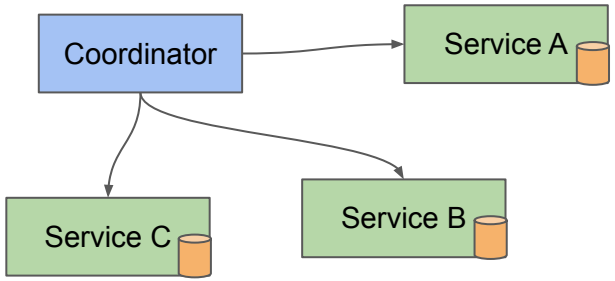
- Two-Phase Commit protocol (2PC)
- Saga pattern

Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

Prepare phase:

1. Coordinator queries services to commit their local transaction

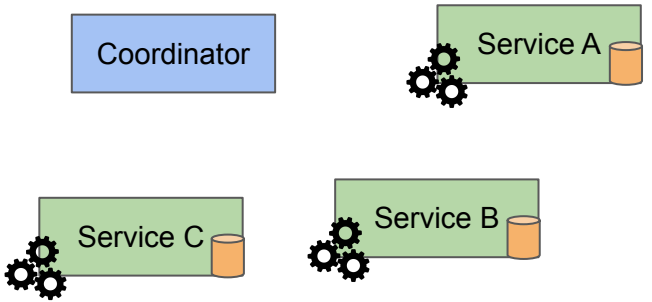


Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

Prepare phase:

1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage

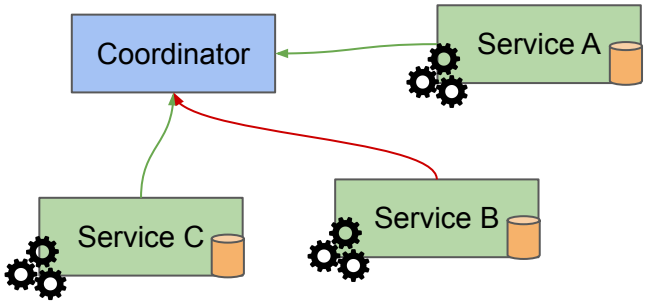


Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

Prepare phase:

1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage
3. Participants vote *yes* (success) or *no* (failure)



Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

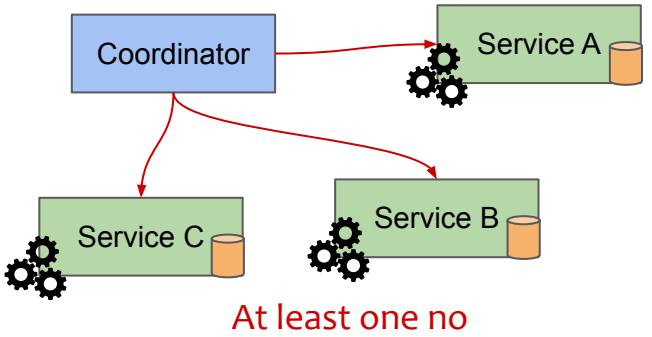
Prepare phase:

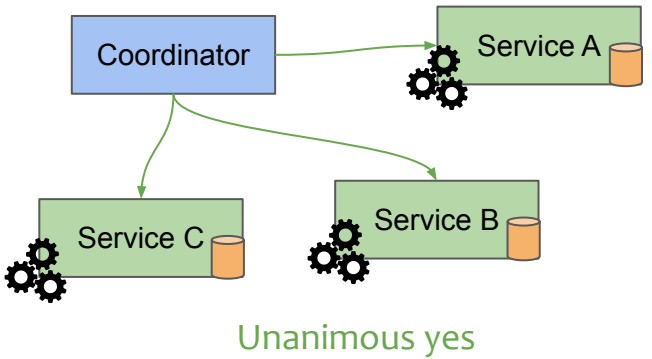
1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage
3. Participants vote *yes* (success) or *no* (failure)

Commit phase:

Vote result: Unanimous yes/at least one no

4. Coordinator sends a *commit*/*rollback* message





Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

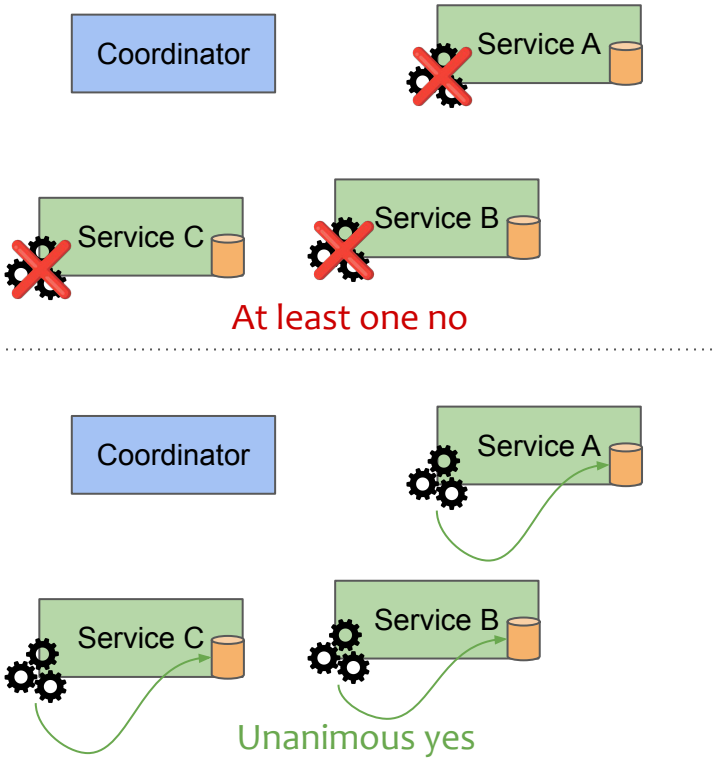
Prepare phase:

1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage
3. Participants vote *yes* (success) or *no* (failure)

Commit phase:

Vote result: Unanimous yes/at least one no

4. Coordinator sends a *commit*/*rollback* message
5. Participants commit the transaction to storage/rollback the transaction



Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

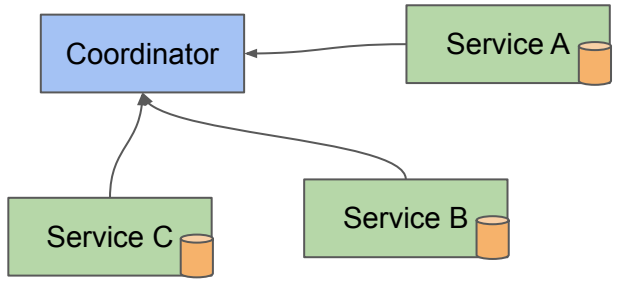
Prepare phase:

1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage
3. Participants vote *yes* (success) or *no* (failure)

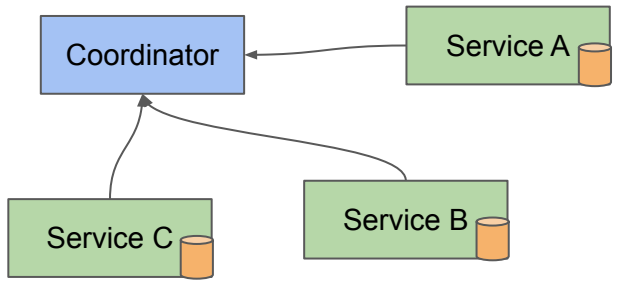
Commit phase:

Vote result: Unanimous yes/at least one no

4. Coordinator sends a *commit*/*rollback* message
5. Participants commit the transaction to storage/rollback the transaction
6. Participants reply with an acknowledgment



At least one no

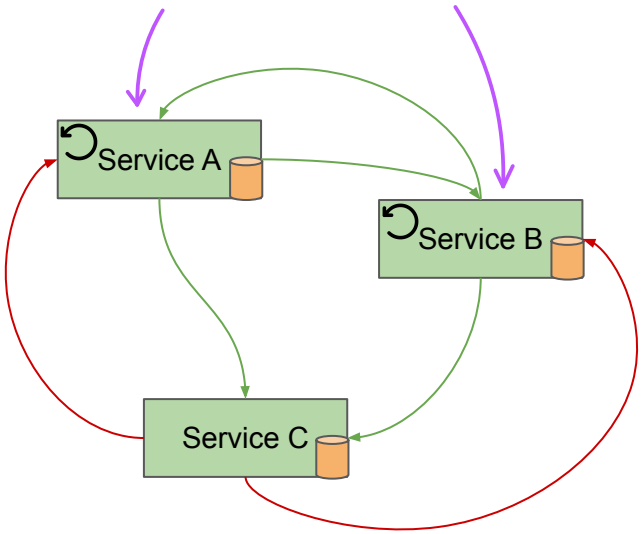


Unanimous yes

Saga pattern

Every microservice runs a local transaction and report back success/failure to all microservices involved.

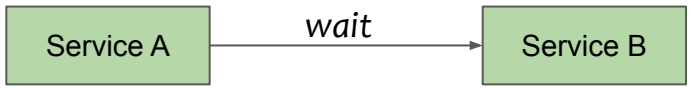
In case of a failure, microservices that already performed their local transaction perform *compensating actions*, i.e., undo the local changes.



Asynchronous communication

We can minimize latency with *asynchronous communication*

Instead of waiting for the reply,

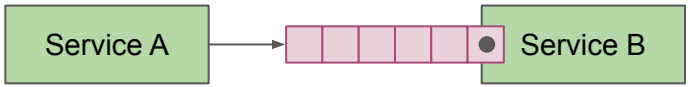


Communication synchronicity

We can minimize latency with asynchronous communication

Instead of waiting for the reply, we can

- Send the request to a *message queue*

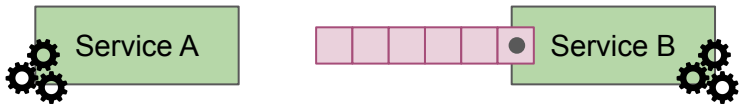


Communication synchronicity

We can minimize latency with asynchronous communication

Instead of waiting for the reply, we can

- Send the request to a *message queue*
- Do something else

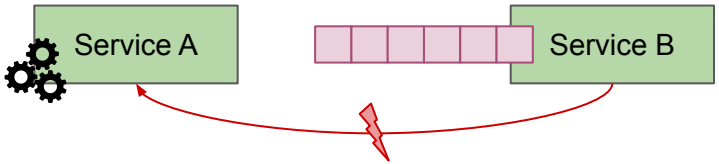


Communication synchronicity

We can minimize latency with asynchronous communication

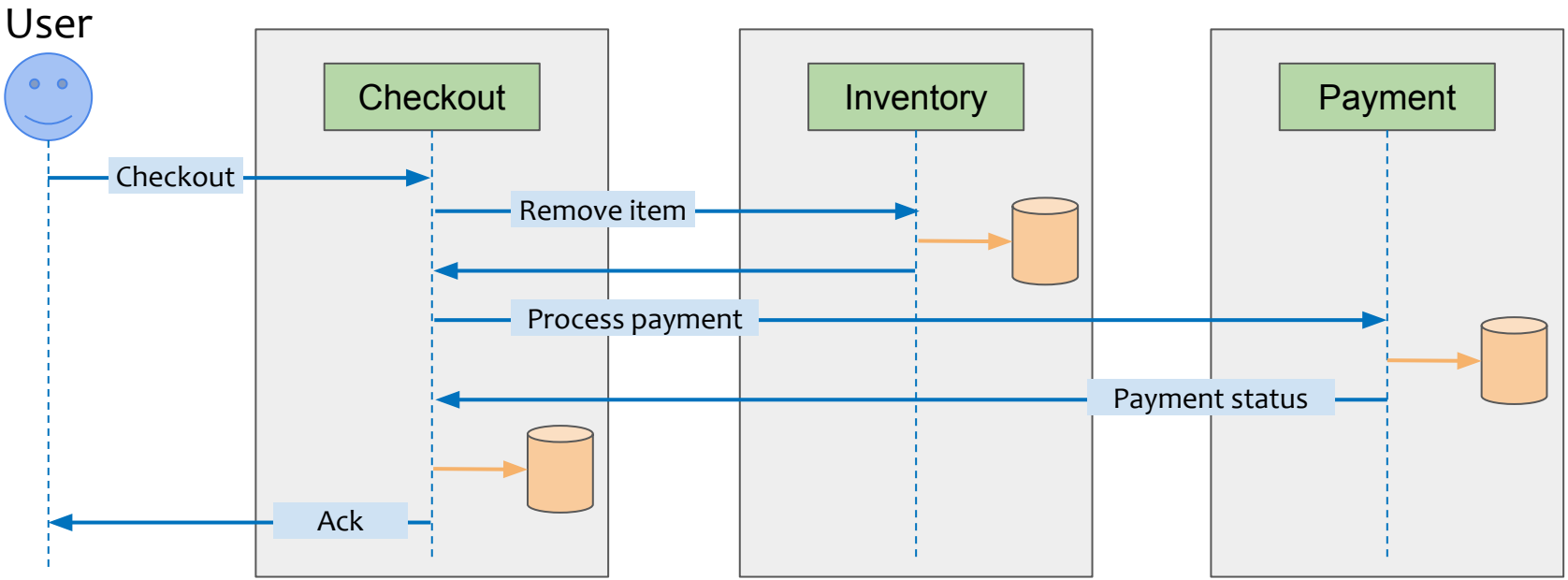
Instead of waiting for the reply, we can

- Send the request to a *message queue*
- Do something else
- Get notified when the request has been processed



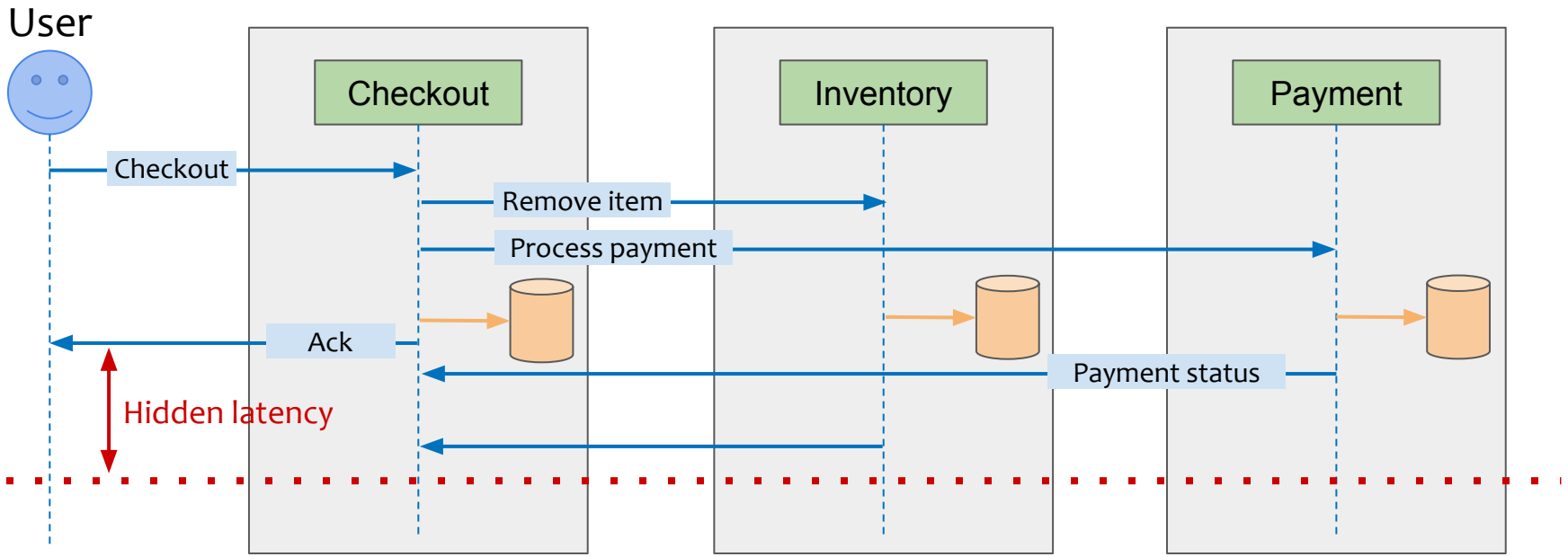
Communication synchronicity

Microservices can interact asynchronously to minimize waiting time



Communication synchronicity

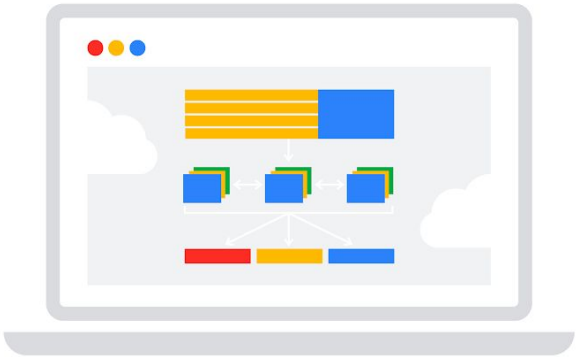
Microservices can interact asynchronously to minimize waiting time



Case study with Google Service Weaver

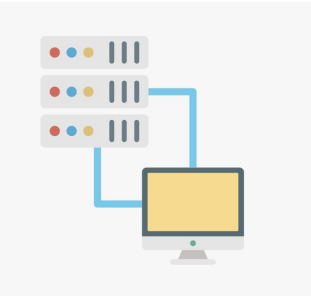
Service Weaver is a programming framework released by Google in March 2023

- Split your components as regular Go functions
- Implement Go function calls, the framework handles network communication
- Automated deployment (local or on private/public clouds)
- Configuration file for placement, replicas, etc…
- Provides logging, tracing, metrics



Check out https://serviceweaver.dev/

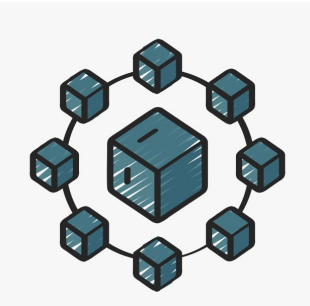
Software architectures



Client-server

Three-tier

Monolithic

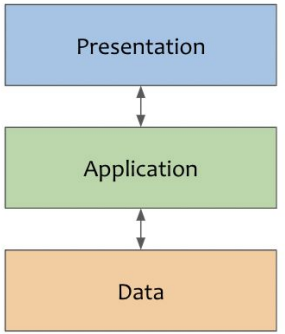


Microservices

Software architecture are composable!

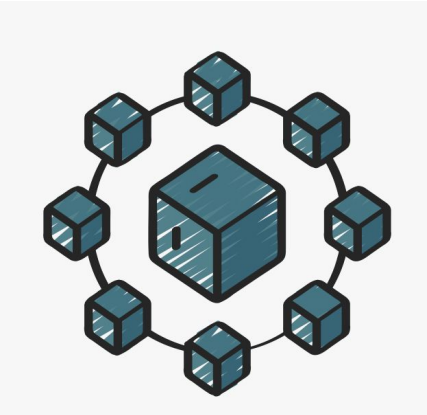
- For example, a monolithic service can be designed as a layered architecture

+



Software architecture are composable!

- For example, a microservice can be designed as a layered architecture



+

References

- **Building Microservices - Designing Fine-Grained Systems - Second Edition**
  Sam Newman, O'Reilly
- **Introduction to microservices** and **Refactoring a monolith into microservices**
  Google Cloud Architecture Center
- **Kubernetes documentation**
- **Google Service Weaver:** Blog post and documentation

Summary

- **Part I:** Requirements engineering
  - Requirement types (functional/non-functional)
  - Stages in requirements engineering
  - Non-functional requirements in the cloud
- **Part II:** Software architectures in the cloud
  - Overview
  - Client-server architecture
    - Communication layers (REST and gRPC)
    - Serialization and deserialization of structured data using Protbuf
  - Monolithic architecture
  - Microservice architecture
    - Strangler pattern: From monoliths to microservices