

Modern software architectures

Tapti Palit

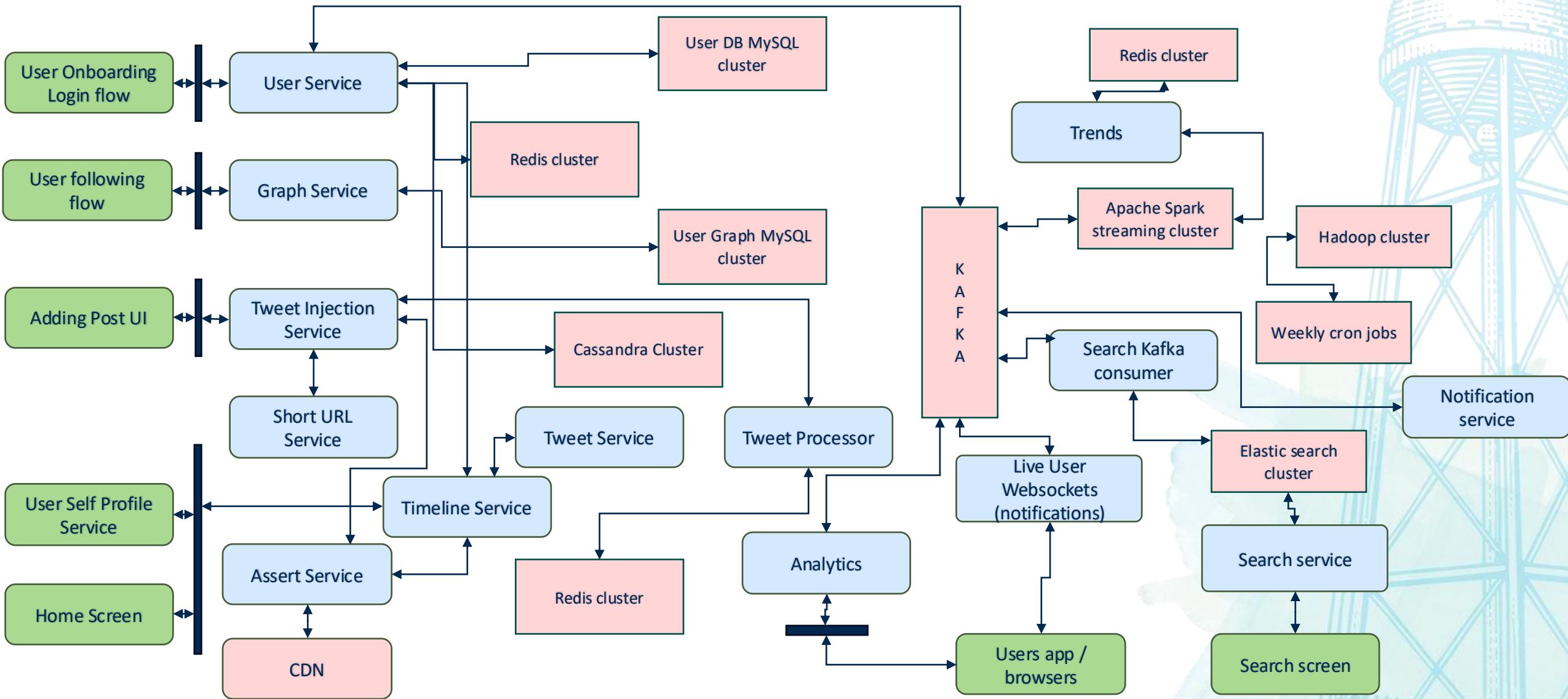


Outline

- Microservice architecture and communication
- Event-driven architecture with Kafka
- Kubernetes



Modern software architecture



Microservices outline

- Monolithic applications
- Microservices and decentralized data
- Data model and storage engine
 - Relational databases, log-structured merge trees (LSM), event logs (more in Kafka section), in-memory cache
- Communication styles
 - Text-based vs binary data exchange formats
 - Synchronous (RPC), asynchronous (MQs), publish-subscribe models

All about the data!

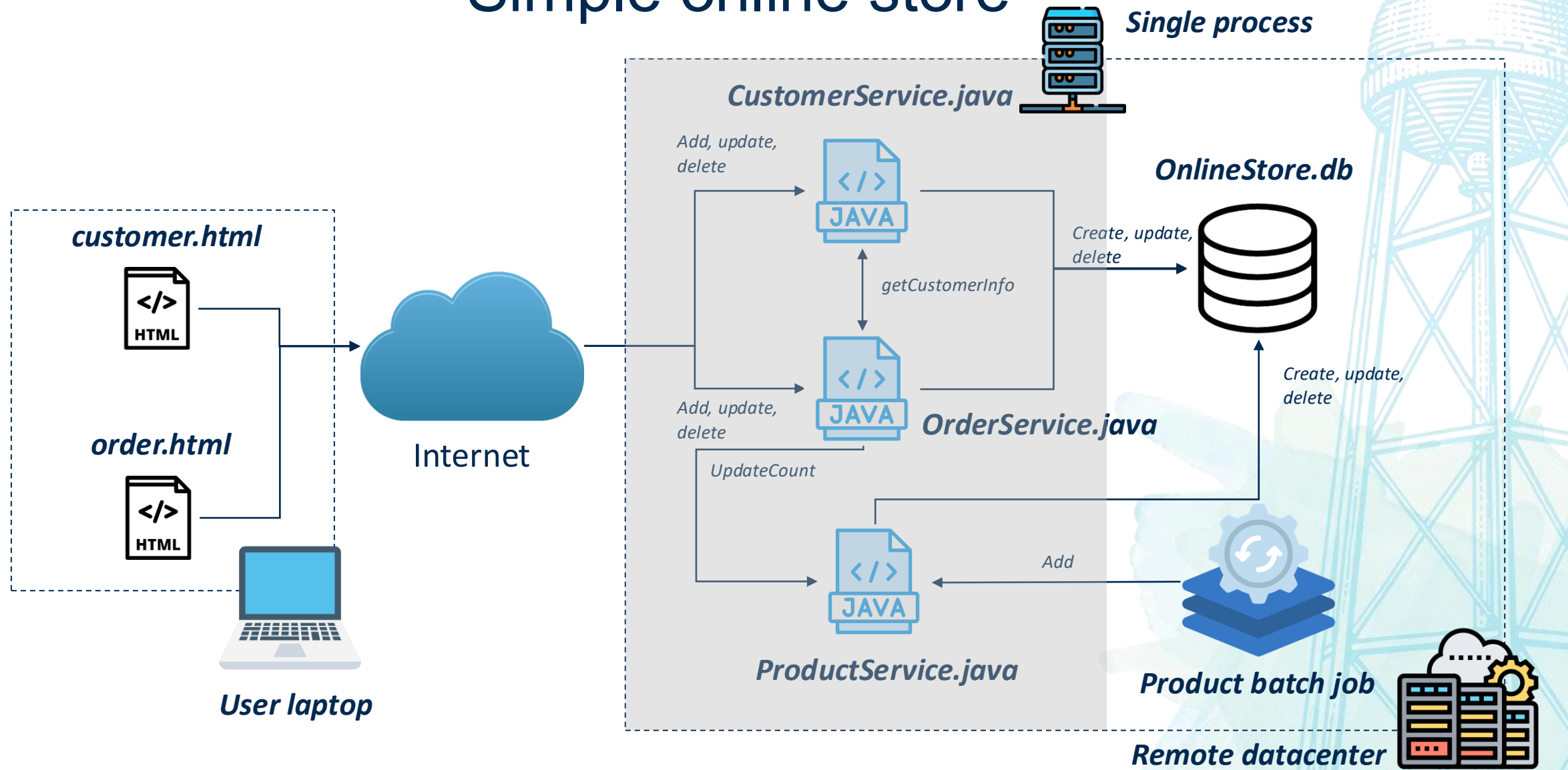
- Modern software architecture is data-intensive
- Primary concerns
 - Who owns data?
 - How to optimally store data?
 - How is data shared?
 - How to limit overhead due to data-sharing?



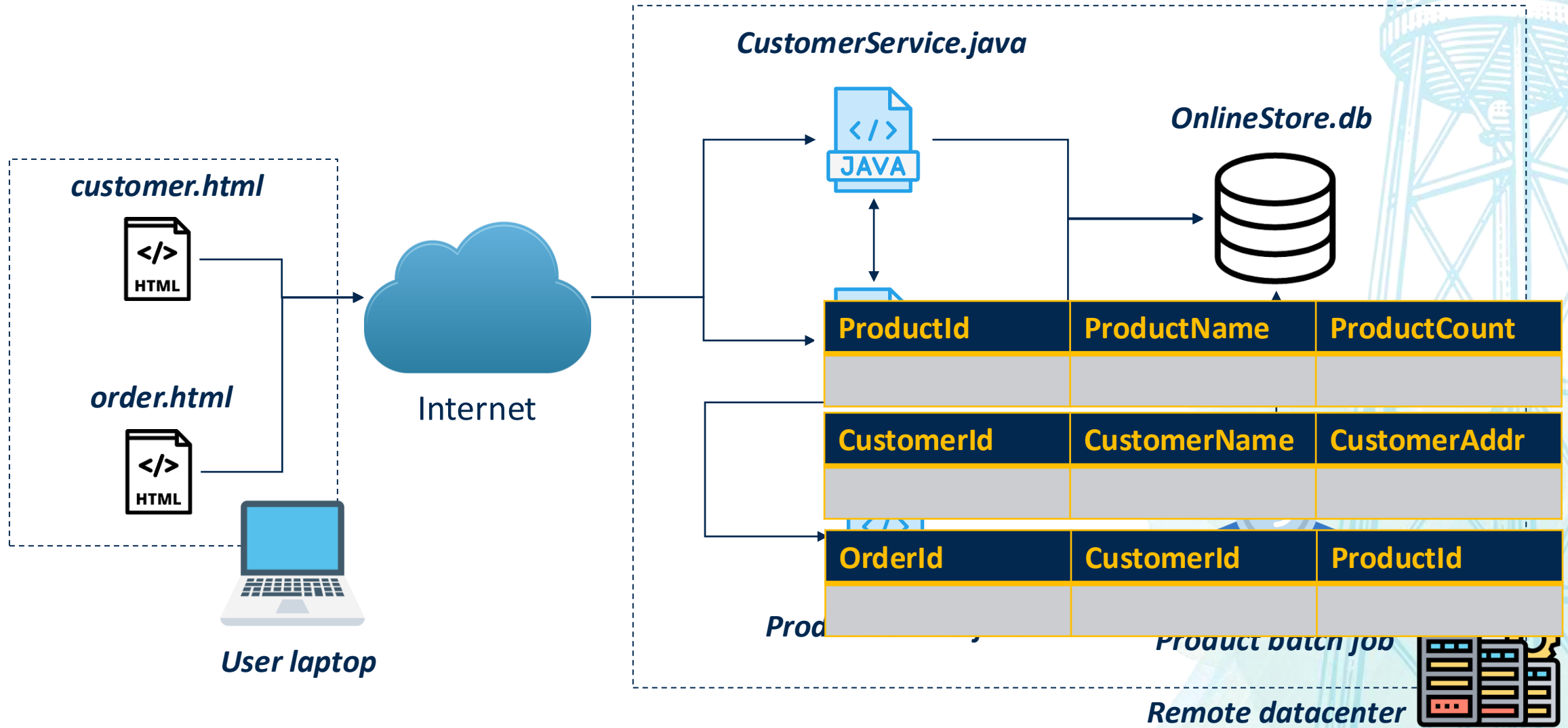
Monolithic software architecture

- Widely used in early, mid 2000s
- All software components are part of the same application
- All software components run on the same machine, in same process
- Typically developed in the same language stack

Simple online store



Simple online store



Relational databases

- Consists of tables
- Each table contains a primary key
- Database will not allow insertion of two records with same primary key

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

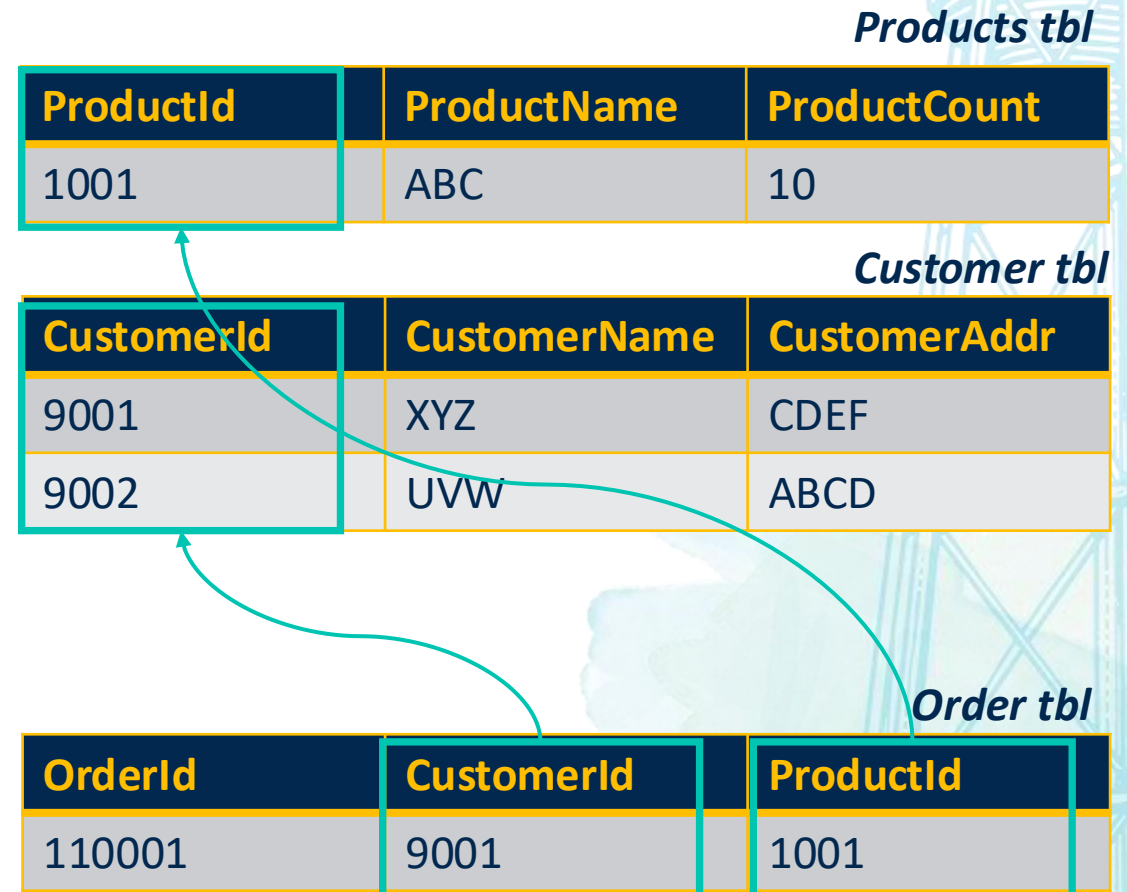
CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF

Order tbl

OrderId	CustomerId	ProductId
110001	9001	1001

Foreign keys

- Relational databases maintain relations through foreign keys
- Foreign keys ***must refer*** to primary keys of other tables
 - Enforce referential integrity
- A table can contain one or more foreign keys

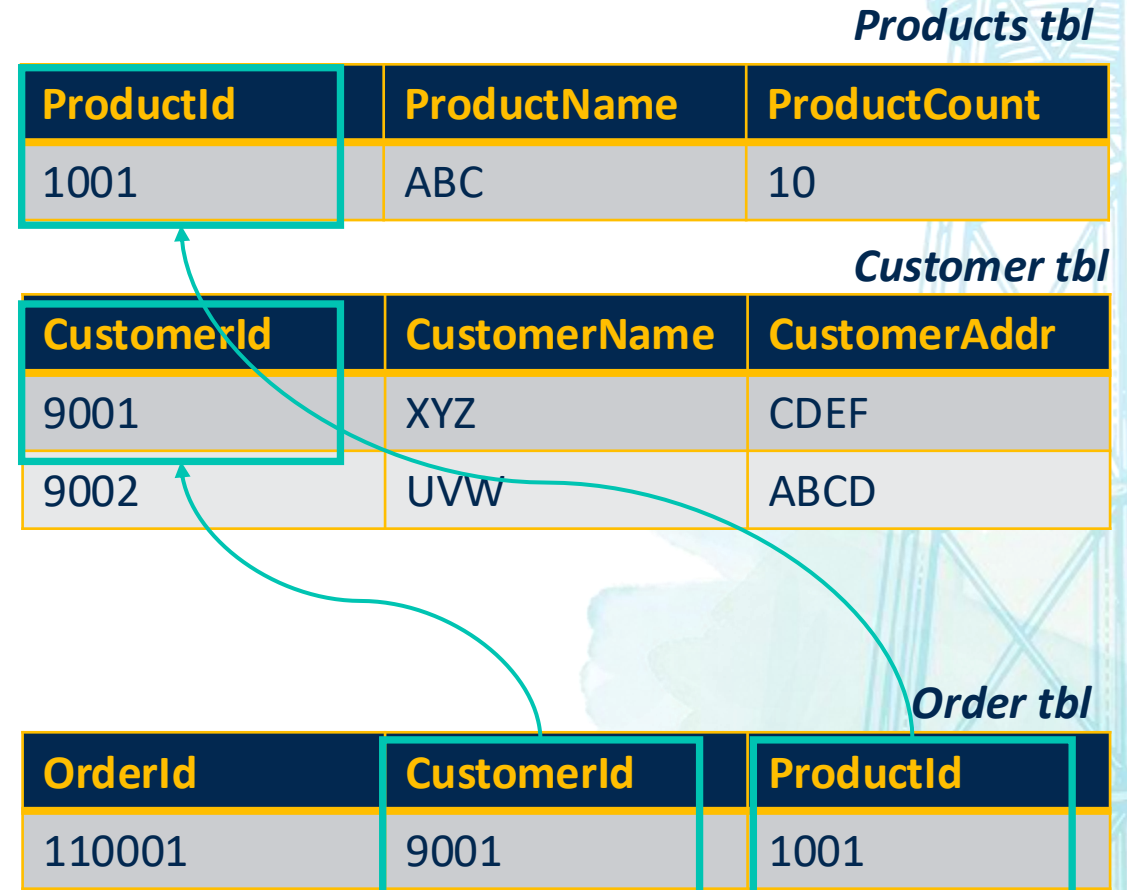


Structured query language (SQL)

- SQL used to interface between application and database
- CREATE TABLE Products (
 ProductId INT PRIMARY KEY,
 ProductName VARCHAR(100) NOT NULL,
 ProductCount INT NOT NULL);
- INSERT INTO Products (ProductId, ProductName,
 ProductCount) VALUES (1001, 'ABC', 10);

SQL joins

- Allows table joins
- For e.g. find order details for shipping, including product name, customer name, and address



SQL joins

SELECT

o.OrderId,
p.ProductName,
c.CustomerName,
c.CustomerAddr

```
FROM Orders o
JOIN Products p
  ON o.ProductId = p.ProductId
JOIN Customers c
  ON o.CustomerId =
     c.CustomerId;
```

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	CustomerId	ProductId
10001	9001	1001

o.OrderId	p.ProductName	c.CustomerName	c.CustomerAddr
110001	9001	XYZ	CDEF

Need for joins

- Imagine no support for joins
- Order information must contain product name, customer name, and customer address to ship the order

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	ProductName	CustomerName	CustomerAddr
110001	ABC	XYZ	CDEF

Need for joins

- Lack of join support increases data duplication
- Data denormalization

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

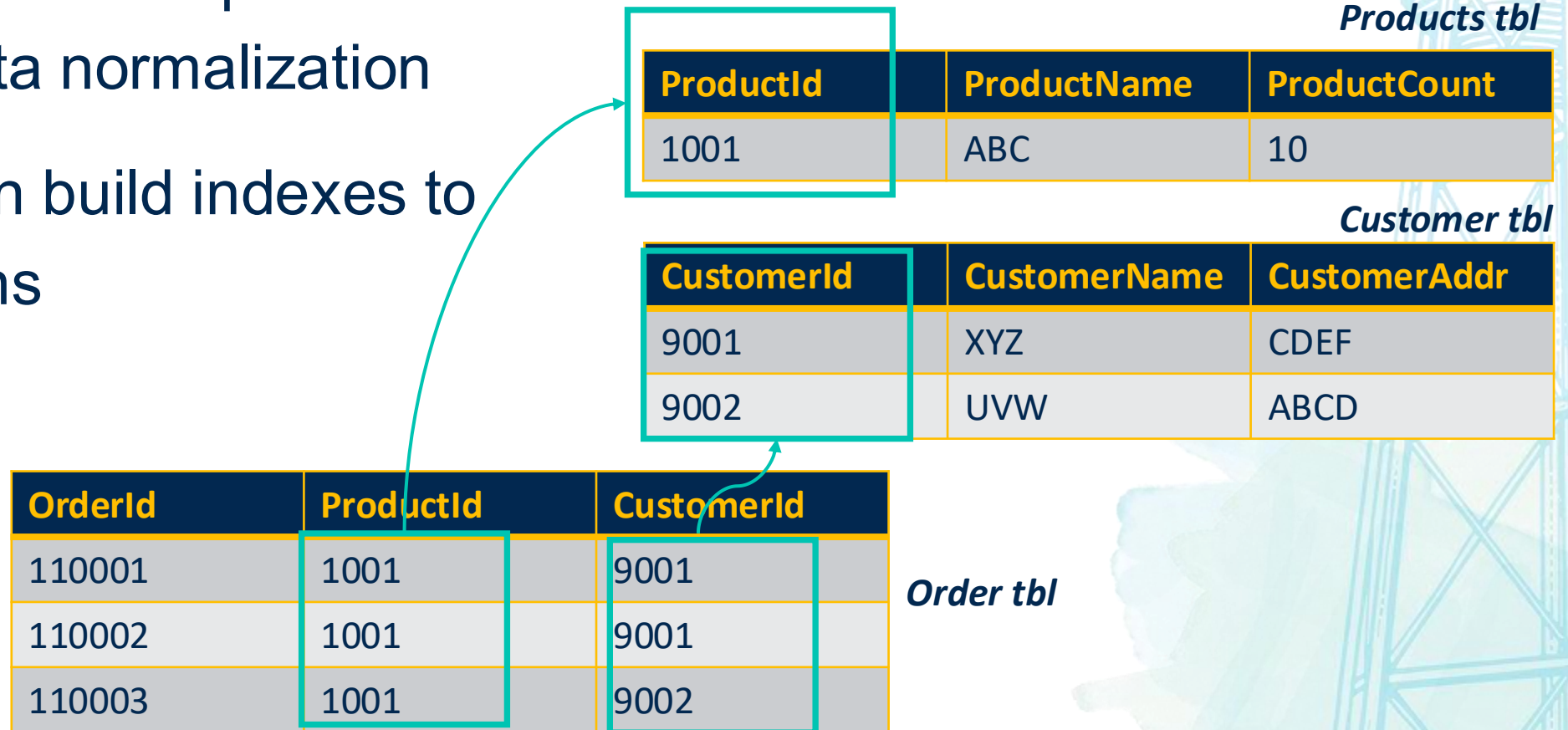
CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

OrderId	ProductName	CustomerName	CustomerAddr
110001	ABC	XYZ	CDEF
110002	ABC	XYZ	CDEF
110003	ABC	UVW	ABCD

Order tbl

Normalization

- Joins reduce data duplication and allow data normalization
- Database can build indexes to speed up joins



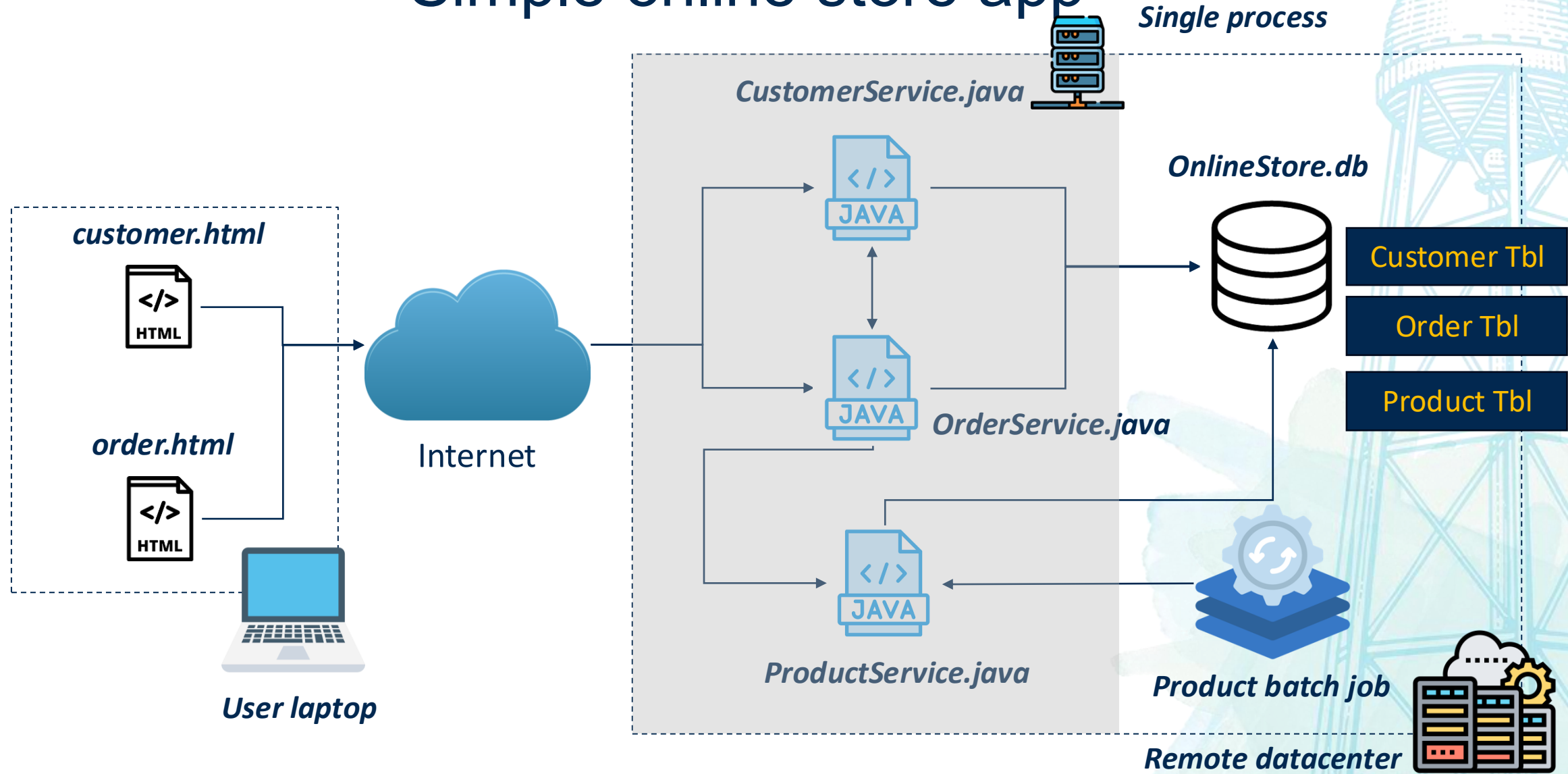
Relational database implications

- Tighter coupling
 - E.g., Customer service cannot alter the Customers table without impacting the Orders service
- Fixed schema
 - E.g., Orders service must know about the schema Customers table

SQL != relational databases

- Note: not quite specific to relational databases
- Apache Cassandra, Apache HBase, Apache Kafka + ksqlDB are not relational databases, but support SQL-like query language

Simple online store app

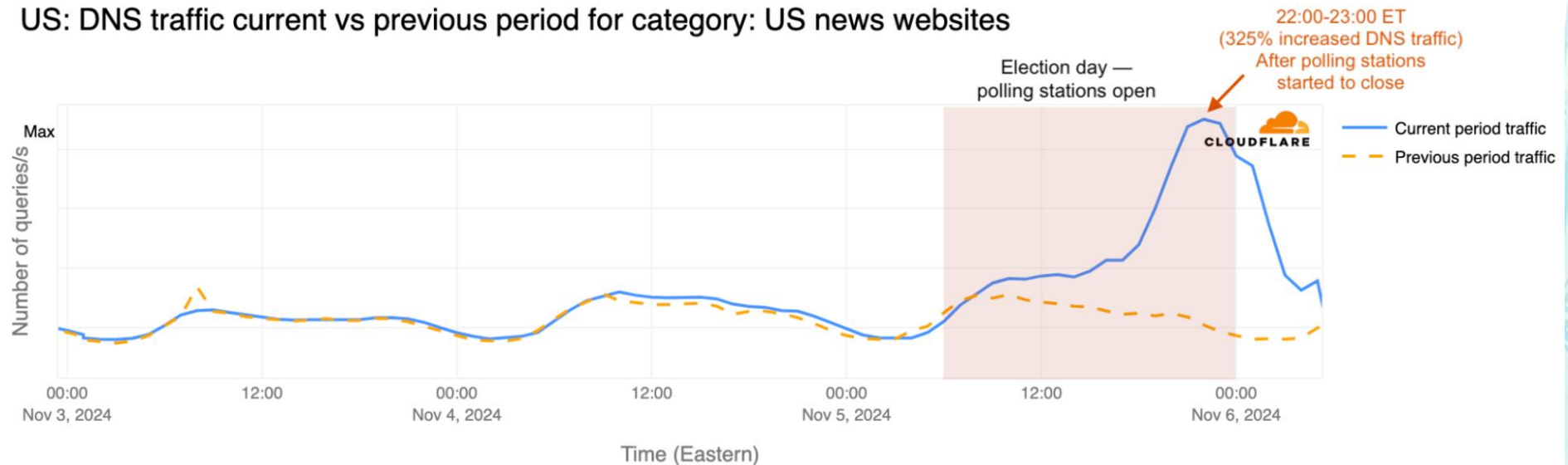


Scalability

- Online traffic is variable
- Can spike due to planned events
 - Product launches
 - Political events

<https://blog.cloudflare.com/exploring-internet-traffic-shifts-and-cyber-attacks-during-the-2024-us-election/>

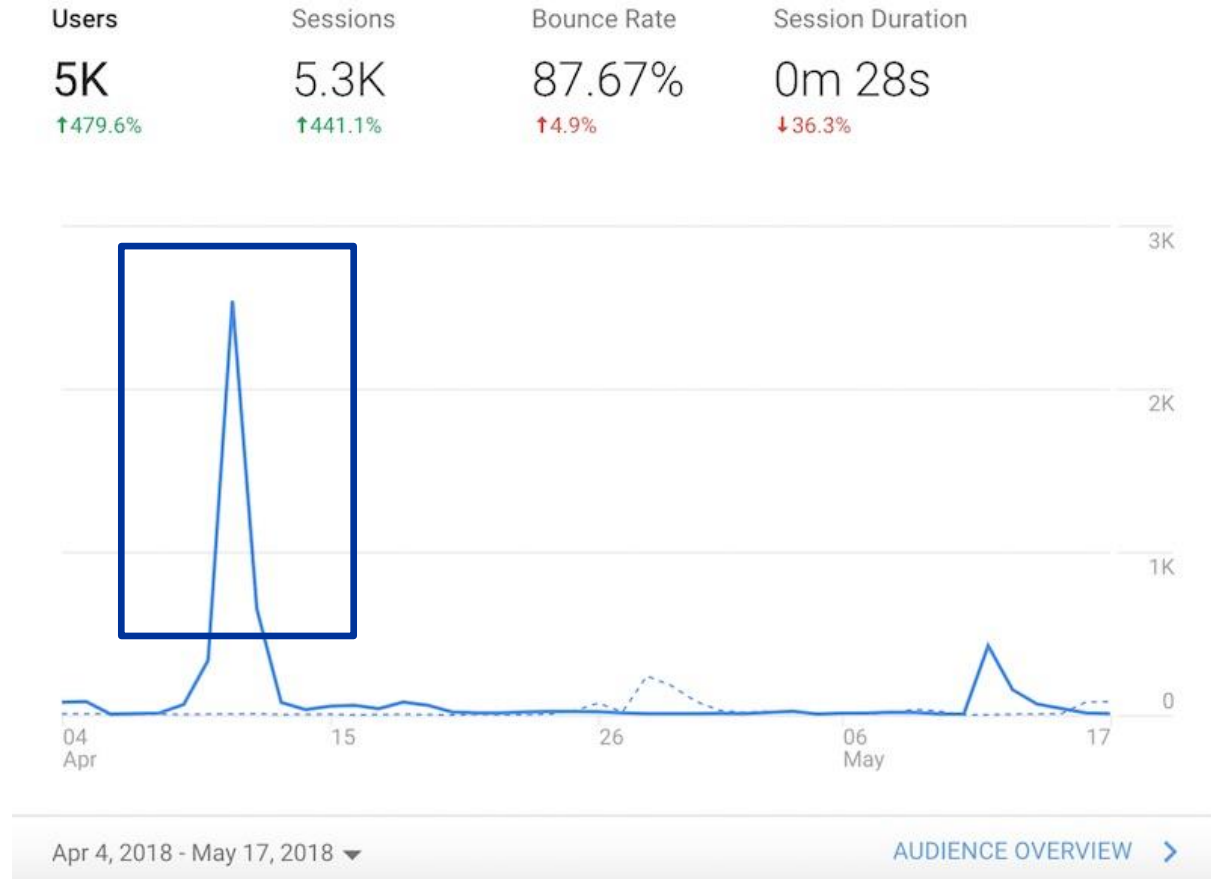
US: DNS traffic current vs previous period for category: US news websites



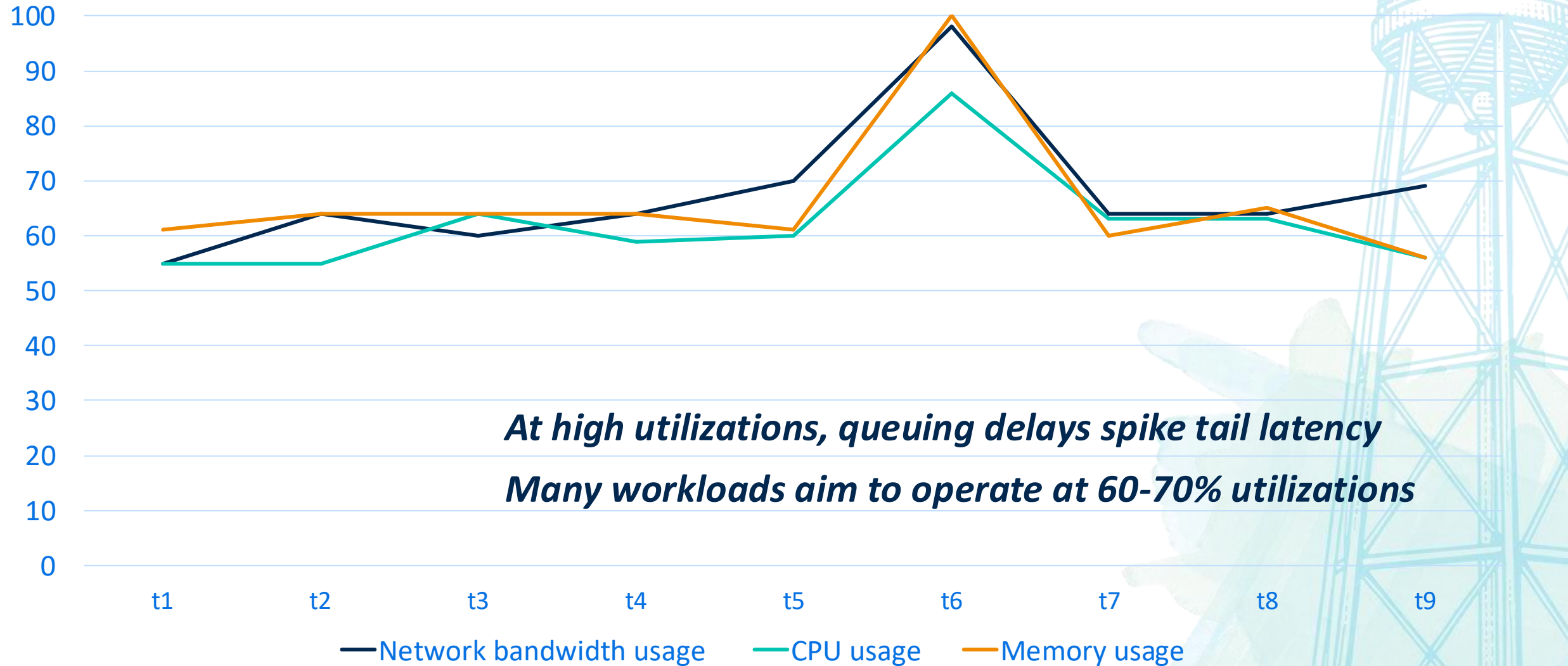
Scalability

- Unplanned events
 - Blog post goes viral
- System design should *adapt* to handle such events

<https://www.residualthoughts.com/2018/05/20/traffic-data-from-a-viral-post/>

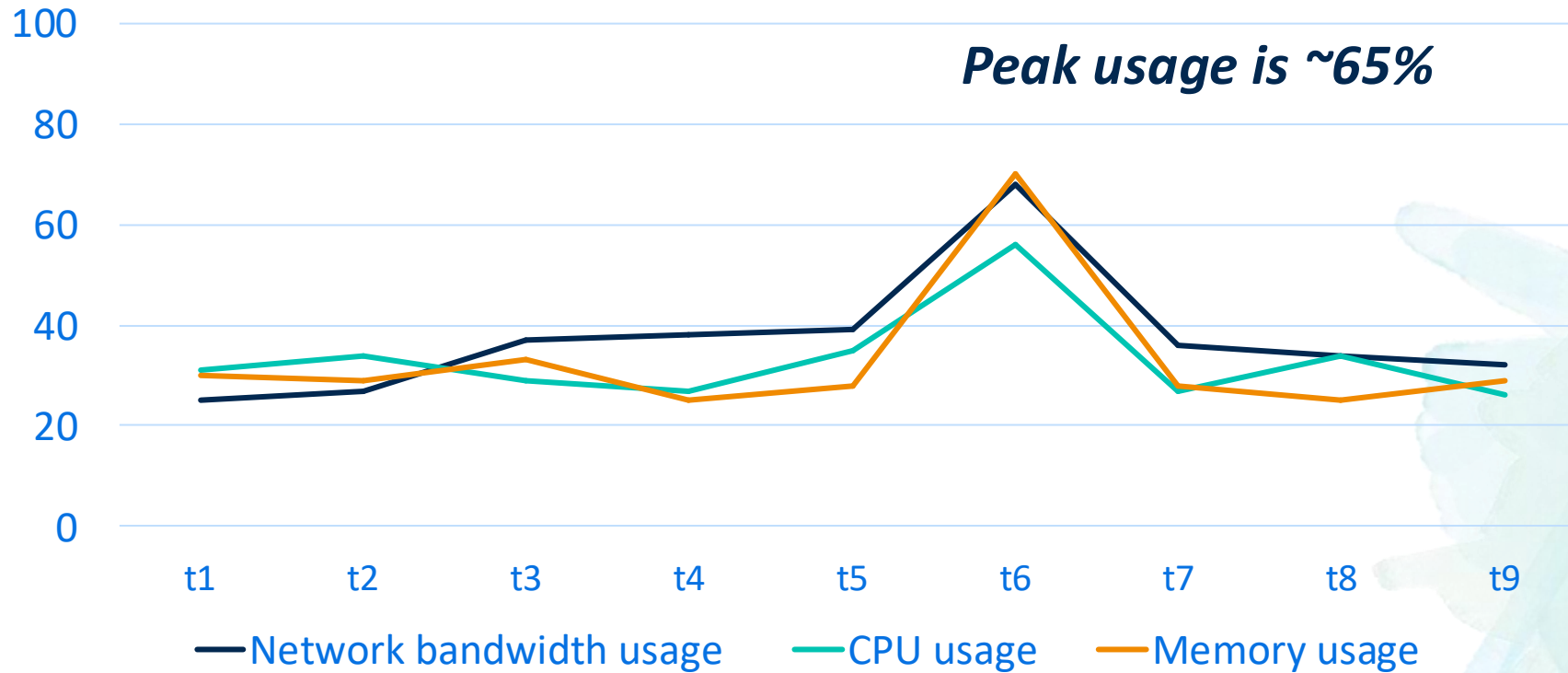


Traffic spike to online store



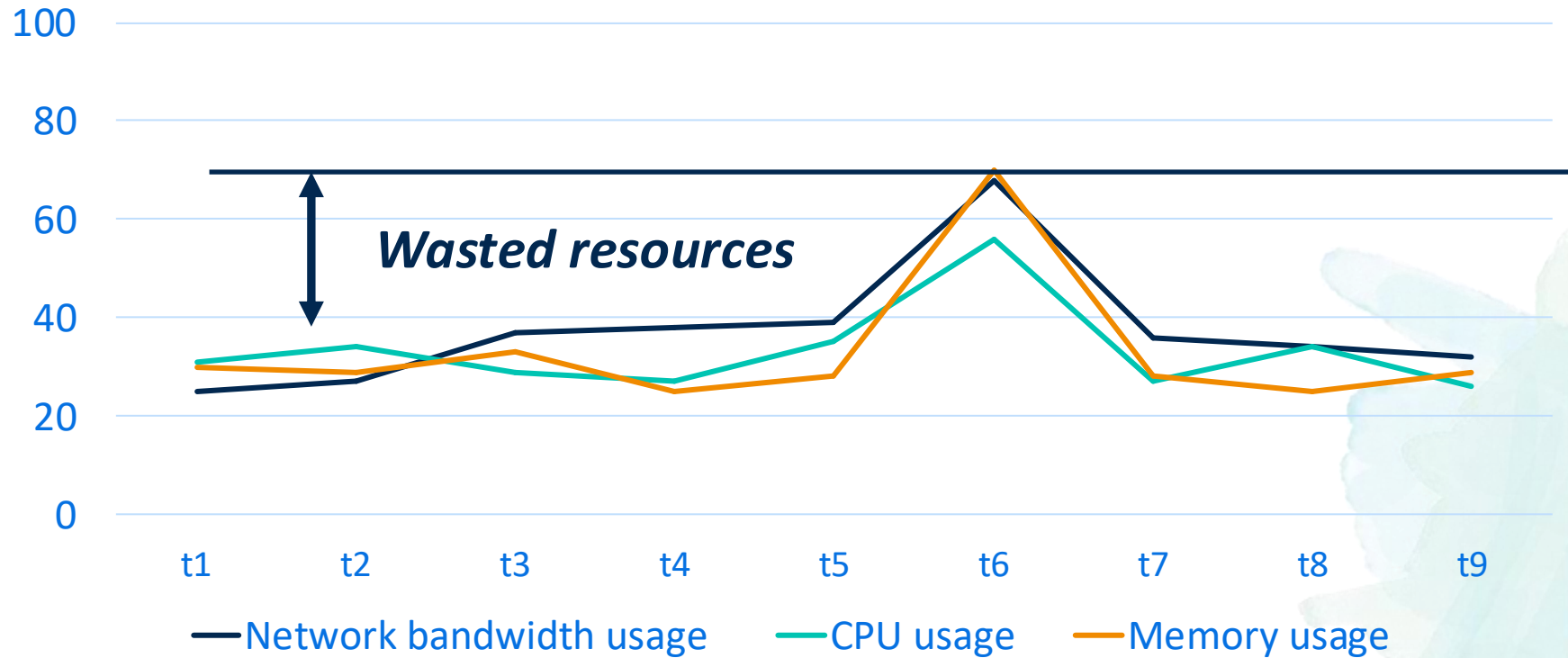
Vertical scaling

- Use more powerful machines
- Add more CPUs, DRAM, network bandwidth



Vertical scaling limitations

Resource wastage when requests are not bursty

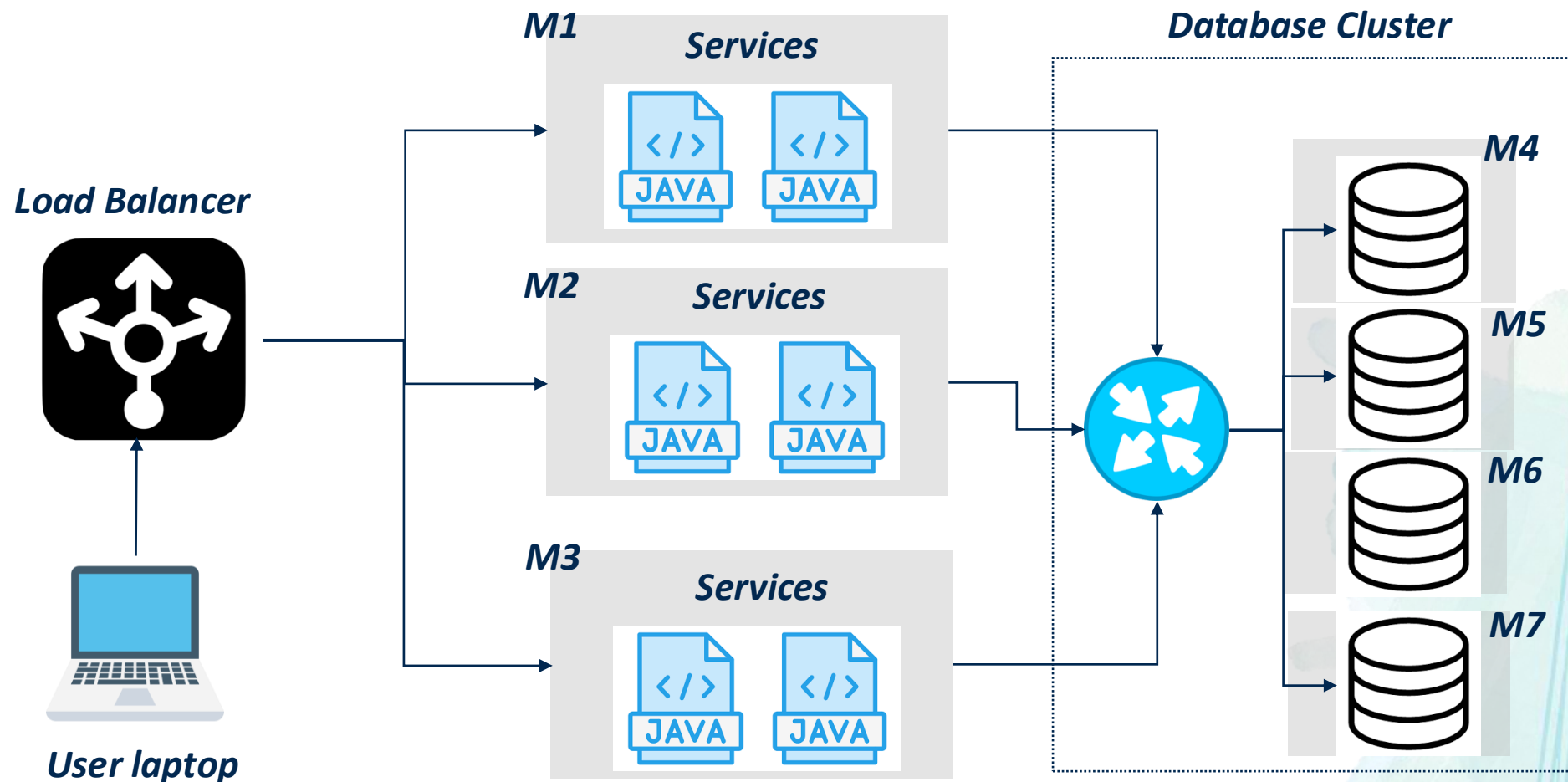


Vertical scaling limitations

- Twitter has 200M-300M active users per day
- No single machine, no matter how powerful, can support such high loads
- Goal: autoscaling
 - **Dynamically** spawn new machines during **high loads**
 - Not possible using vertical scaling alone
 - More in Kubernetes module

Horizontal scaling for monolithic apps

Add more machines and replicate application on each machine

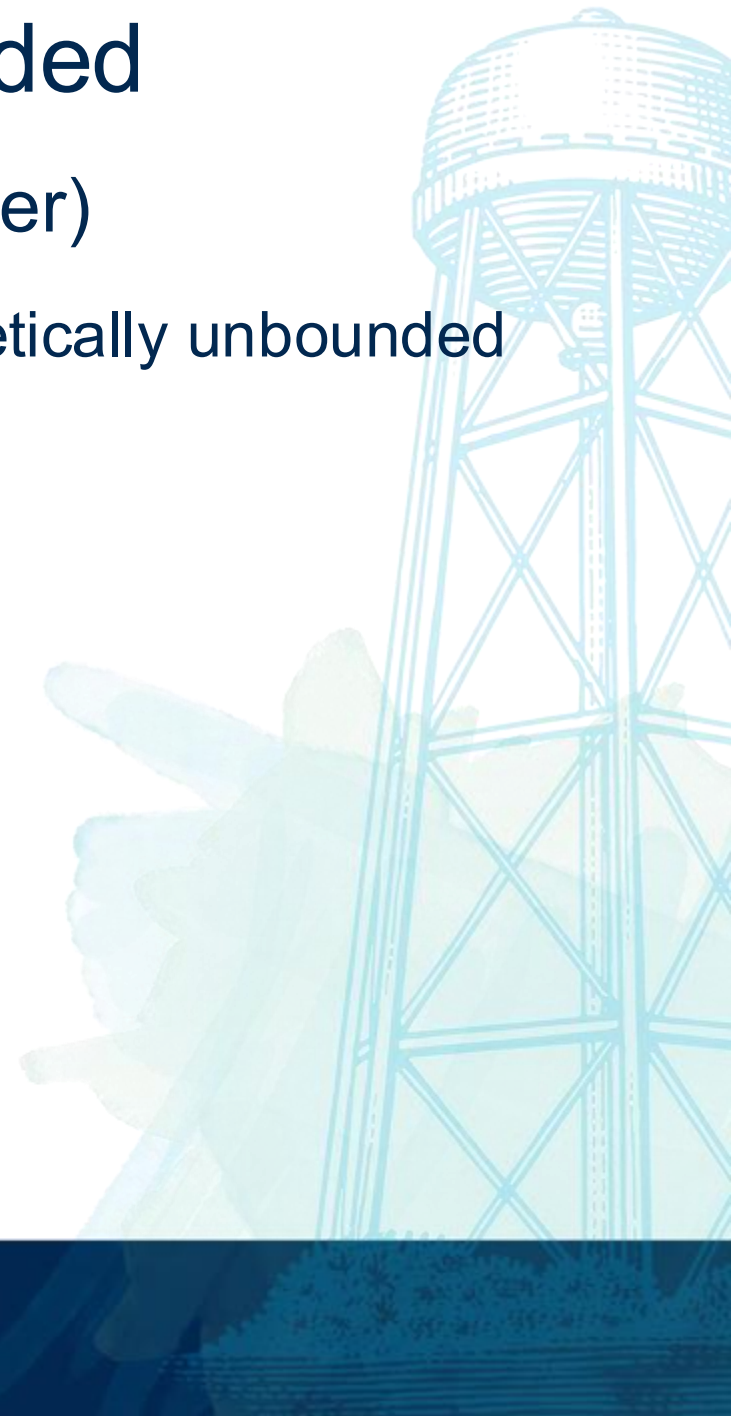


Load balancer

- Distributes incoming requests across multiple servers to improve scalability and availability
- Strategies
 - Round Robin – Sequentially routes requests across servers; simple but doesn't account for server load
 - Least Connections – Directs traffic to the server with the fewest active connections; adapts well to uneven load
 - Least Response Time – Chooses the server with the fastest response time and fewest connections; performance-oriented
 - Random Policy – Selects servers randomly; useful in stateless, uniform environments
 - Weighted Distribution – Allocates requests based on server capacity (e.g., CPU power, memory)
- Each strategy has tradeoffs
- More details in Kubernetes module

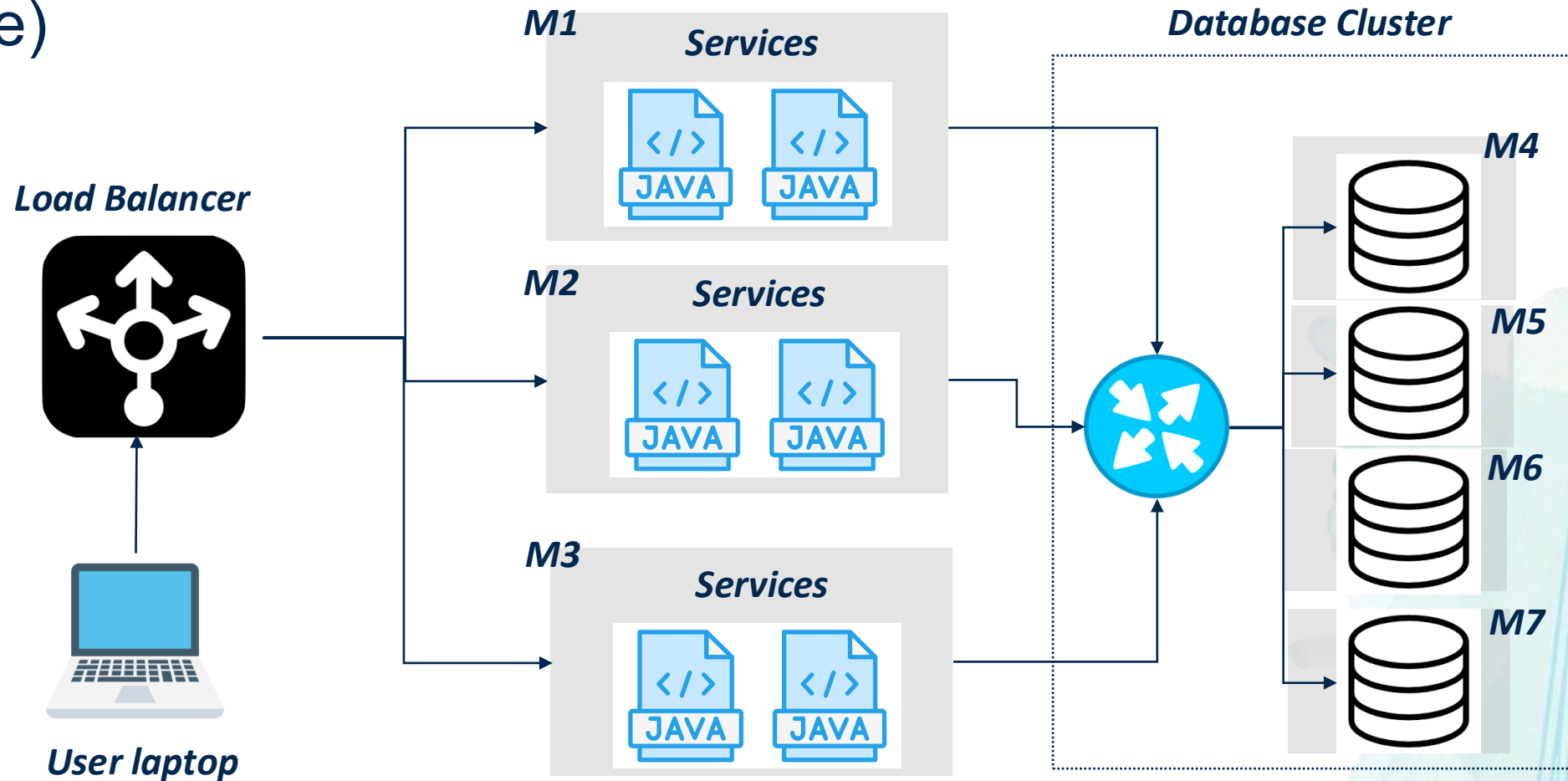
Horizontal scaling is unbounded

- Necessary for applications with high load (like Twitter)
 - Can add more machines to support more requests (theoretically unbounded performance)
- Better resource utilization



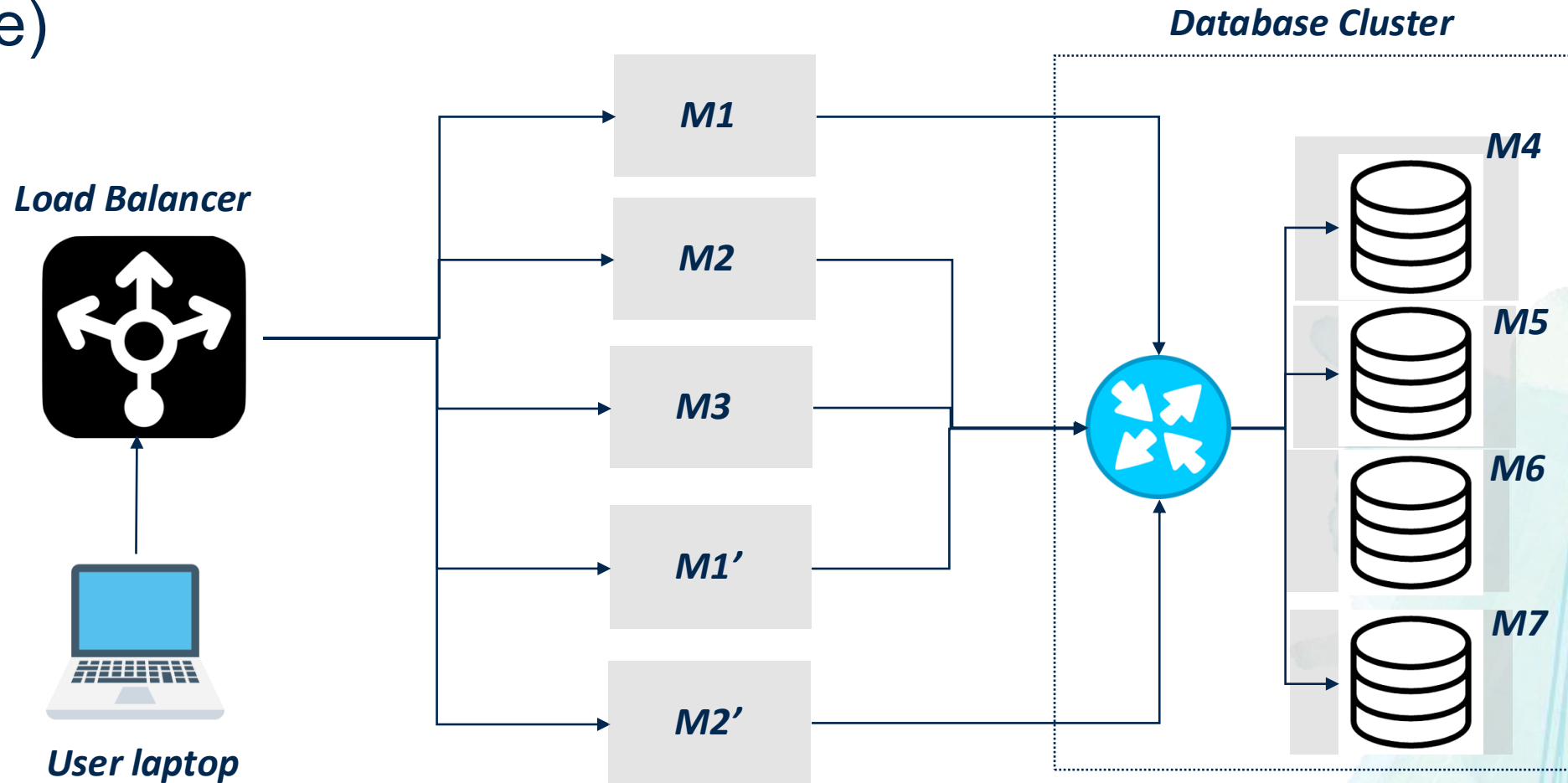
Horizontal scaling is necessary for autoscaling

Auto-scale to more machines during traffic spike (more in Kubernetes module)



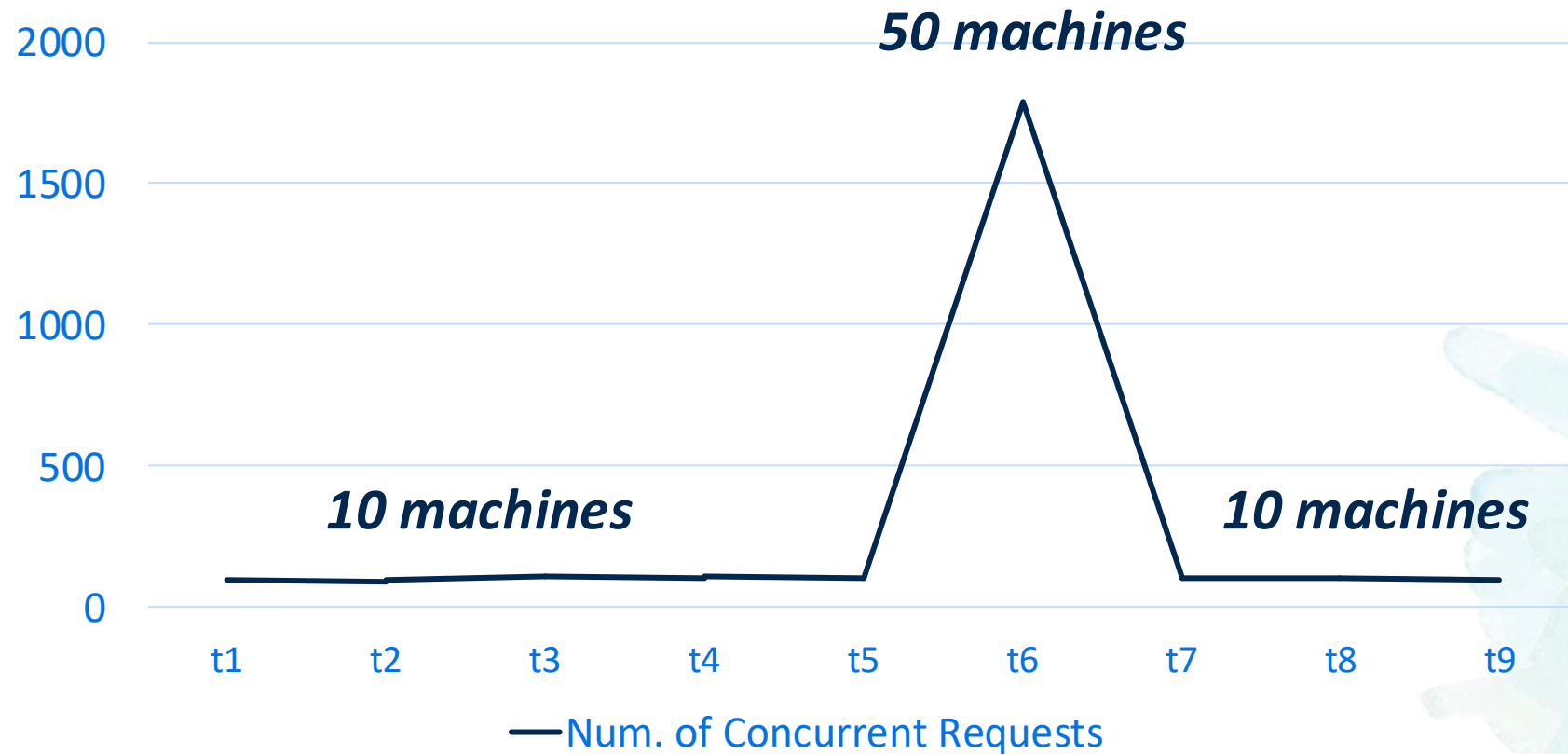
Horizontal scaling is necessary for autoscaling

Auto-scale to more machines during traffic spike (more in Kubernetes module)



Autoscaling

Minimum resource wastage



Horizontal scaling allows autoscaling, but did we solve all problems?

Heterogenous resource requirements across services

- Provisioning is driven by the most resource-hungry service
- Example RAM requirements
 - CustomerService: 32 GB
 - OrderService: 18 GB
 - ProductService: 16 GB
 - Minimum machine RAM? 32 GB

Deployability concerns

- Updating one component requires redeploying the entire application
- Reverting a change requires redeploying the entire application
- Slow, error-prone process

Need for low interdependence

- Software often consists of thousands of components
 - Each component has a dedicated team working on it
- Teams need to work independently
 - E.g., ProductService team should be able to update the Products Tbl schema without consulting other services
- Need low coupling between services
- Solution: microservices

Microservices

- An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API
- Independently deployable by automated processes
- Bare minimum centralized management
- Smart endpoints connected by “dumb” pipes

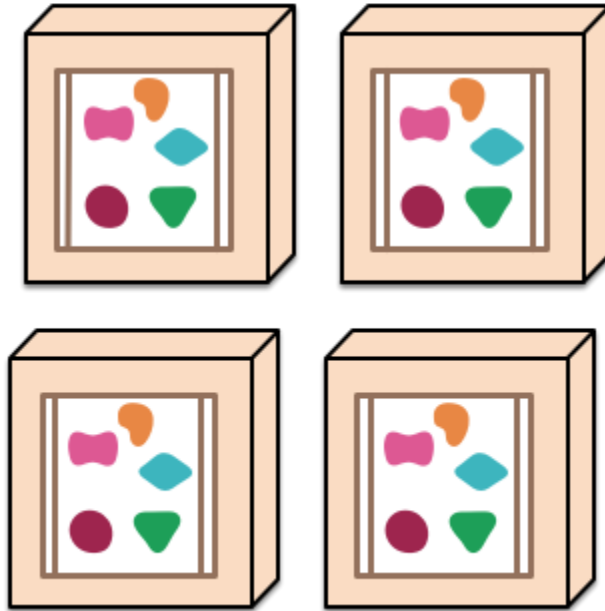
<https://martinfowler.com/articles/microservices.html>

Microservices overview

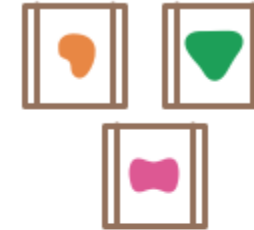
A monolithic application puts all its functionality into a single process...



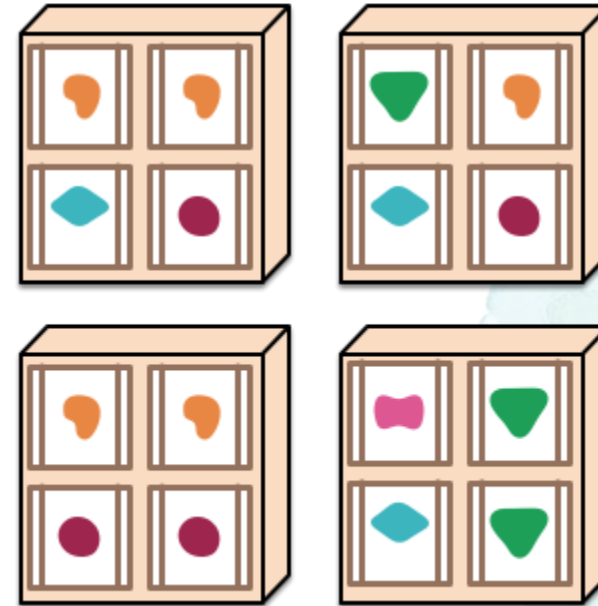
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

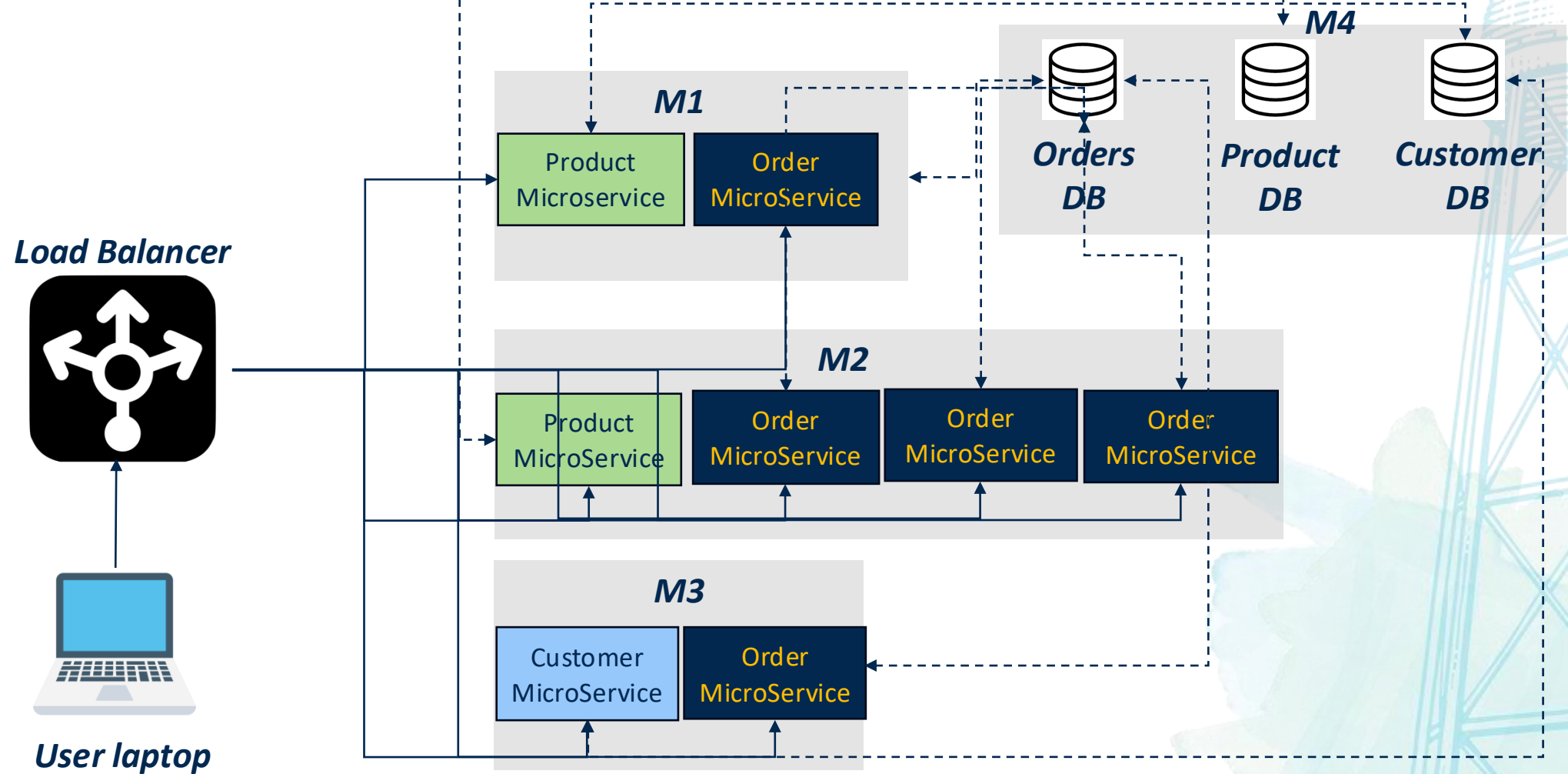


... and scales by distributing these services across servers, replicating as needed.

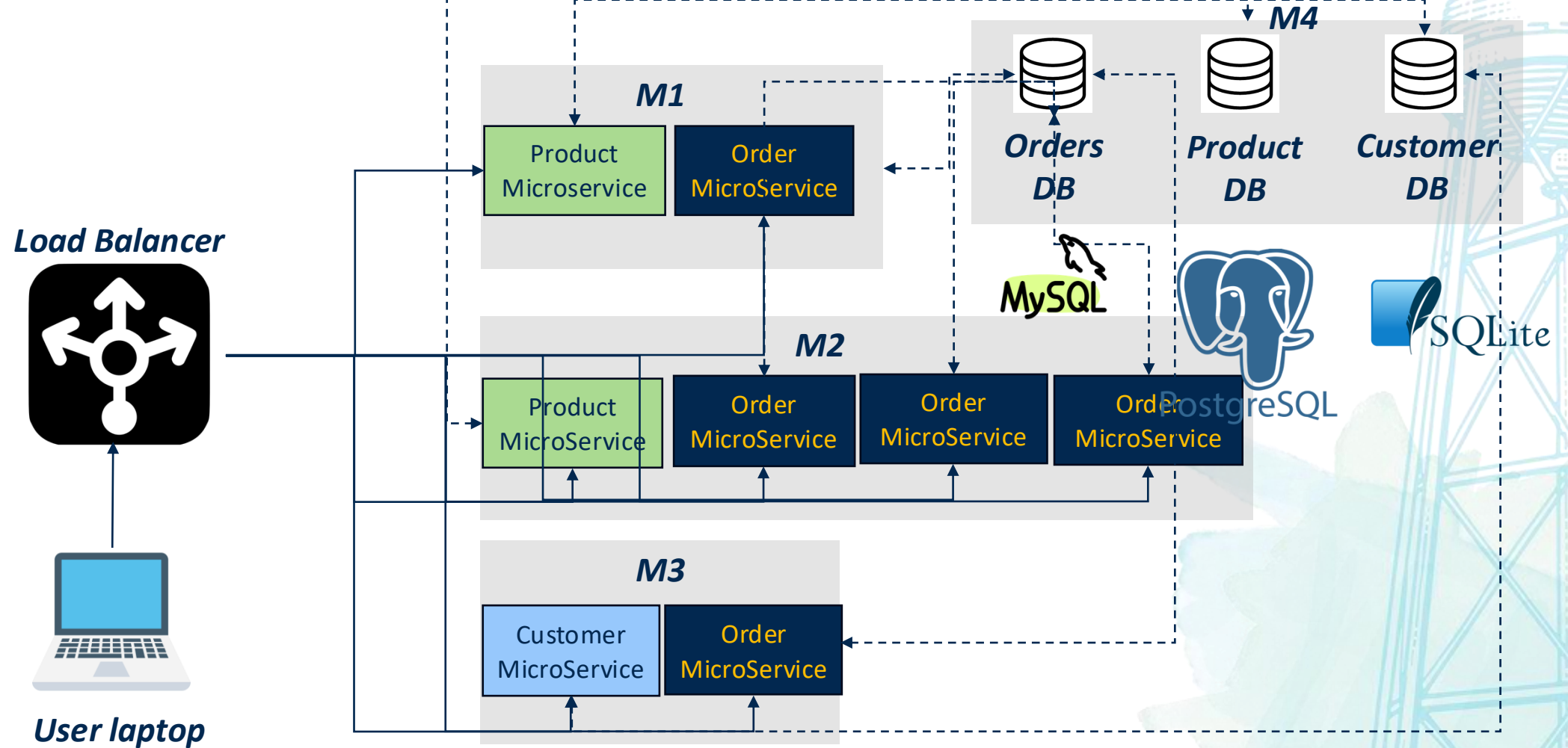


<https://martinfowler.com/articles/microservices.html>

Microservice architecture for online store



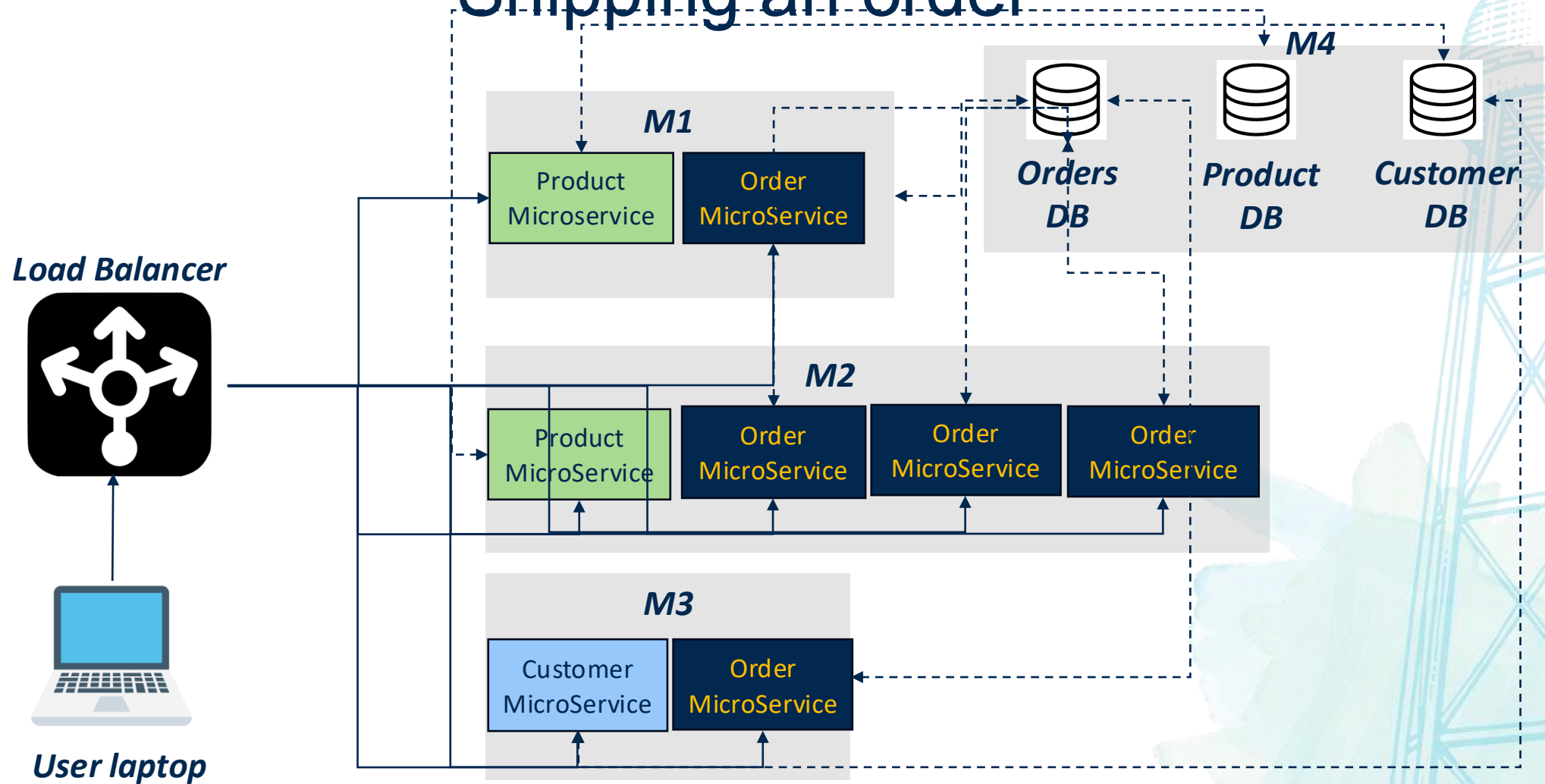
Each microservice chooses its own database



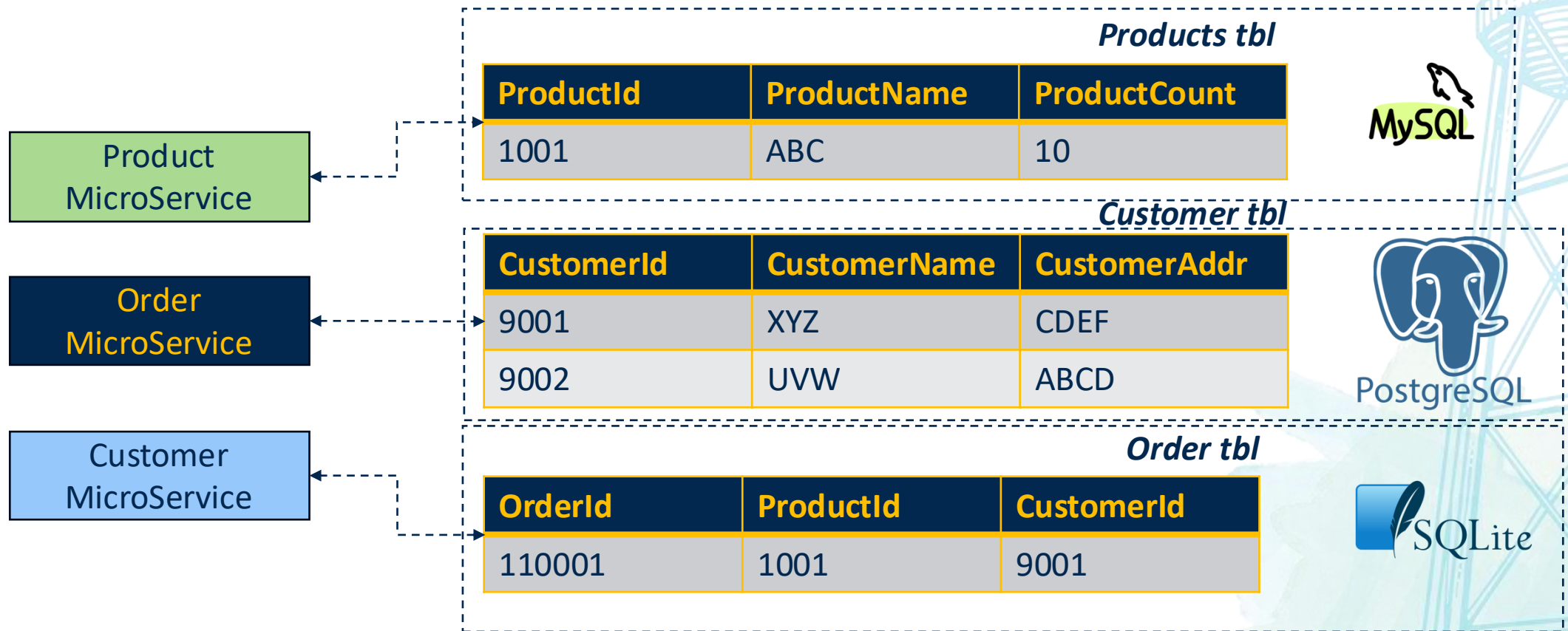
What did we achieve?

- Decompose monolithic app into microservices
- Improve decoupling
 - Each microservice can be scaled independently
 - Each microservice can be deployed independently
 - Each microservice can evolve independently – DB schema, choice of programming languages
- ***Did we solve all problems?***

Shipping an order



Reusing monolithic order schema



Non-solution

- DO NOT want to query the Customer and Product microservices during shipping
- Will introduce tight coupling and interdependence
 - For example, what if the Customer microservice is down during shipment?

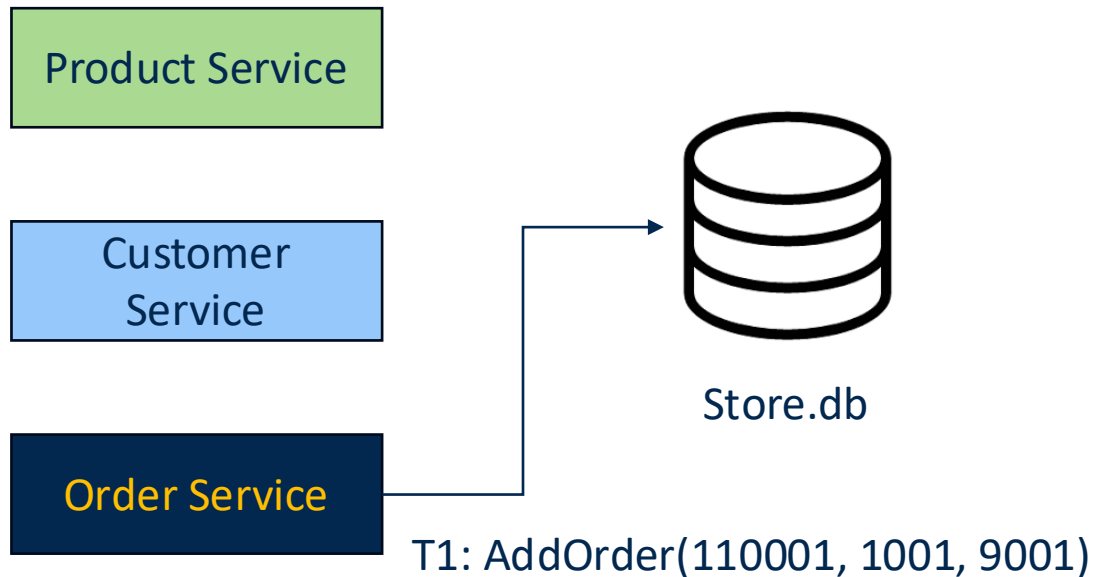
Denormalization

- Must denormalize the data
- Order microservice duplicates and store the customer details in the Order db
- How? (after a few slides)

OrderId	ProductName	CustomerName	CustomerAddr
110001	ABC	XYZ	CDEF
110002	ABC	XYZ	CDEF
110003	ABC	UVW	ABCD

Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



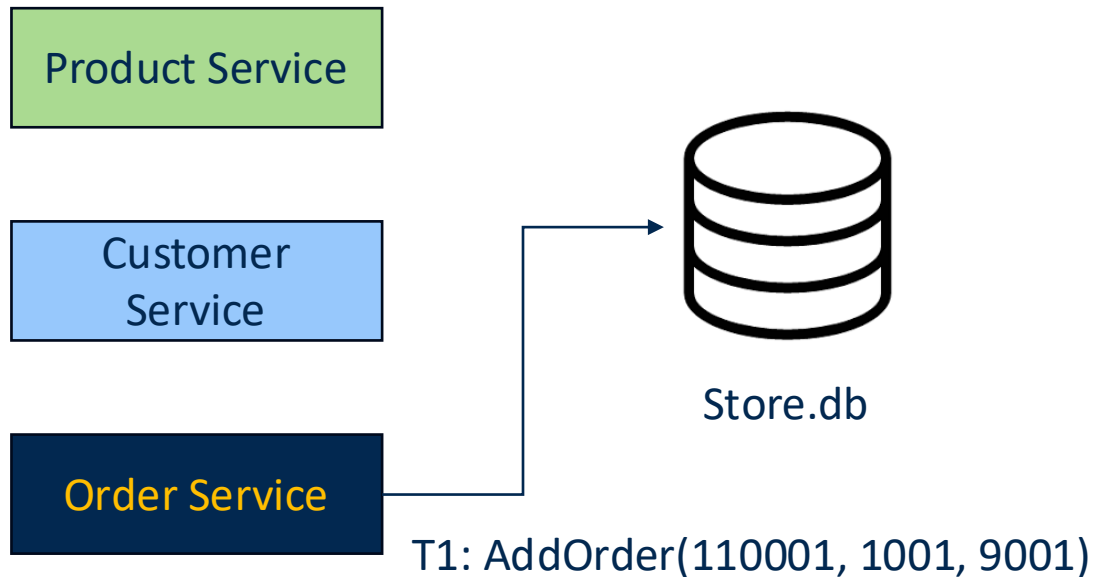
ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

OrderId	ProductId	CustomerId

Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



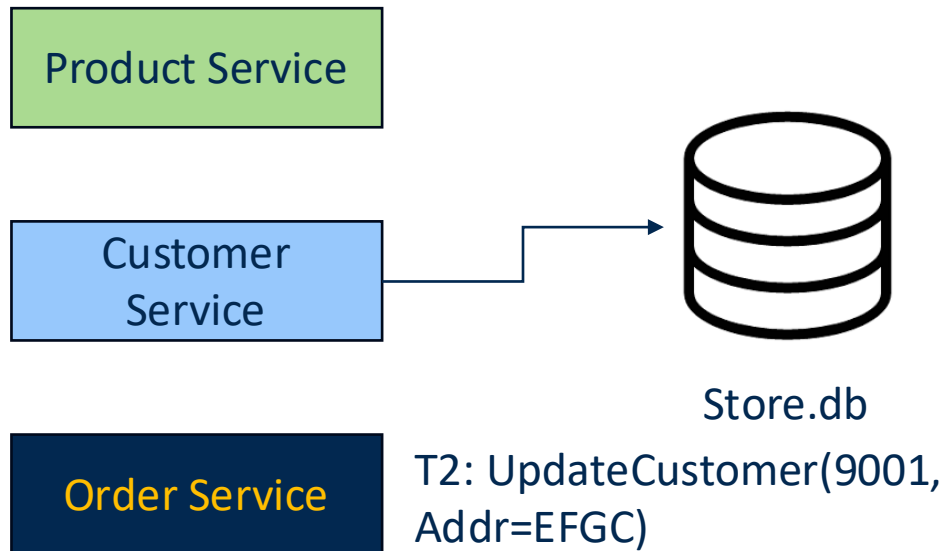
ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

OrderId	ProductId	CustomerId
110001	1001	9001

Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



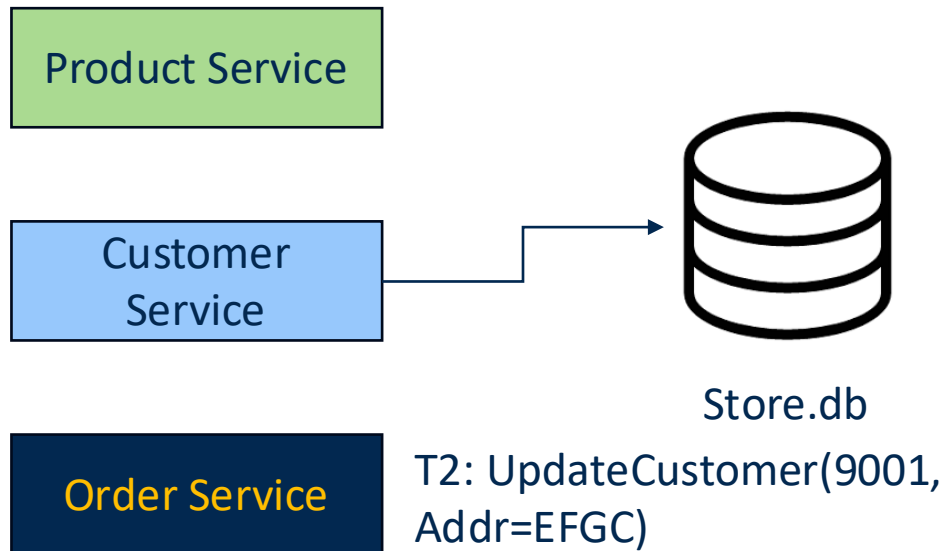
ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

OrderId	ProductId	CustomerId
110001	1001	9001

Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



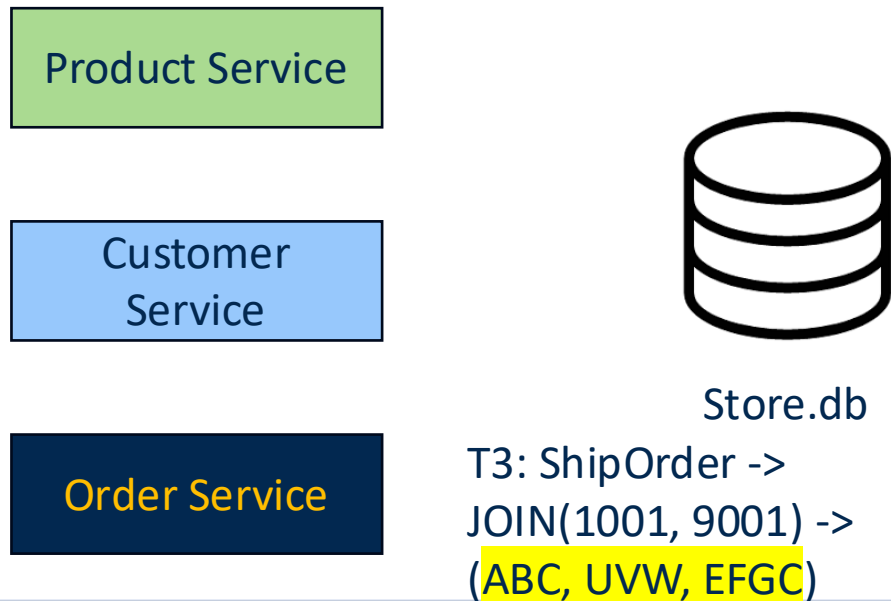
ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

OrderId	ProductId	CustomerId
110001	1001	9001

Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

OrderId	ProductId	CustomerId
110001	1001	9001

Microservice consistency guarantees

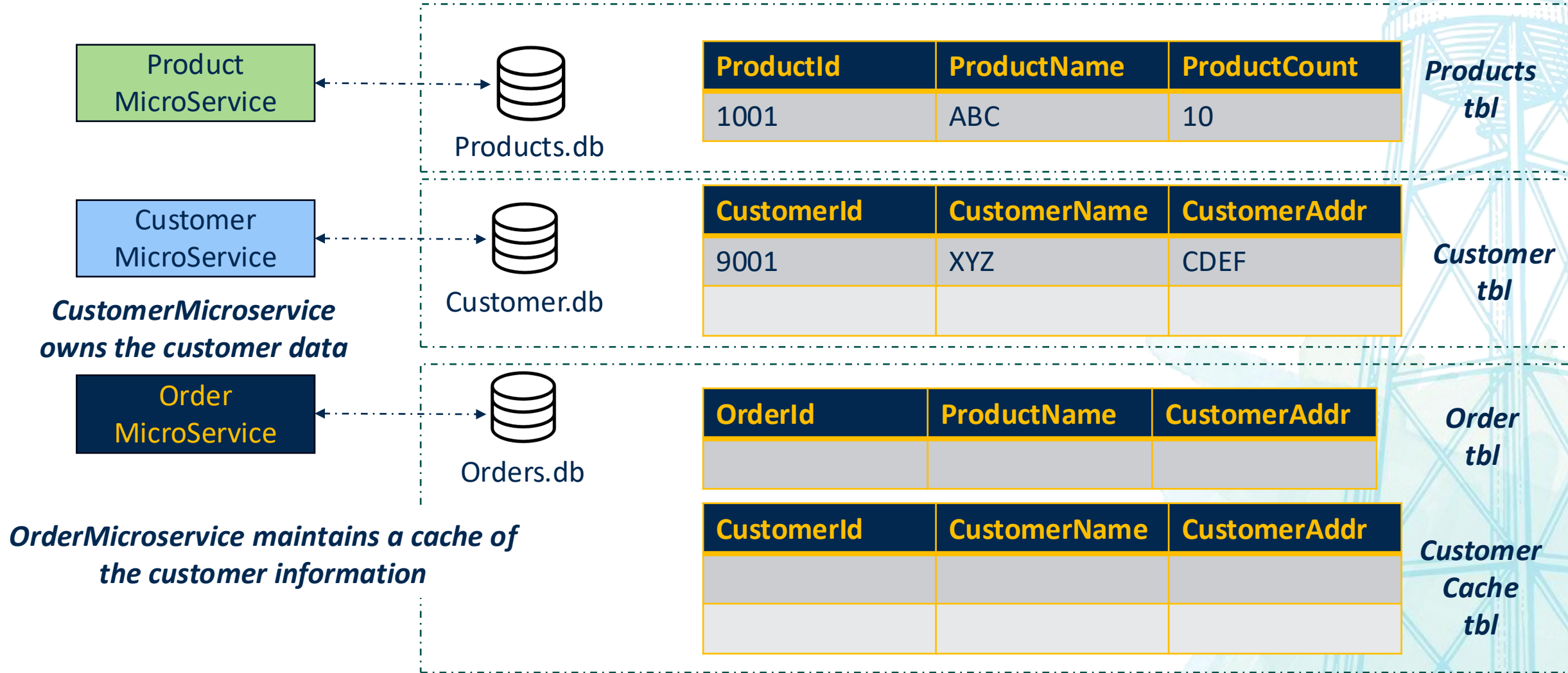
- Core idea: a microservice ***owns*** some data
 - ***Notifies*** dependent microservices of change
 - ***No guarantees*** on when the updates synchronize



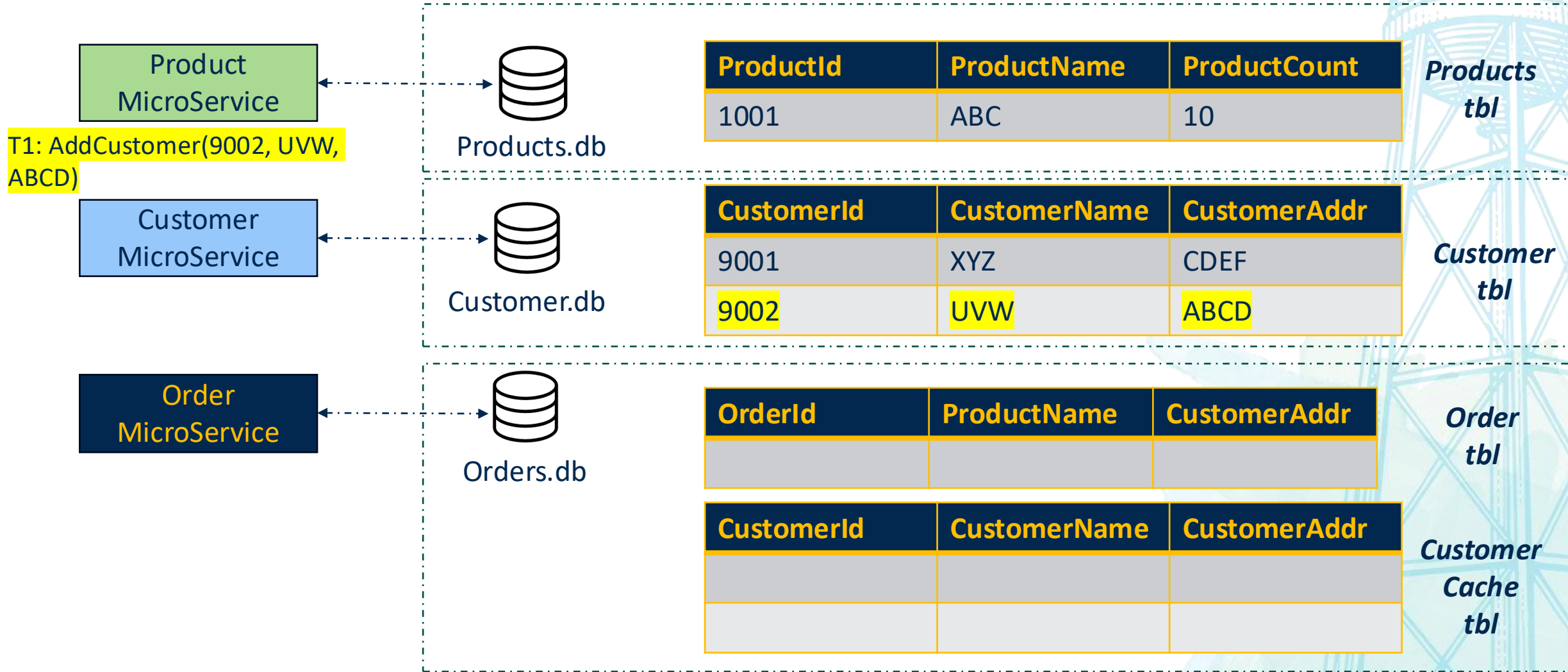
Microservice consistency guarantees

- Microservice-oriented apps are distributed
- Enforcing strong consistency guarantees in distributed systems is very hard
 - Network overhead, network partition, node failures
- Typically have **eventual consistency** guarantees – updates are **eventually** visible to all components
- System must work around these limitations

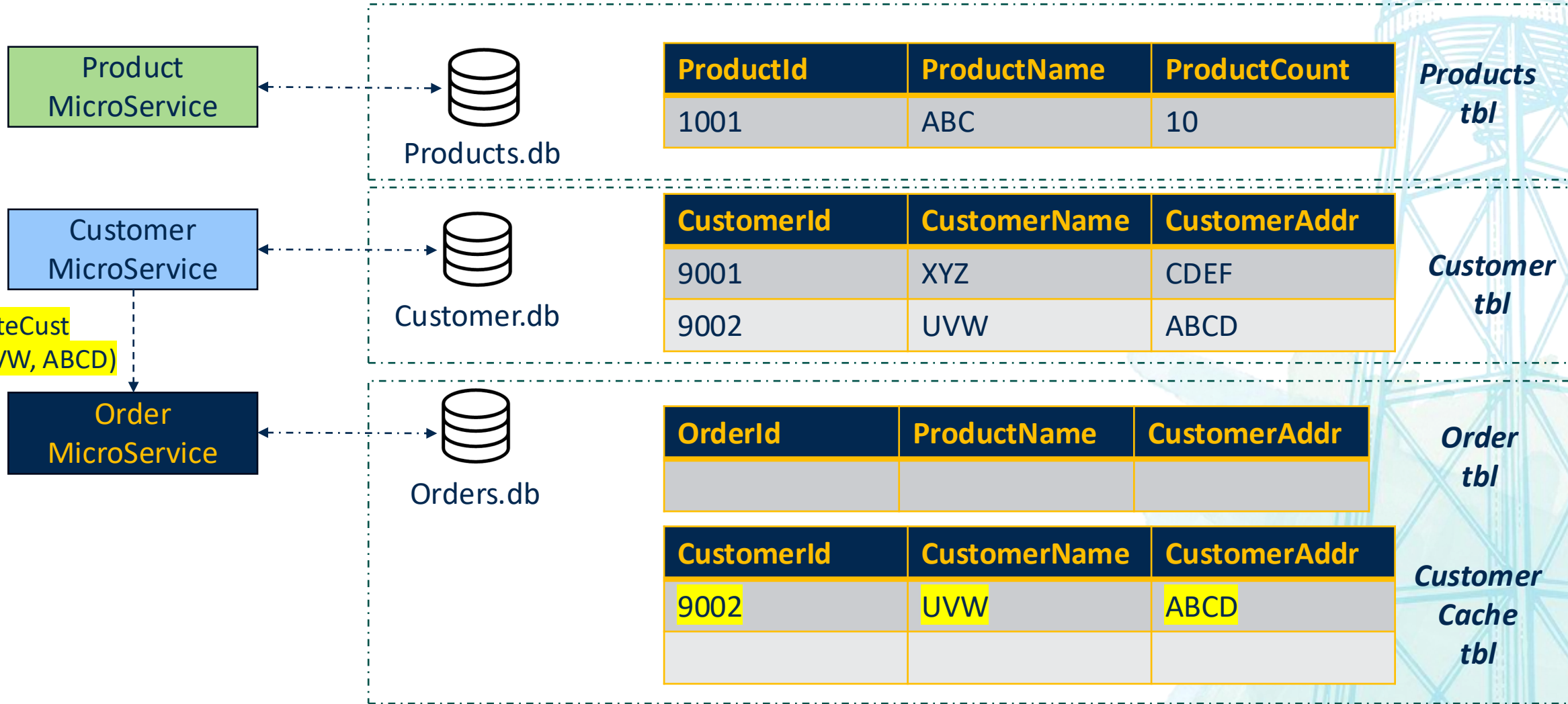
Working with eventual consistency



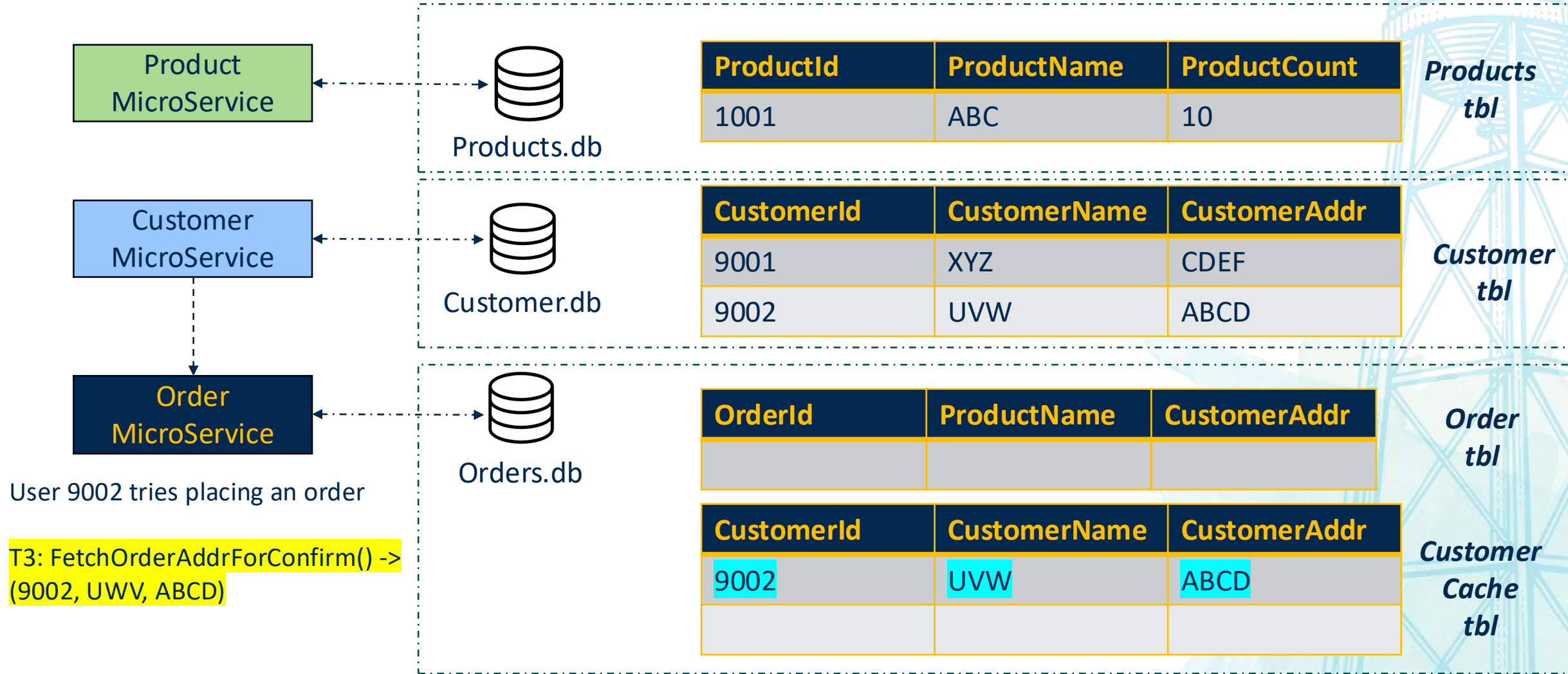
Working with eventual consistency



Working with eventual consistency



Working with eventual consistency



Working with eventual consistency

Name: UVW,
Address: ABCD

amazon prime

Secure checkout ▾

Delivering to Tapti Palit


Add delivery instructions

FREE pickup available nearby ▾

Paying with [REDACTED]

Use a gift card, voucher, or promo code

Arriving Feb 14, 2026 - Feb 20, 2026



Harney & Sons Loose Leaf Black Tea, Darjeeling 8 Ounce

\$18.00 (\$2.25 / ounce)

Ships from and sold by Amazon.com

1

+

Add gift options

Subscribe & Save:

☐ Save 5% today; Save up to 15% on future auto-deliveries ▾

Delivery every: 3 months (most common)

Saturday, Feb 14 - Friday, Feb 20

FREE

Change

Place your order

By placing your order, you agree to Amazon's [privacy notice](#) and [conditions of use](#).

Items:

Shipping & handling:

Estimated tax to be collected:

Order total:

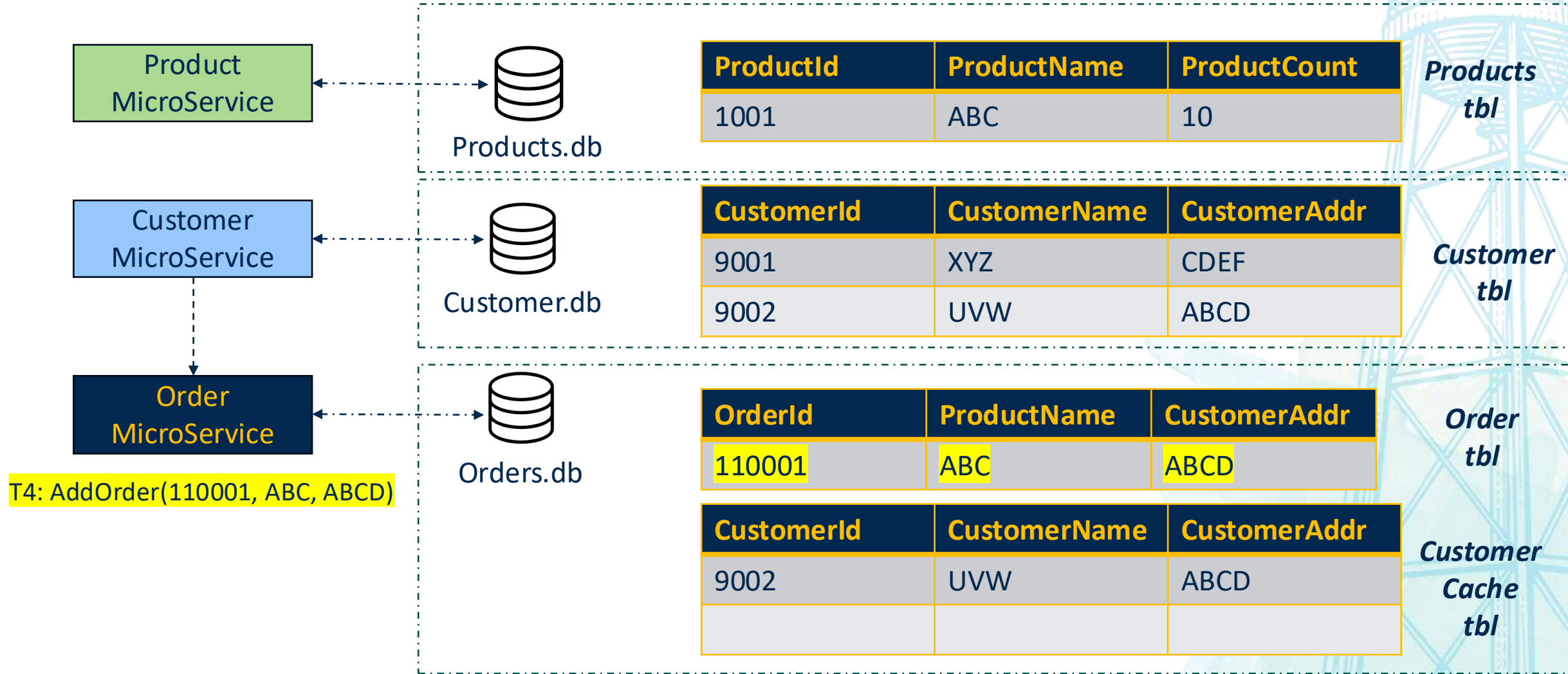
\$18.00

\$0.00

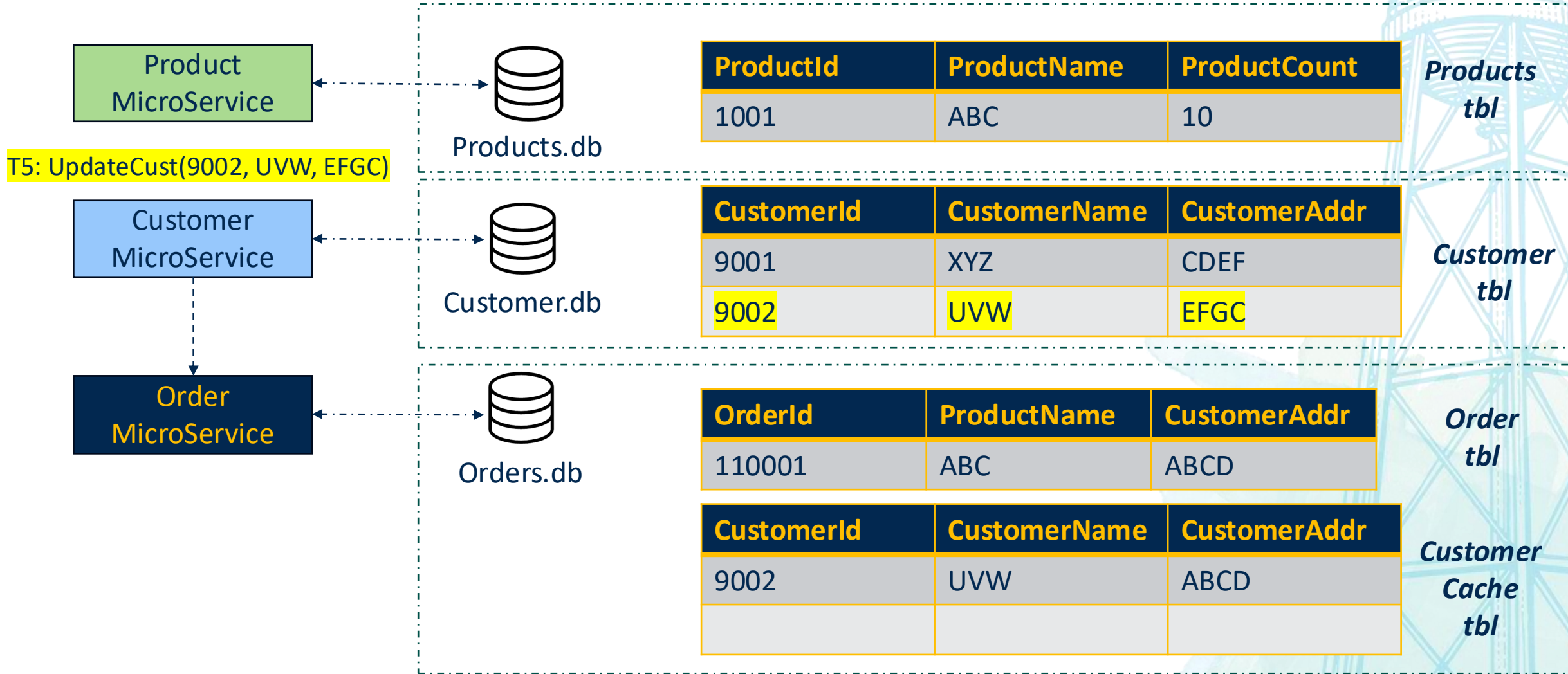
\$0.00

\$18.00

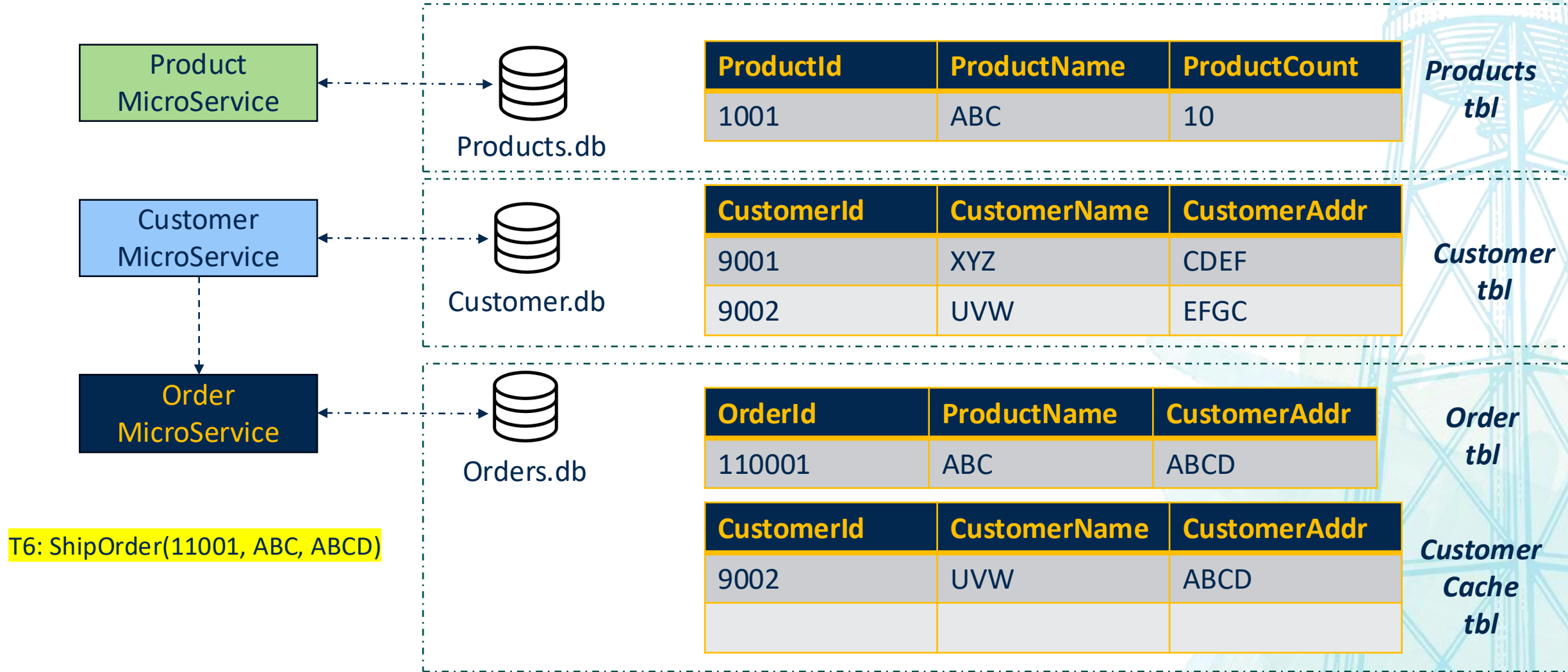
Working with eventual consistency



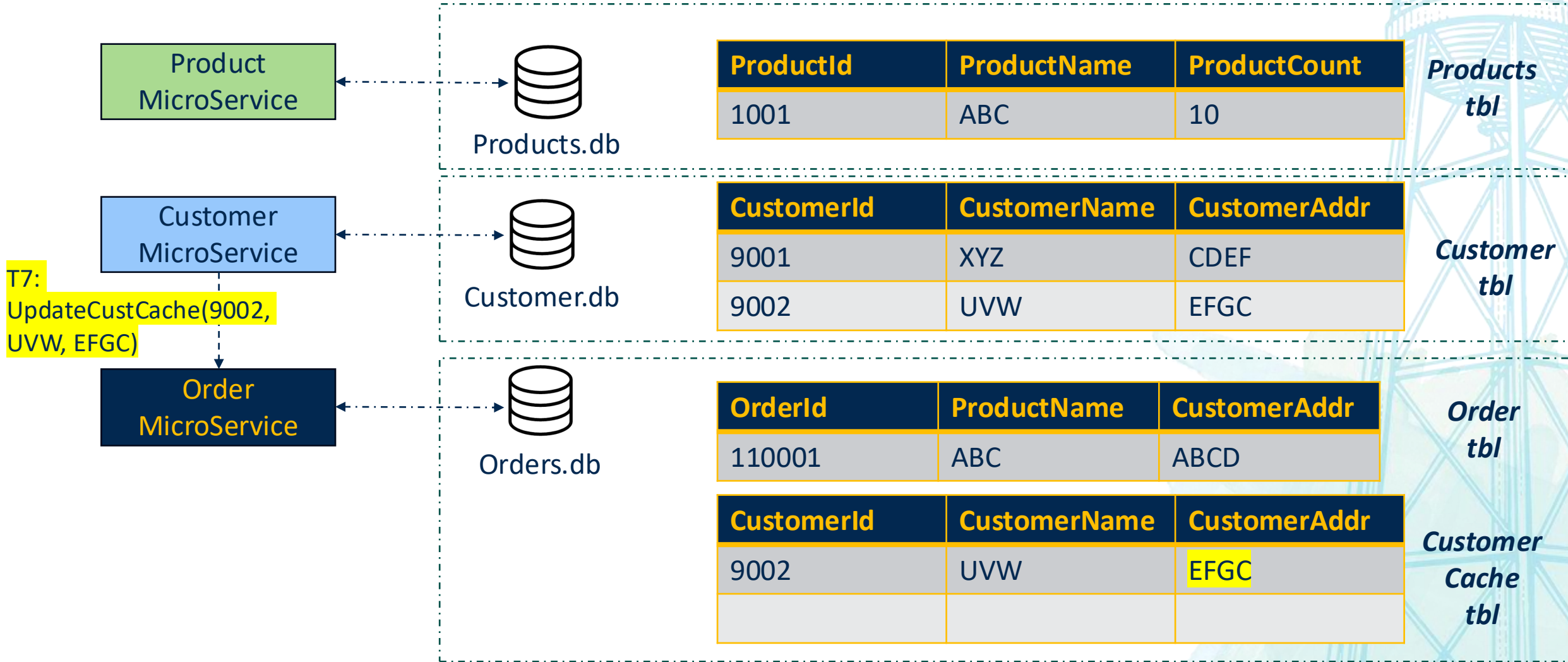
Working with eventual consistency



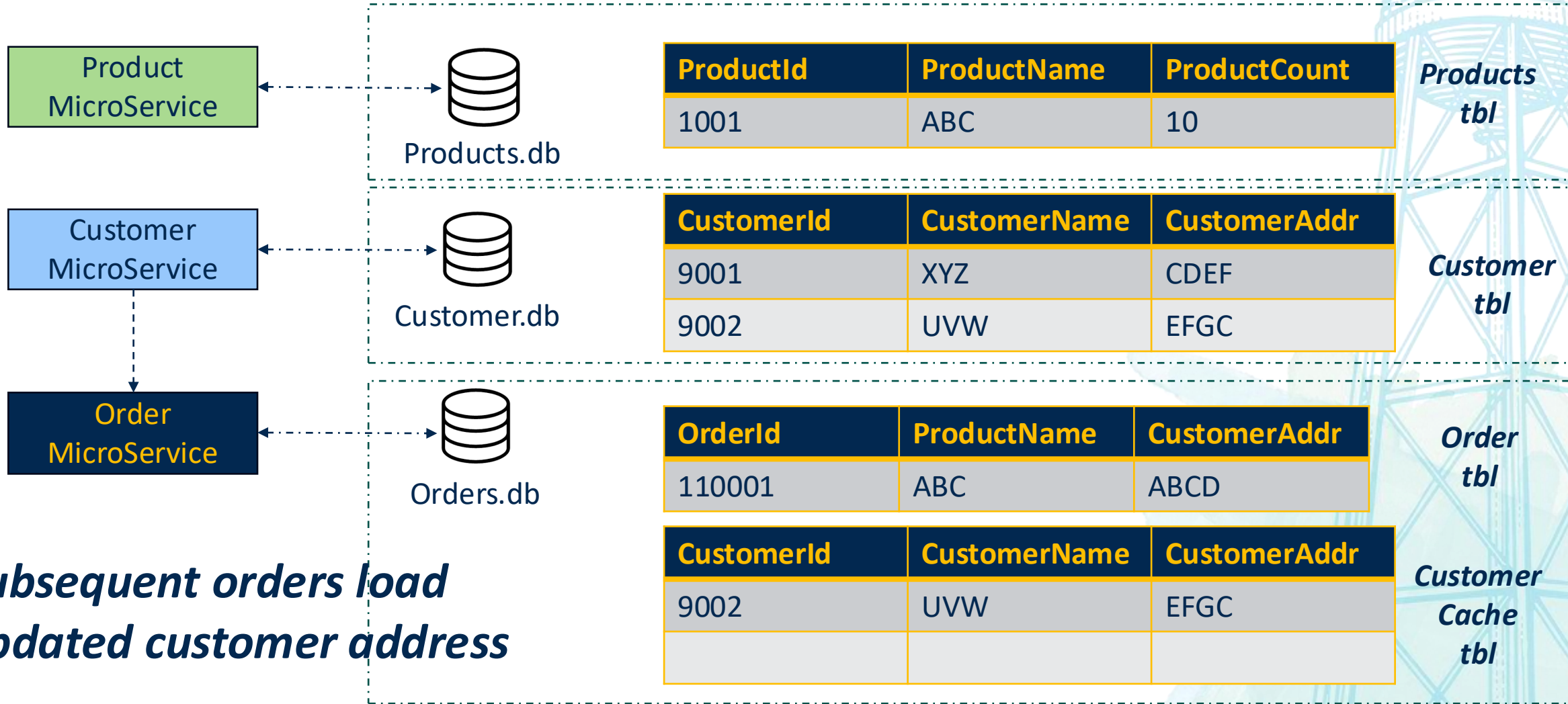
Working with eventual consistency



Working with eventual consistency



Working with eventual consistency

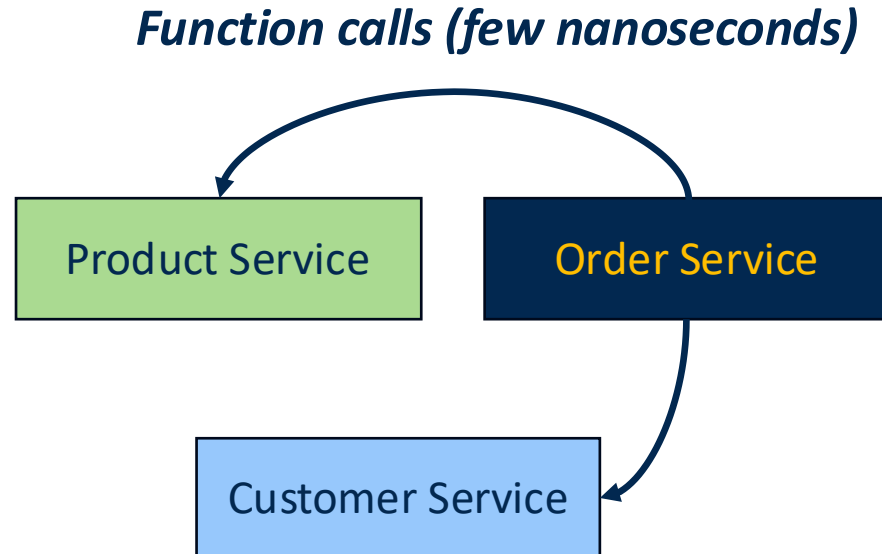


Microservice design concerns

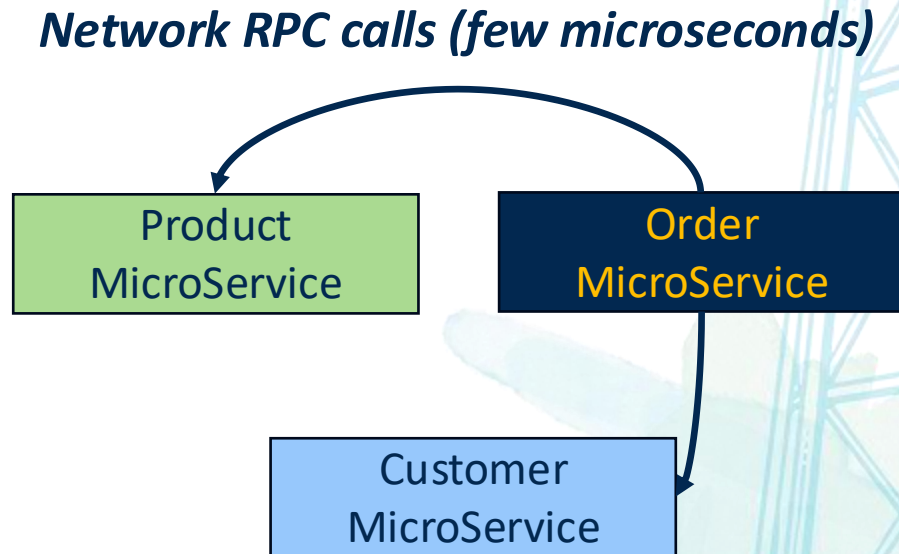
- Solution for eventual consistency is generally use-case dependent
- Some systems aim to provide strong consistency guarantees for distributed systems
 - Overhead often makes them impractical for microservices
- Reduce tight coupling
 - A microservice should not know the internal DB schema of another microservice
 - A microservice should not depend on another to be running to perform its task
- Limit network communication

Network communication cost

Monolithic applications



Microservices



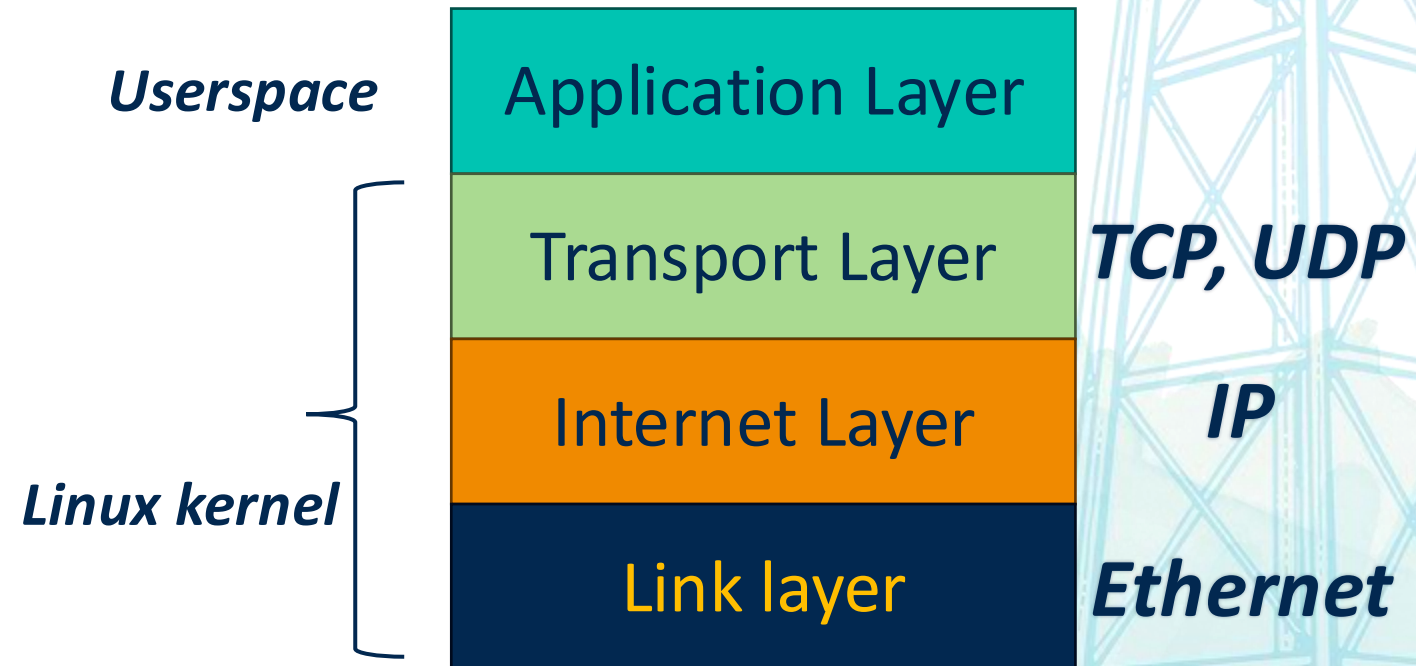
Network overhead decomposition

- Network latency
 - Datacenters use fast network connections
 - Infiniband has ~1-2 microsecond latency, 400+ Gbps bandwidth
 - Still not as fast as a local function call
- OS kernel overhead



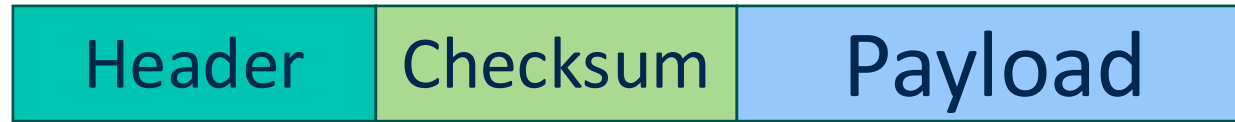
Kernel overhead for network comm.

- The OS kernel contains the networking code
- TCP-IP is the most common networking stack
- It is organized in layers
- Every layer has a ***protocol***

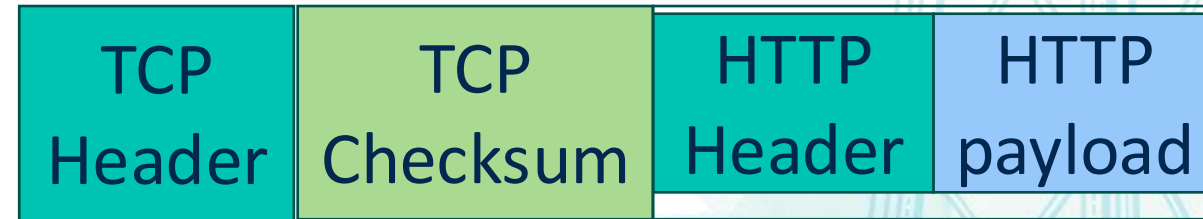
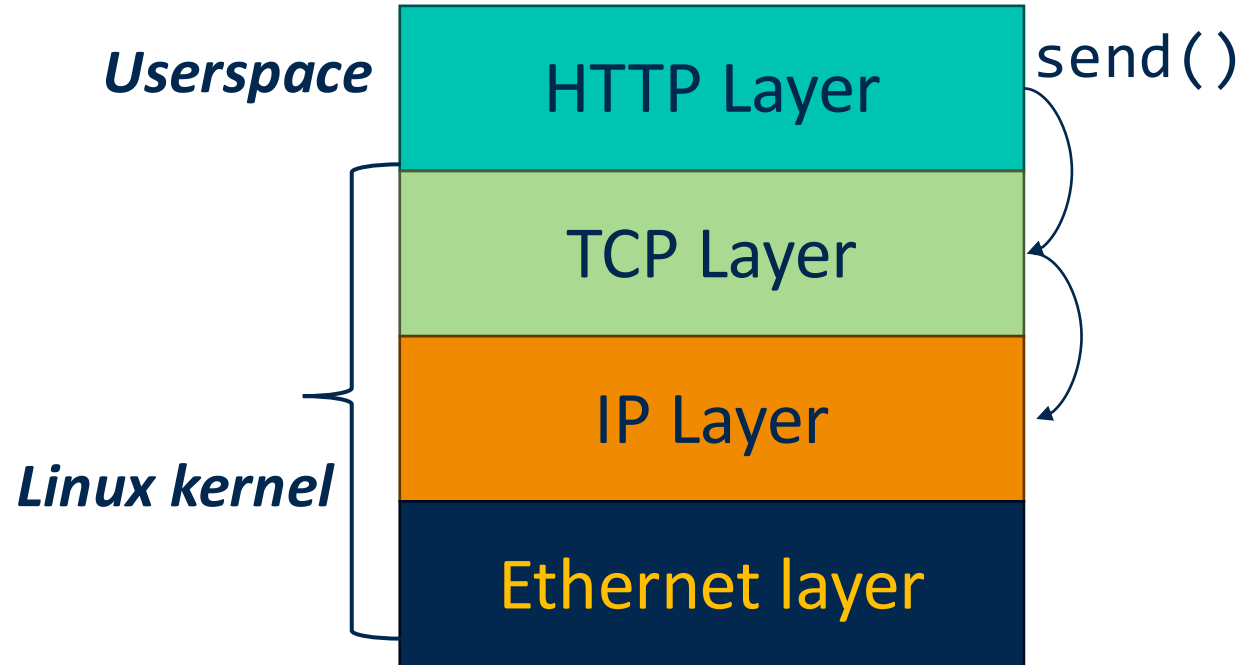


Each layer has a protocol

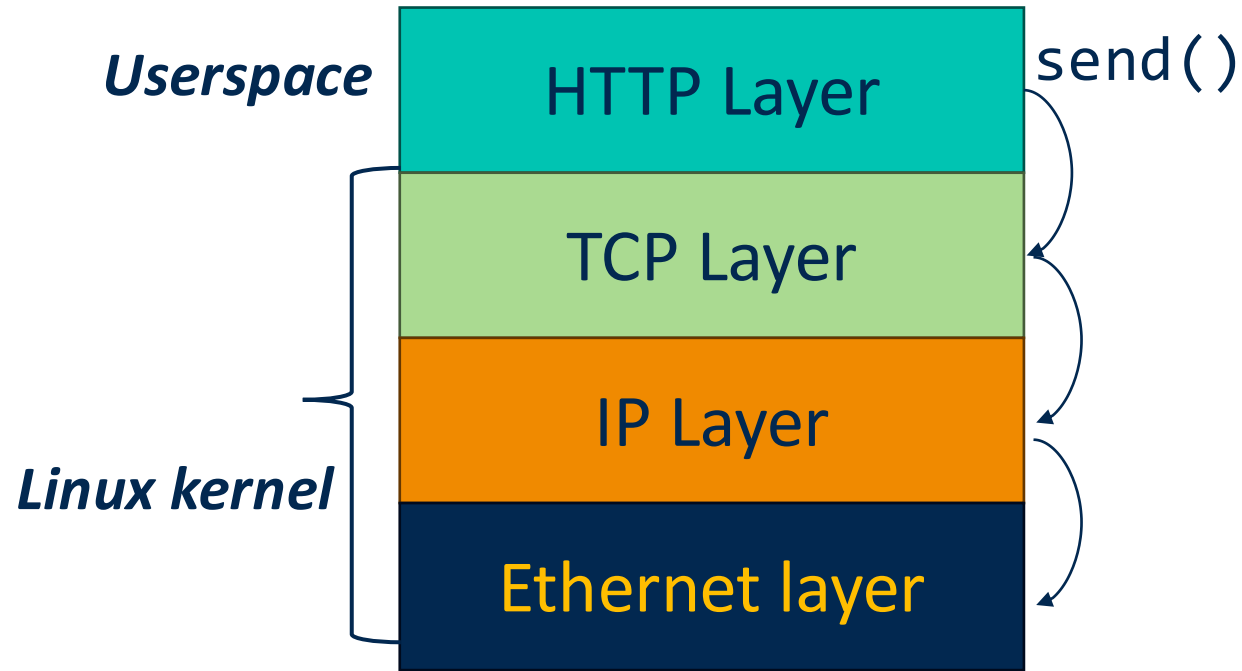
- Every layer/protocol has a fixed message format
 - Header
 - Payload
 - [optional] Checksum
- As the packet traverses through the layers, packets are rewrapped



Life of a packet

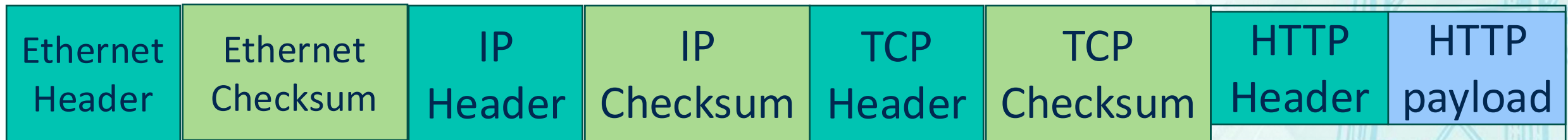


Life of a packet



Packet encapsulation process + kernel context switch can take 100+ microseconds

Reading 6 will discuss an alternative



Microservice pros

- Stronger decoupling and lower interdependence
- Improved scalability
- Easier deployment



Microservice cons

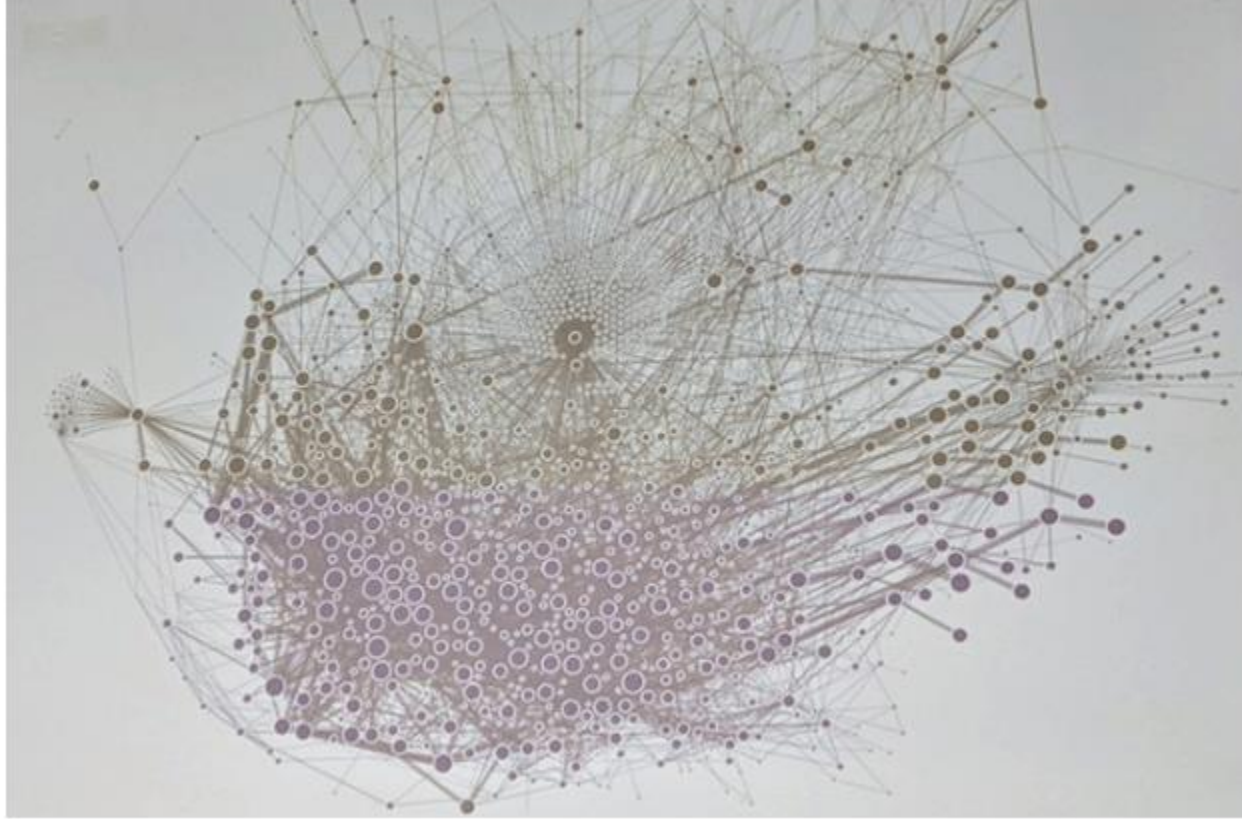
- Causes data denormalization
- Network overhead
- Higher complexity
- Debugging complex interactions is harder



Latency vs throughput

- Latency – time taken for one operation
 - Measured in seconds, milliseconds, microseconds, etc
 - Service Level Objectives/Agreements (SLO/SLA)
 - Example: 95% of all requests should be served in under 2 ms
- Throughput – number of operations performed in unit time (requests/sec)
- Microservices increase latency compared to monolithic operation, but improve throughput

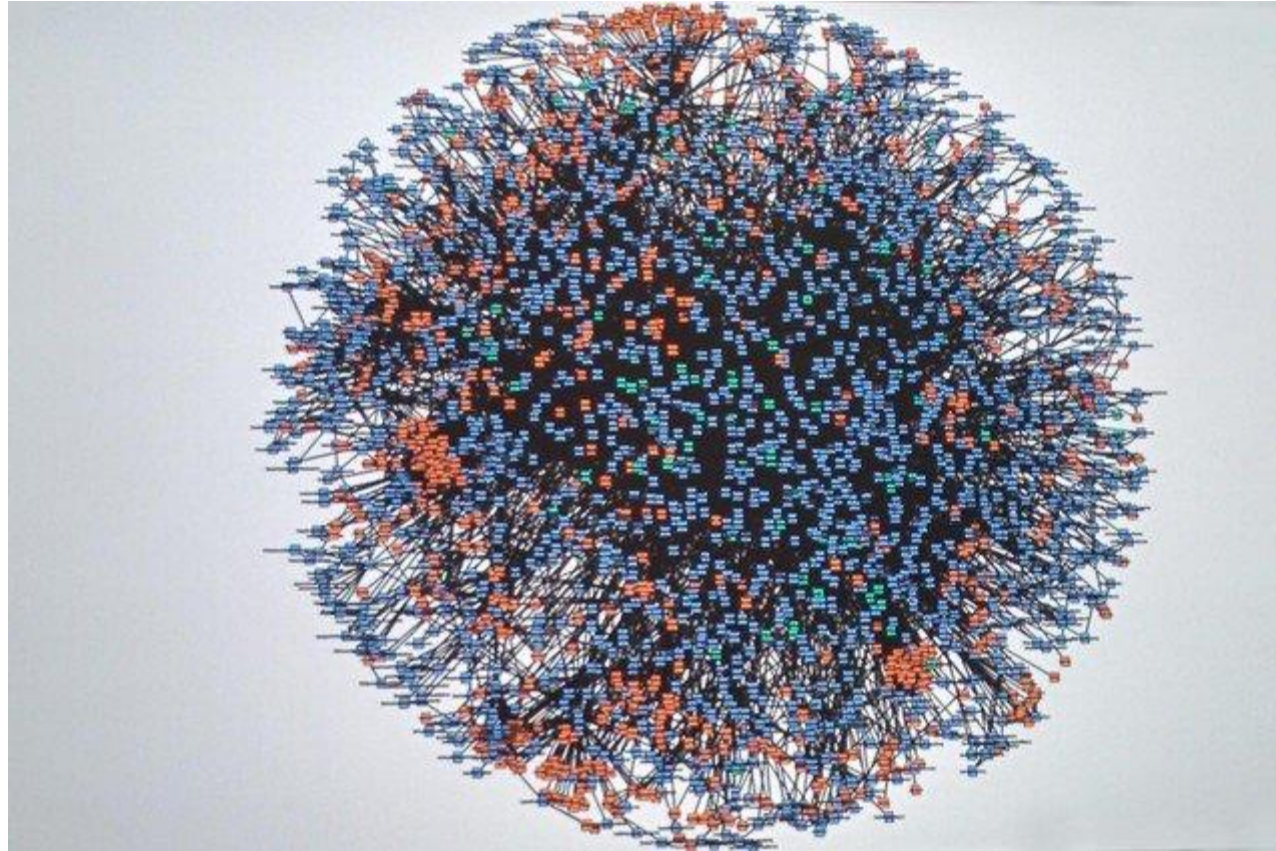
Microservices at Uber (2019)



<https://x.com/msuriar/status/1110244877424578560>

Microservices at Amazon (2008)

- Code-named “Deathstar”



<https://x.com/Werner/status/741673514567143424>

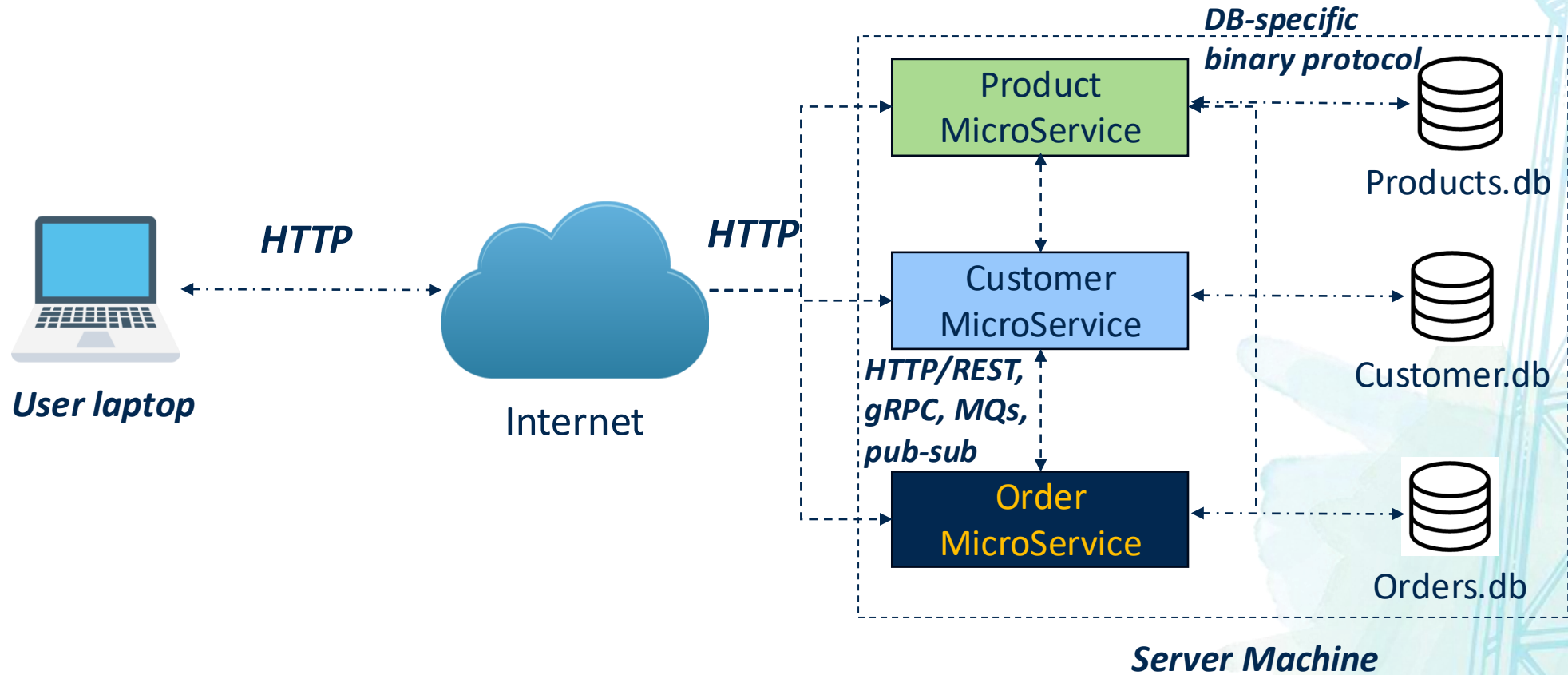
Data management and communication

- Once data and computation are split across services, two problems remain
 - How is data stored?
 - How services communicate?
- First, Spring Boot overview



Spring Boot with REST API overview

Microservices over HTTP



HTTP GET and POST request

- GET request - Used to retrieve data from the server

- GET `/index.html` HTTP/1.1

URL

- GET `/index.html?name=ECS160&data=02132025`

Request parameters

HTTP GET and POST request

- POST request - used to send data to the server

- POST **URL** `/users` HTTP/1.1

Content-Type: application/json{

"name": "John Doe",

"email": john.doe@example.com

}

Post "body"

Spring Boot Overview

- Framework for creating RESTful microservices
- Reduces boilerplate configuration code
- Embedded server (Tomcat/Jetty)
- Simplifies microservice creation through annotations
- Built-in support for REST APIs



RESTful microservices with Spring Boot

- Create classes that can act as REST endpoints
- Uses annotations to denote REST endpoint URLs
 - Allows complete decoupling from the boilerplate code
- Types of requests
 - @GetMapping, @PostMapping, @PutMapping, and so on... for all HTTP methods
- @PathVariable – extract variable from GET request
- @RequestBody – extract the post request body

```
class MyRequest {  
    private String postDate;  
    private String postContent;  
    // .. Getters and setters  
}
```

```
@RestController  
@RequestMapping("/myservice")
```

```
public class MyController {  
    @PostMapping("/sayhello")  
    public String sayHello(@RequestBody MyRequest  
request) {  
        return "";  
    }  
}
```

Effective URL: `http://[serverip]/myservice/sayhello`

Spring Boot Framework

- Uses reflection to first look up all classes with `@RestController` annotation
- Then automatically creates Servlets out of the methods annotated with `@GetMapping`, `@PostMapping`, etc.
- Uses reflection to parse the request parameters into class objects annotated with `@RequestBody`
- Generates the WAR file and launches the Apache Tomcat server
 - Simply execute `mvn spring-boot:run`

```
class MyRequest {  
    private String postDate;  
    private String postContent;  
    // .. Getters and setters  
}  
  
@RestController  
@RequestMapping("/myservice")  
public class MyController {  
    @PostMapping("/sayhello")  
    public String sayHello(@RequestBody MyRequest  
request) {  
        return "";  
    }  
}
```

Data management and communication

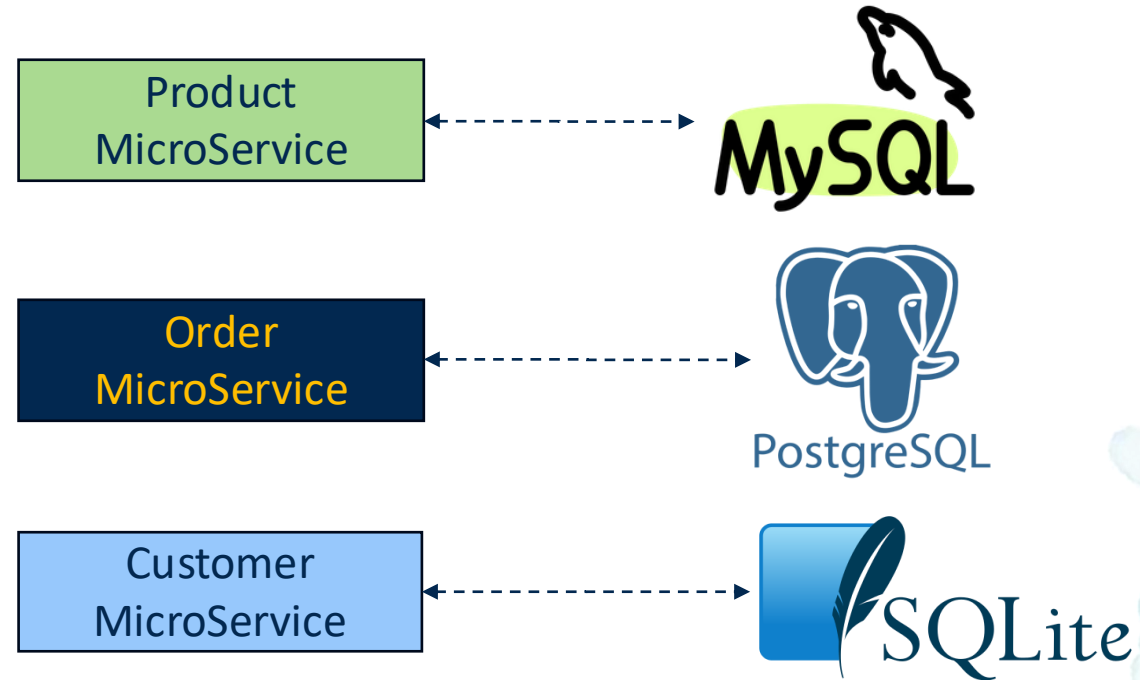
- Once data and computation are split across services, two problems remain
 - How is data stored?
 - How services communicate?



Databases

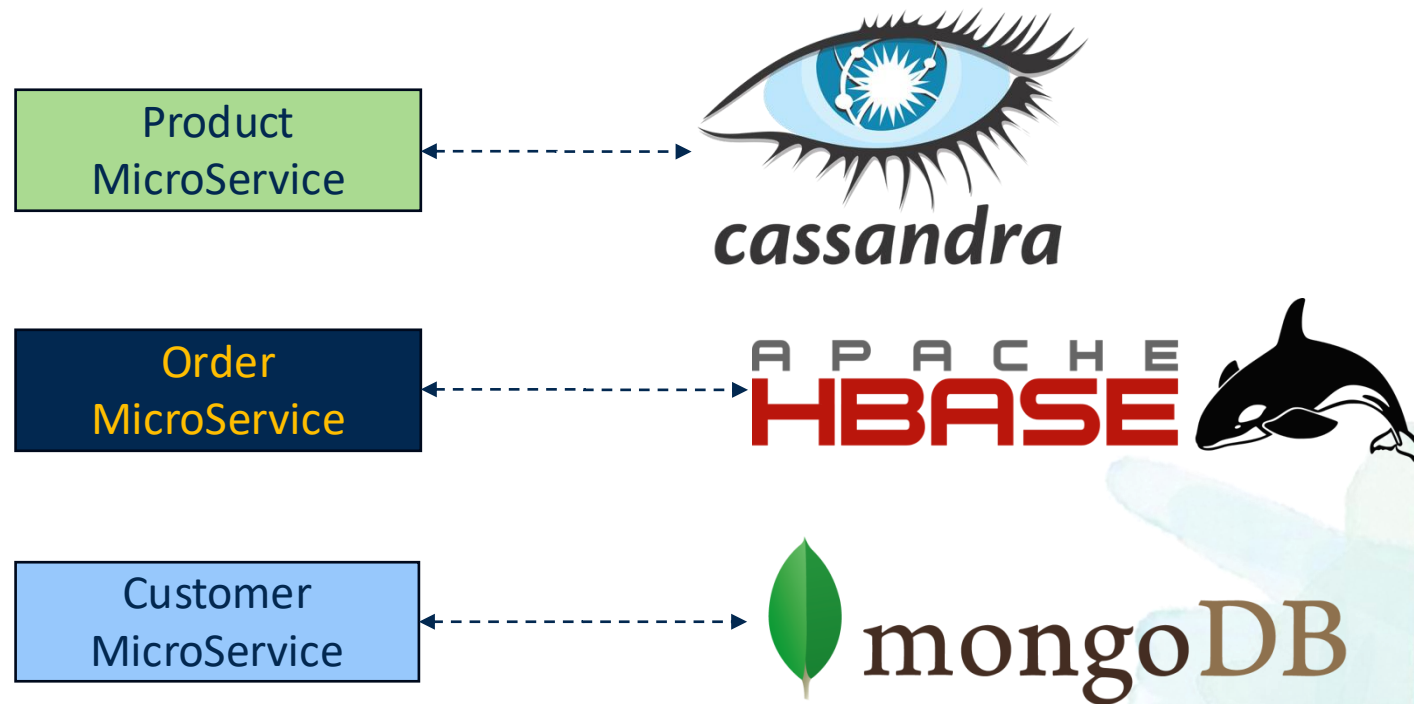


Microservices can have individual DBs



MySQL, PostgreSQL, SQLite are relational databases

Microservices can have individual DBs



Non-relational, NoSql databases

How to select which database to use?



Database choice factors

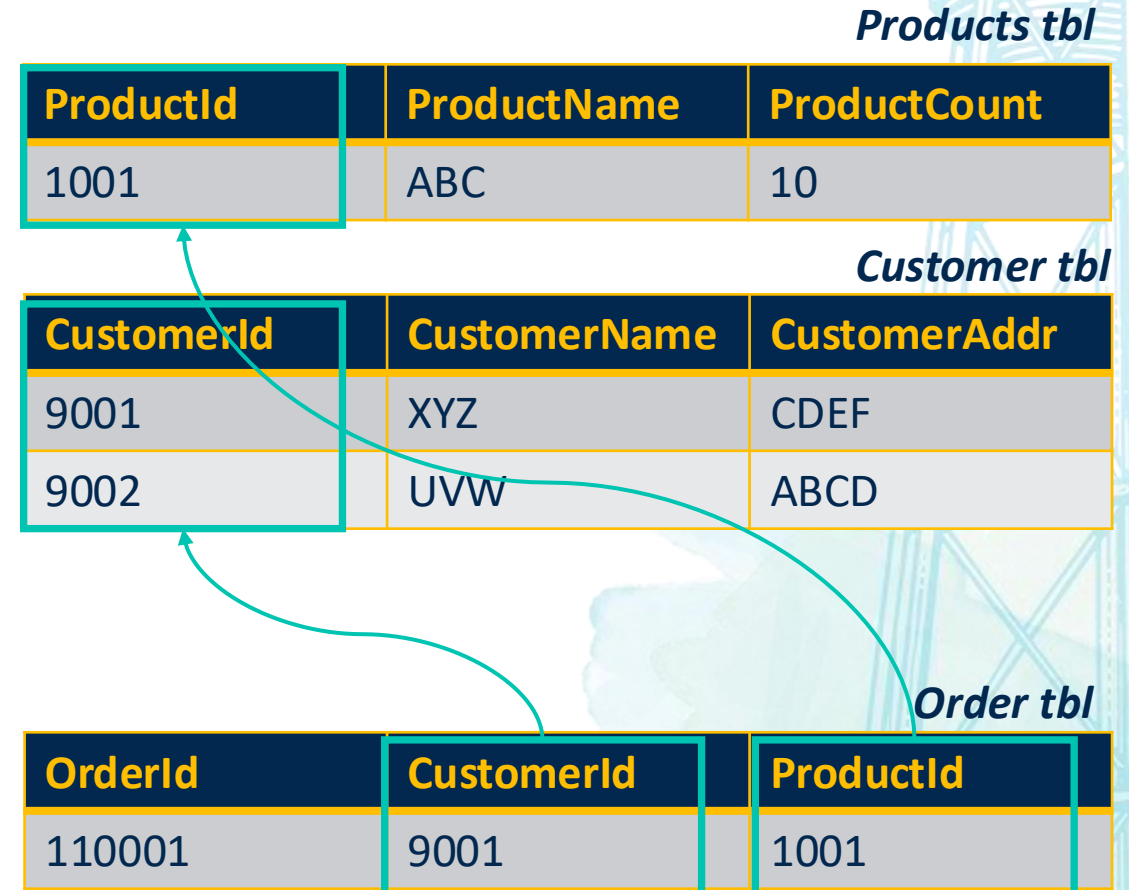
- Data model
 - How data is represented and which operations are natively supported
 - Examples - relational, document, graph, key-value
 - Shapes query semantics (joins, traversals, etc.)
- Storage engine
 - How data is physically stored and indexed
 - B-Trees, SSTables and LSM-Trees, Hash Indexes
 - Impacts performance tradeoffs (read vs. write performance, range scans, etc)

Data models

Relational databases

- Stores data as tables
- Supports relations using foreign keys and joins
- SELECT

```
o.OrderId,  
p.ProductName,  
c.CustomerName,  
c.CustomerAddr  
FROM Orders o  
JOIN Products p  
  ON o.ProductId = p.ProductId  
JOIN Customers c  
  ON o.CustomerId =  
c.CustomerId;
```



Document model

- Data is a document!
 - JSON, XML
- The database maintains this document
 - Provides a query language to inspect the contents
- Flexible schema
- Good for 1-to-1 or 1-to-many data relationships
- E.g. MongoDB



MongoDB BSON format

- String that represents an object
- Objects consists of key-value pairs
- Values can be primitives, arrays, or nested JSON objects
- Naturally supports hierarchical and semi-structured data

```
[  
  {  
    "id": "usr123",  
    "username": "coder_gal",  
    "details": {  
      "age": 28,  
      "city": "San Francisco"  
    },  
    "skills": [  
      "JavaScript",  
      "Python",  
      "JSON",  
      "APIs"  
    ]  
  }  
]
```

JSON object

Nested JSON object

JSON array

MongoDB BSON format

- Fields are accessed using dot notation
 - `user.id`
 - `user.details.age`
- Arrays are accessed using index notation
 - `user.skills[0]`

```
[  
  {  
    "id": "usr123",  
    "username": "coder_gal",  
    "details": {  
      "age": 28,  
      "city": "San Francisco"  
    },  
    "skills": [  
      "JavaScript",  
      "Python",  
      "JSON",  
      "APIs"  
    ]  
  }  
]
```

MongoDB BSON format

- Fields are accessed using dot notation
 - `user.id`
 - `user.details.age`
- Arrays are accessed using index notation
 - `user.skills[0]`

```
[  
  {  
    "id": "usr123",  
    "username": "coder_gal",  
    "details": {  
      "age": 28,  
      "city": "San Francisco"  
    },  
    "skills": [  
      "JavaScript",  
      "Python",  
      "JSON",  
      "APIs"  
    ]  
  }  
]
```

Graph-like data model

- Good for many-to-many relationships
- Neo4j



Key-value stores

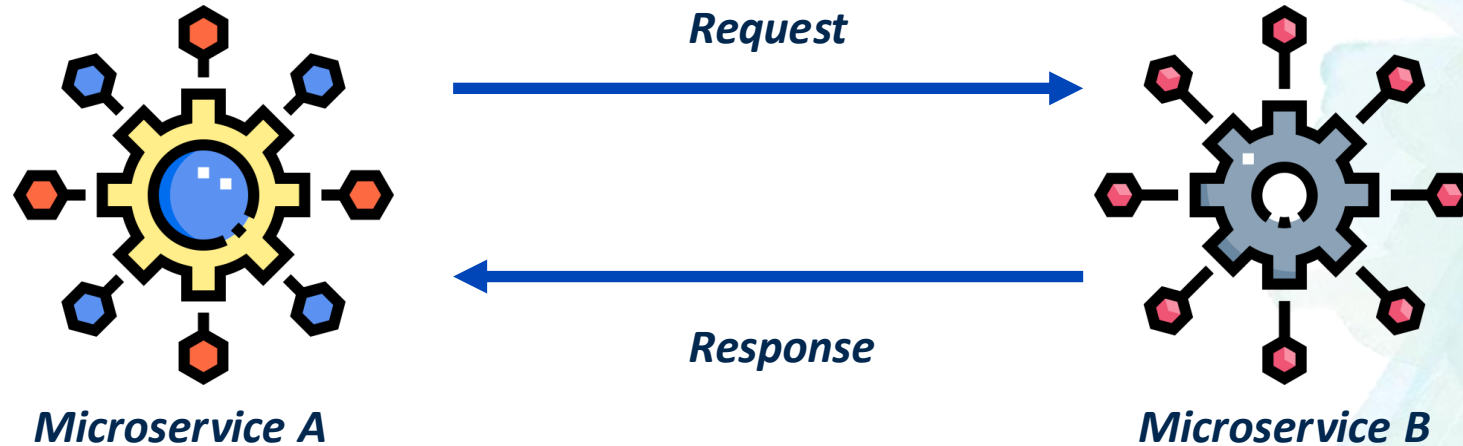
- Simple key-value pairs
- Redis, Memcached
- Rarely used to store data
- Cache expensive queries
- Communication
 - PUBLISH and SUBSCRIBE primitives



Software communication styles

Remote procedure call (RPC)

- Library/framework gives the illusion that the other microservice is running locally
 - Protocol that allows a program to execute a procedure on a remote server as if it were local
- Synchronous communication – caller waits for response before proceeding



RPC key features

- Language agnostic: the RPC itself does not depend on the service language
- Abstracts network details
 - Typically, over HTTP

RPC formats

- Text-based (e.g. REST APIs)
 - Uses JSON or XML for data exchange
 - Human-readable, but larger payloads
- Binary formats (e.g. gRPC)
 - Uses binary standards (such as protobuf) for serialization
 - Compact but faster, but less human-readable
 - ... *why?*



JSON and REST APIs

- Representational State Transfer (REST) is an architectural style for web services
- Application/microservice exposes an URL
- Uses HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources
- Commonly paired with JSON for data exchange

// GET Request to Fetch a User denoted by ID

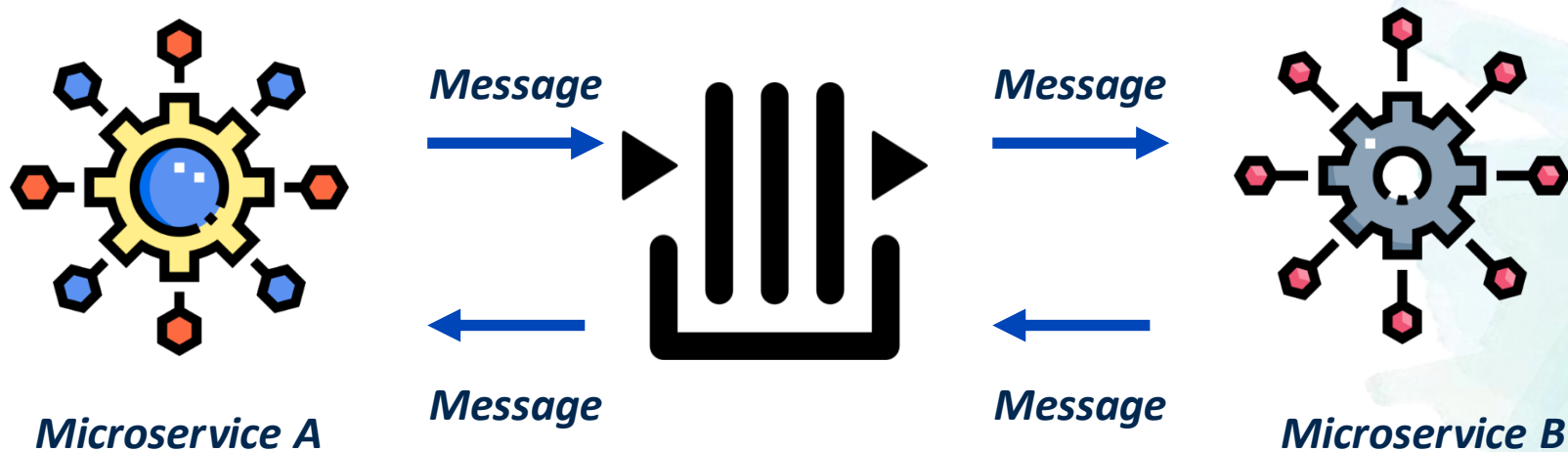
> GET /users/123

// Response

```
{  
  "id": 123,  
  "name": "John Doe",  
  "email": john.doe@example.com  
}
```

Message queuing (MQ)

- Asynchronous communication model
 - Messages sent to a queue and processed by consumers independently of the producer
- Stronger decoupling



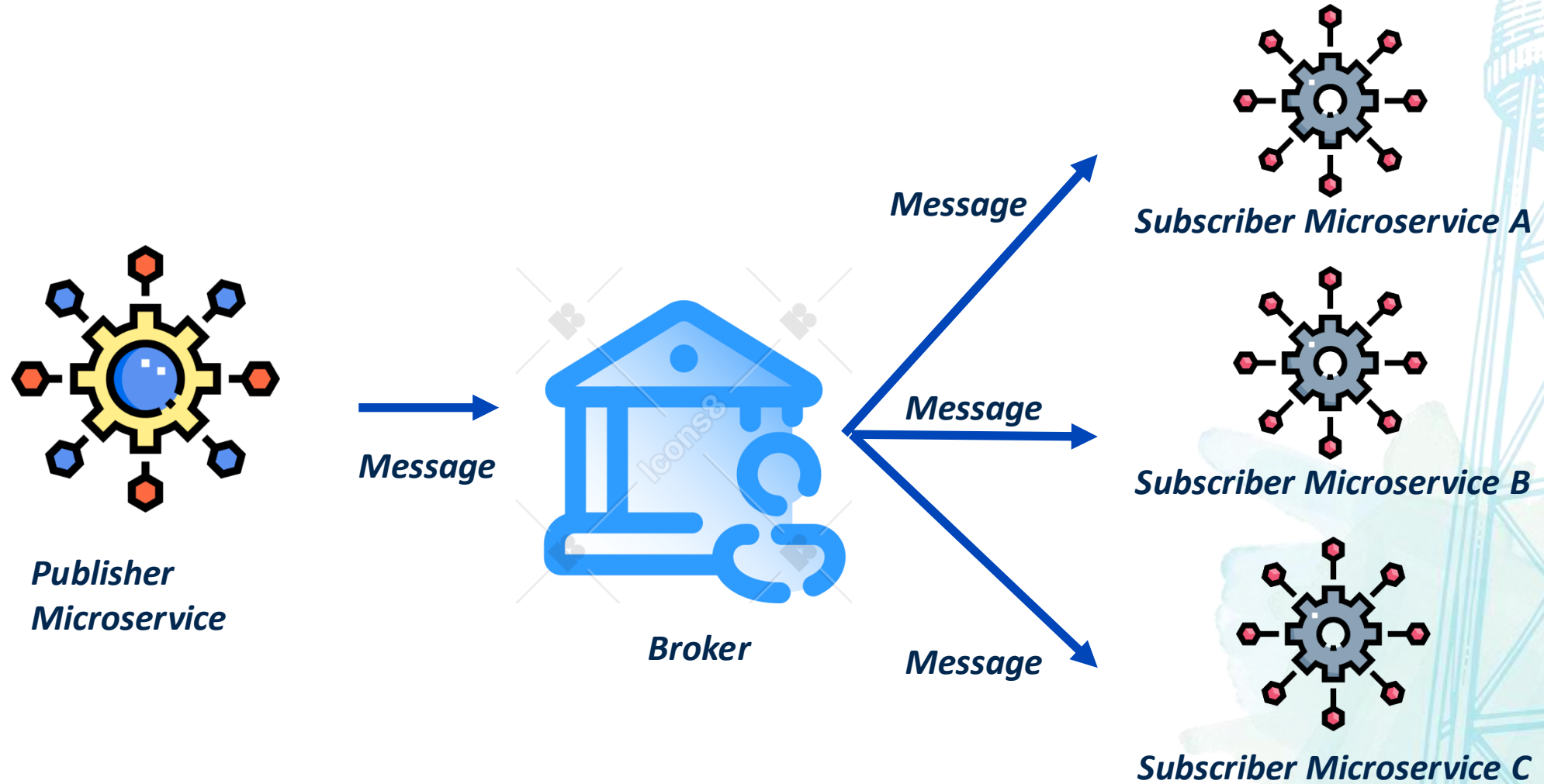
Open source and paid MQ services



Pub-sub architecture

- Asynchronous messaging pattern where **publishers** send messages to a central **message broker** or **topic**, and **subscribers** receive messages based on their subscriptions
- Broadcasting: messages can be sent to multiple subscribers
- Typically, messages are persistent at the broker and must be explicitly deleted
- Same frameworks often can act as both MQ or Pub-Sub depending on configuration

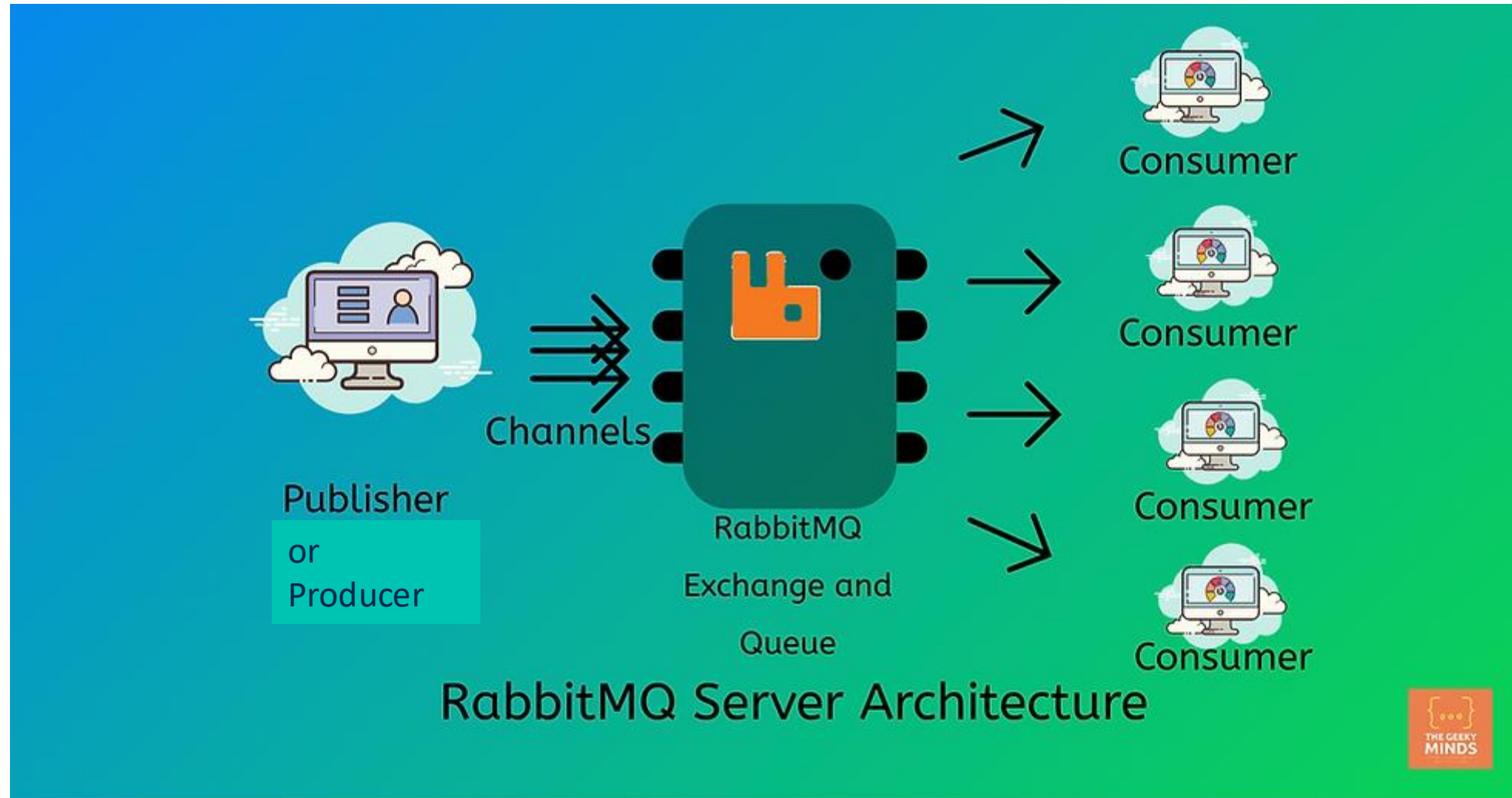
Pub-sub architecture



Pub-sub frameworks



RabbitMQ architecture



Asynchronous communication with RabbitMQ

```
public class RabbitMQProducer {
    private static final String QUEUE_NAME = "hello_queue";

    public static void main(String[] args) {
        // Create a connection factory
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost"); // RabbitMQ server
        address

        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {

            // Declare a queue (it must exist before publishing)
            channel.queueDeclare(QUEUE_NAME, false, false,
            false, null);

            String message = "Hello, RabbitMQ!";
            channel.basicPublish("", QUEUE_NAME, null,
            message.getBytes());

            System.out.println(" [x] Sent: '" + message + "'");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
public class RabbitMQConsumer {
    private static final String QUEUE_NAME = "hello_queue";

    public static void main(String[] args) {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");

        try {
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            // Declare the queue in case it doesn't exist
            channel.queueDeclare(QUEUE_NAME, false, false, false,
            null);

            System.out.println(" [*] Waiting for messages...");

            // Create a consumer to receive messages
            DeliverCallback deliverCallback = (consumerTag, delivery) -
            > {
                String message = new String(delivery.getBody(), "UTF-
            8");
                System.out.println(" [x] Received: '" + message + "'");

            };

            channel.basicConsume(QUEUE_NAME, true, deliverCallback,
            consumerTag -> { });

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```