Modern software architectures

Tapti Palit

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
  - Synchronous, Asynchronous (MQs), Publish-subscribe models

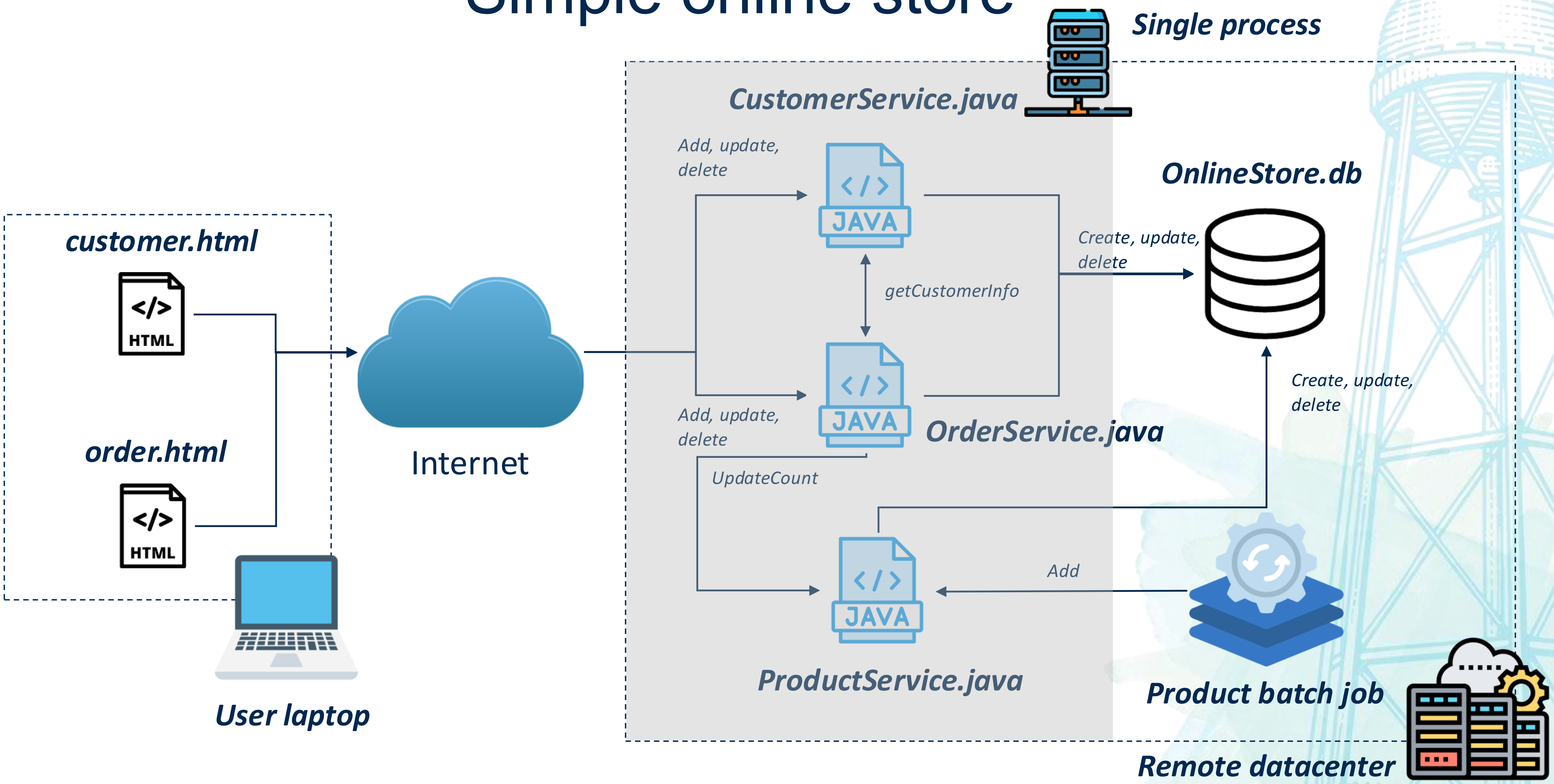
All about the data!

- Modern software architecture is data-intensive
- Most of the time we'll be concerned about
  - Who owns data?
  - How to optimally store data?
  - How is data shared?
  - How to limit overhead due to data-sharing?

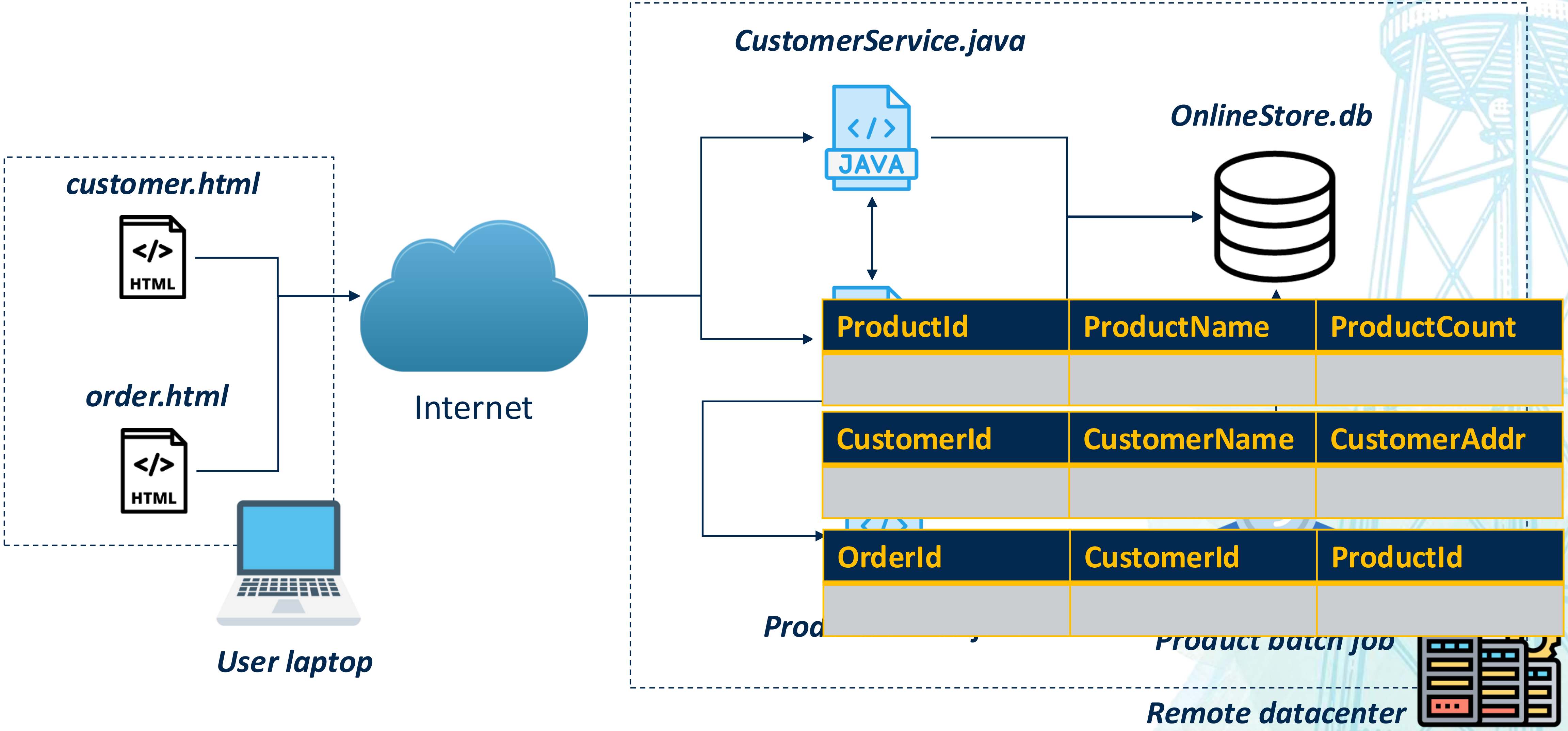
Monolithic software architecture

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

Simple online store



Simple online store



ProductId	ProductName	ProductCount

CustomerId	CustomerName	CustomerAddr

OrderId	CustomerId	ProductId

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
- A table can contain one or more foreign keys

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	CustomerId	ProductId
110001	9001	1001

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

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	CustomerId	ProductId
110001	9001	1001

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
110001	9001	1001

Need for joins

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

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
- Foreign keys enforce referential integrity
- 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

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

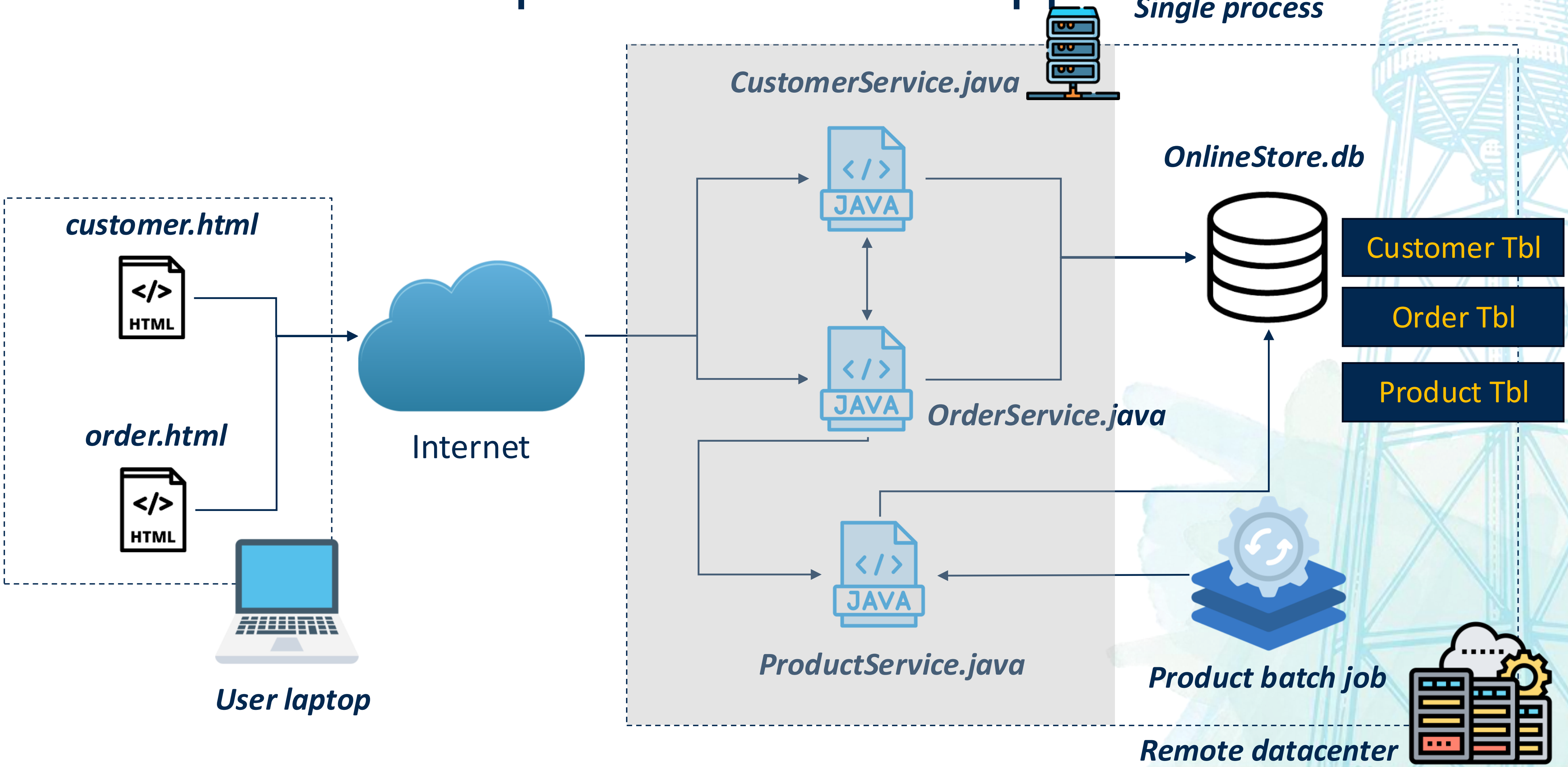
OrderId	ProductId	CustomerId
110001	1001	9001
110002	1001	9001
110003	1001	9002

Order tbl

SQL != relational databases

- Note: not quite specific to relational databases
- Apache Cassandra, Apache HBase, Apache Kafka + ksqlDB

Simple online store app

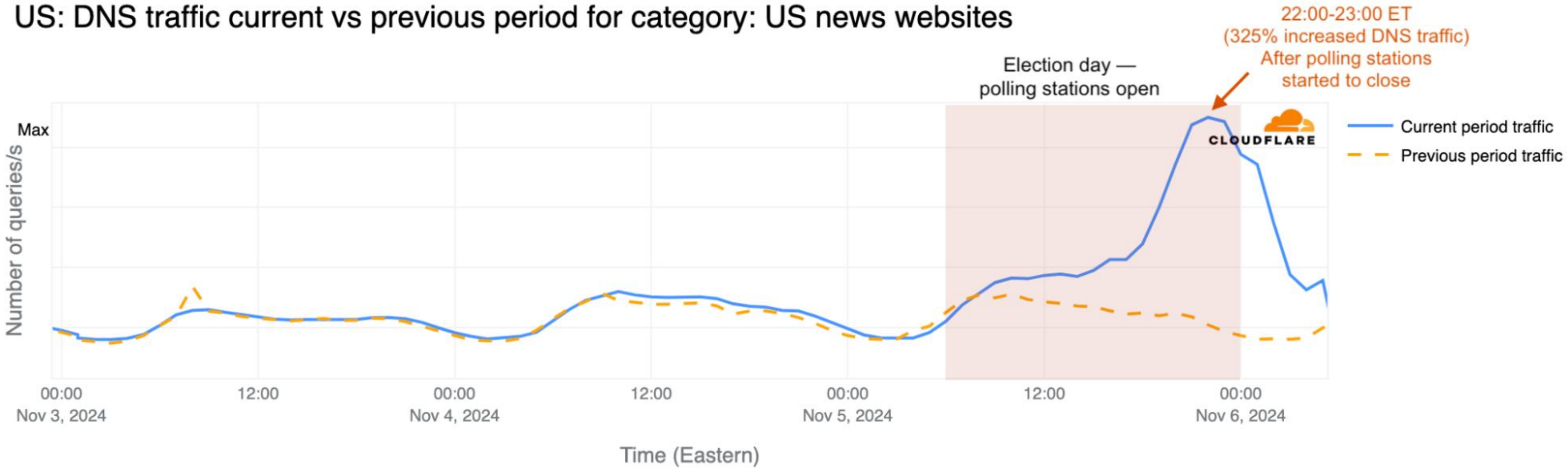


UCDAVIS

Scalability

- Website traffic is not constant
- 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/

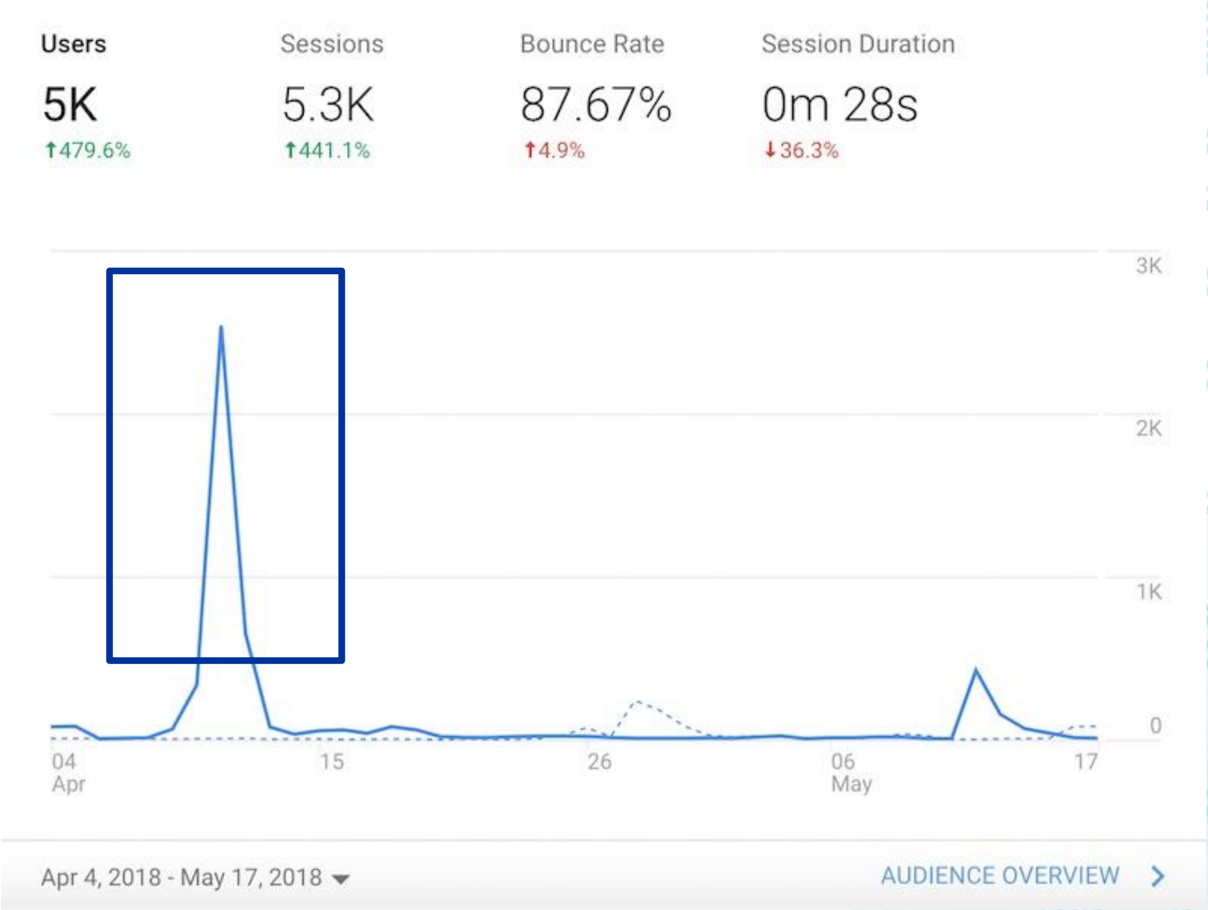


UC DAVIS

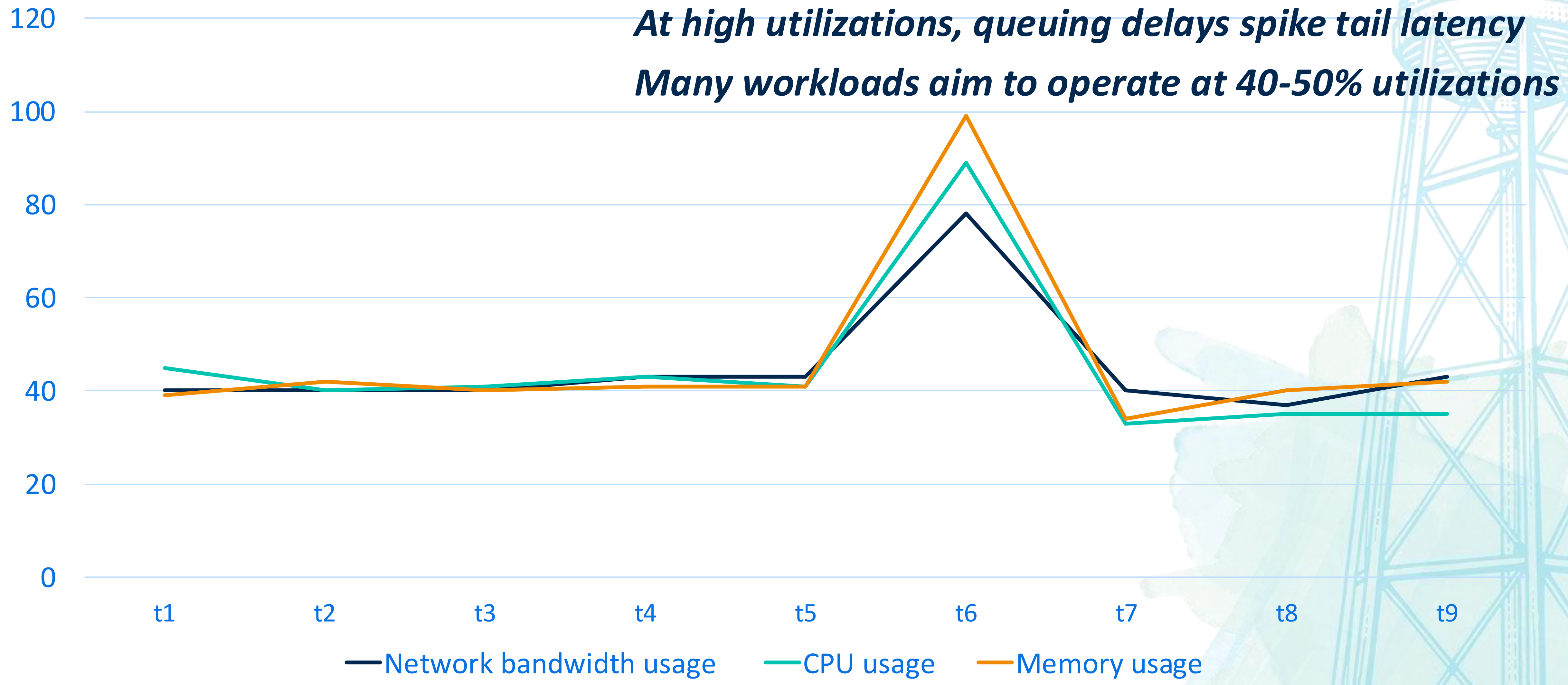
Scalability

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

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

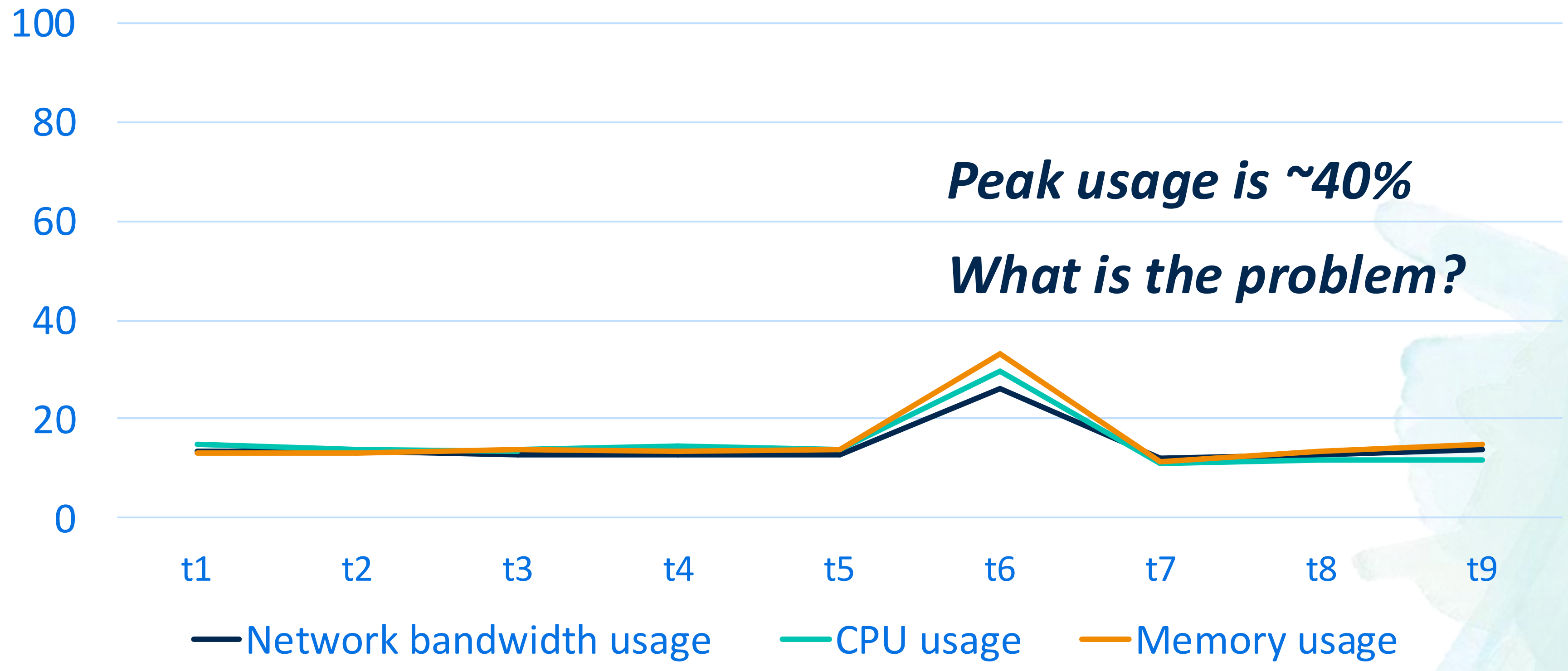


Traffic spike to online store



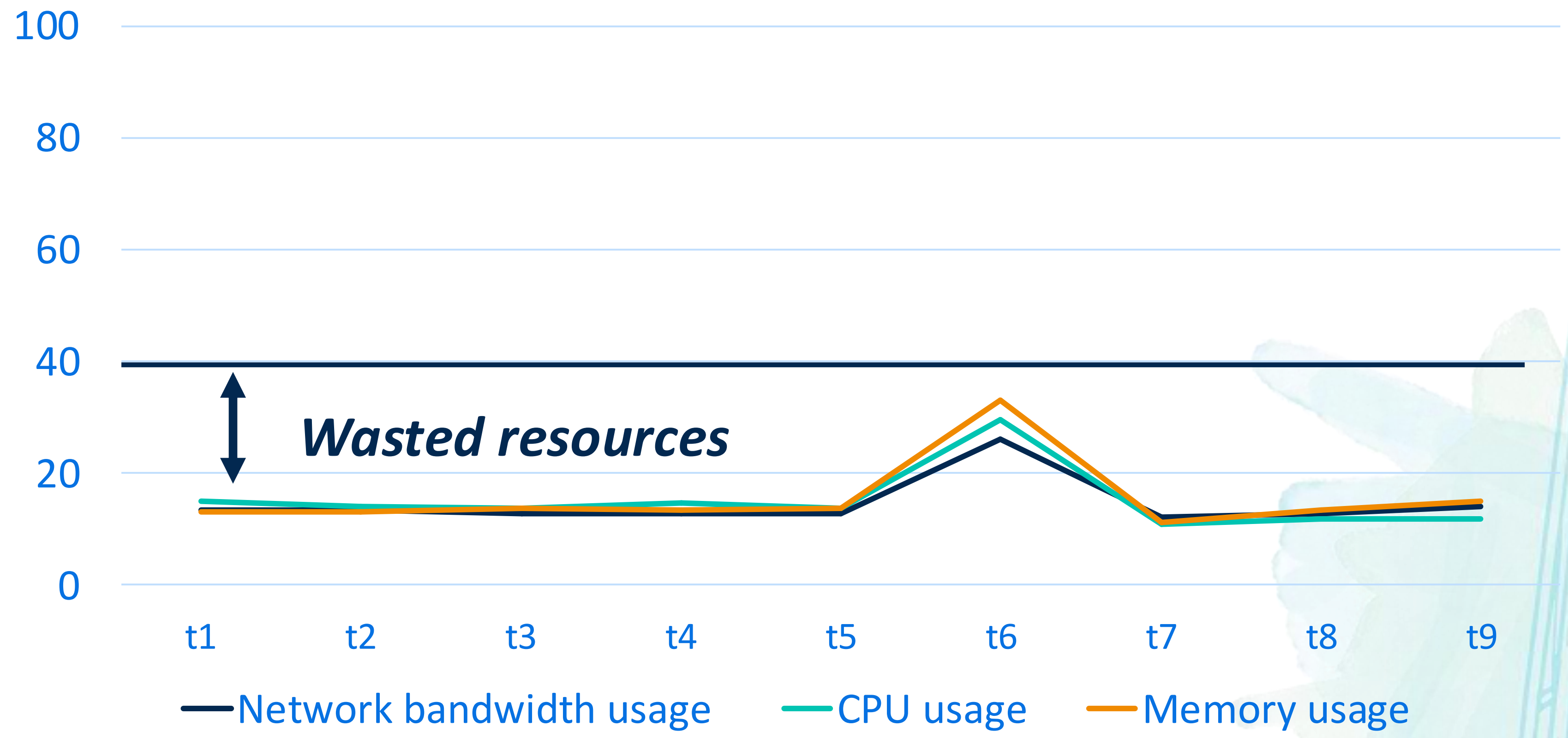
Vertical scaling

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



Vertical scaling limitations

Resource wastage (assuming we aim for 40% usage)



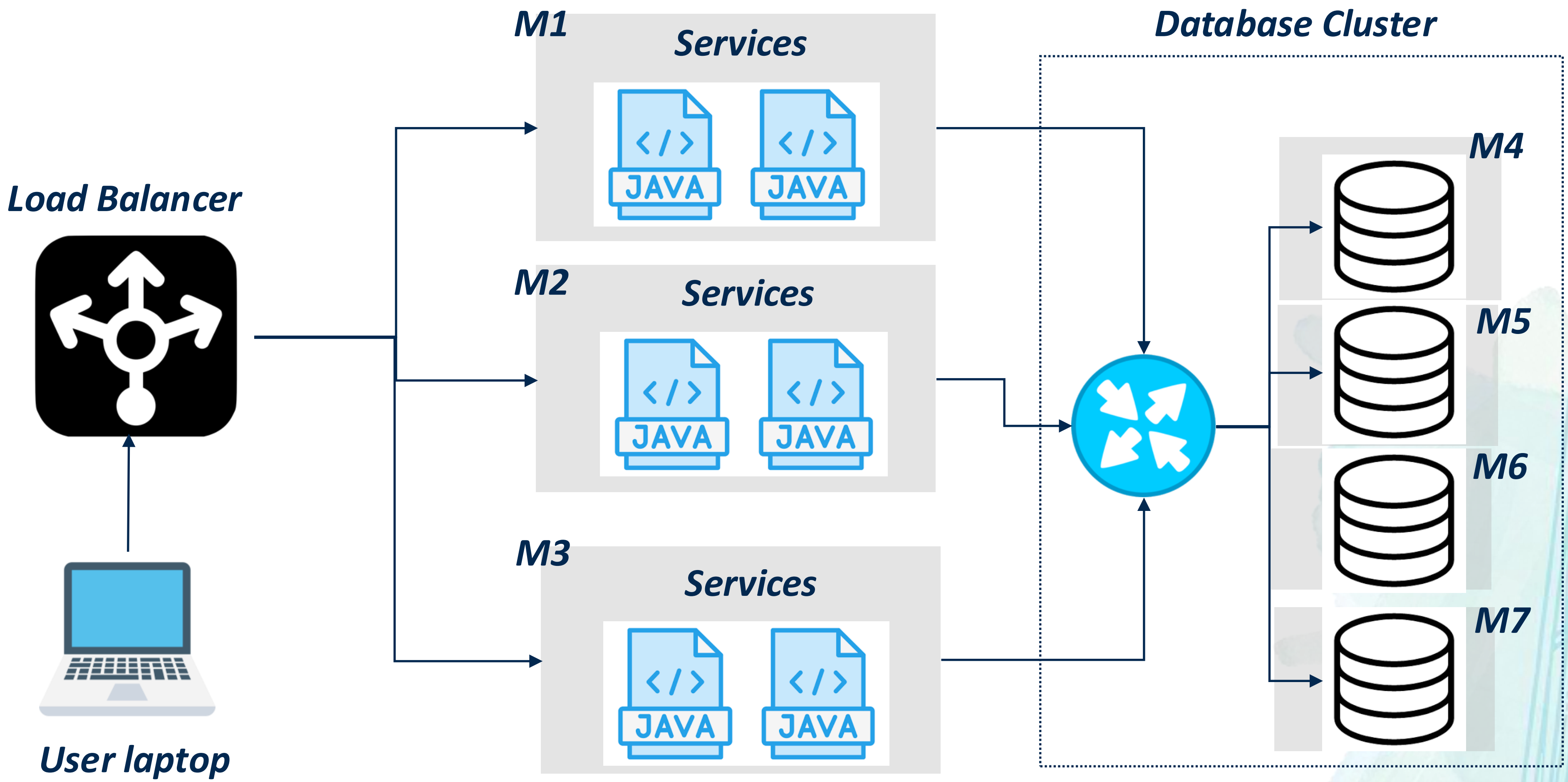
UCDAVIS

Vertical scaling limitations

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

Horizontal scaling for monolithic apps

Add more machines and replicate application on each machine



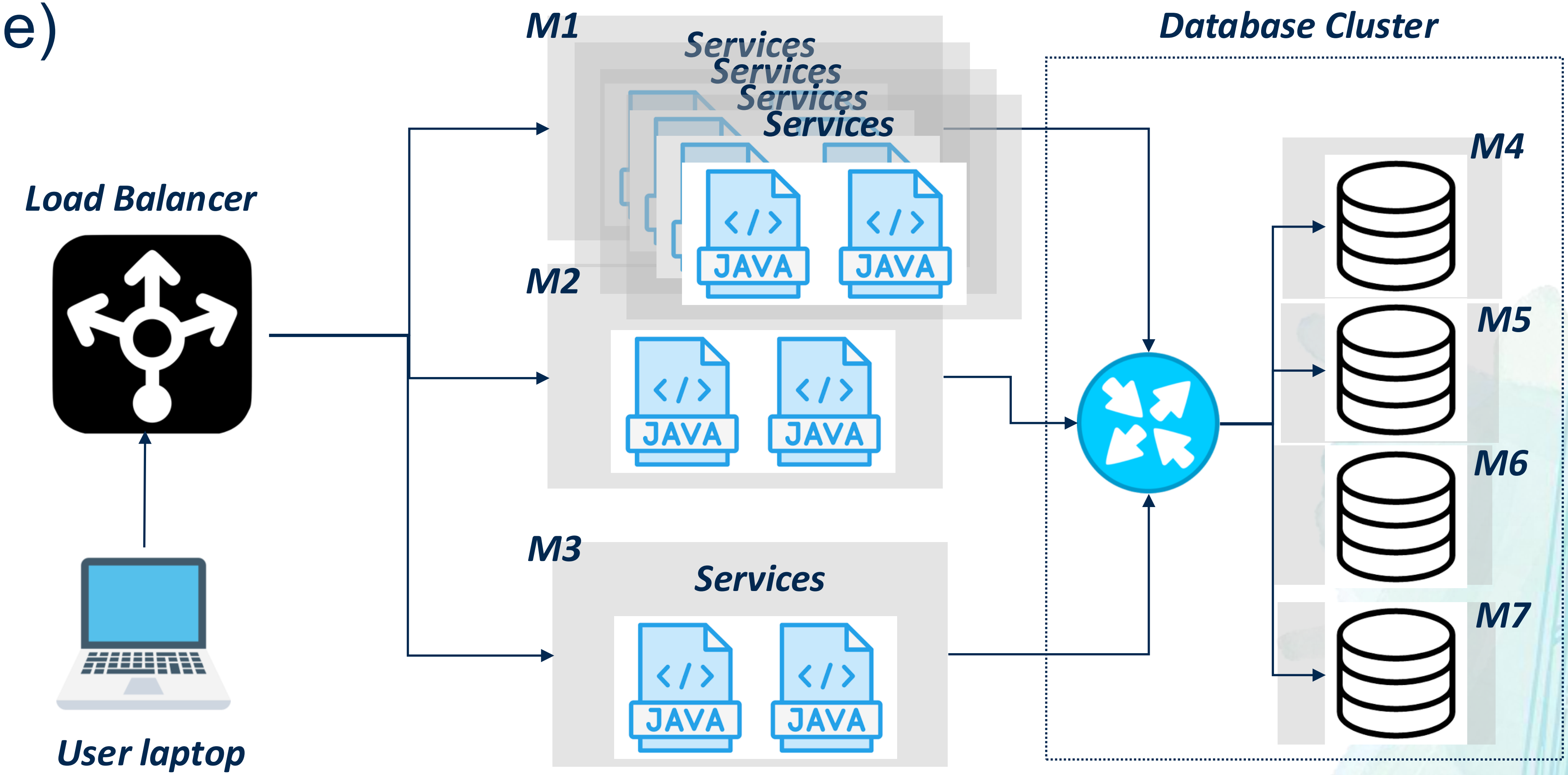
UCDAVIS

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

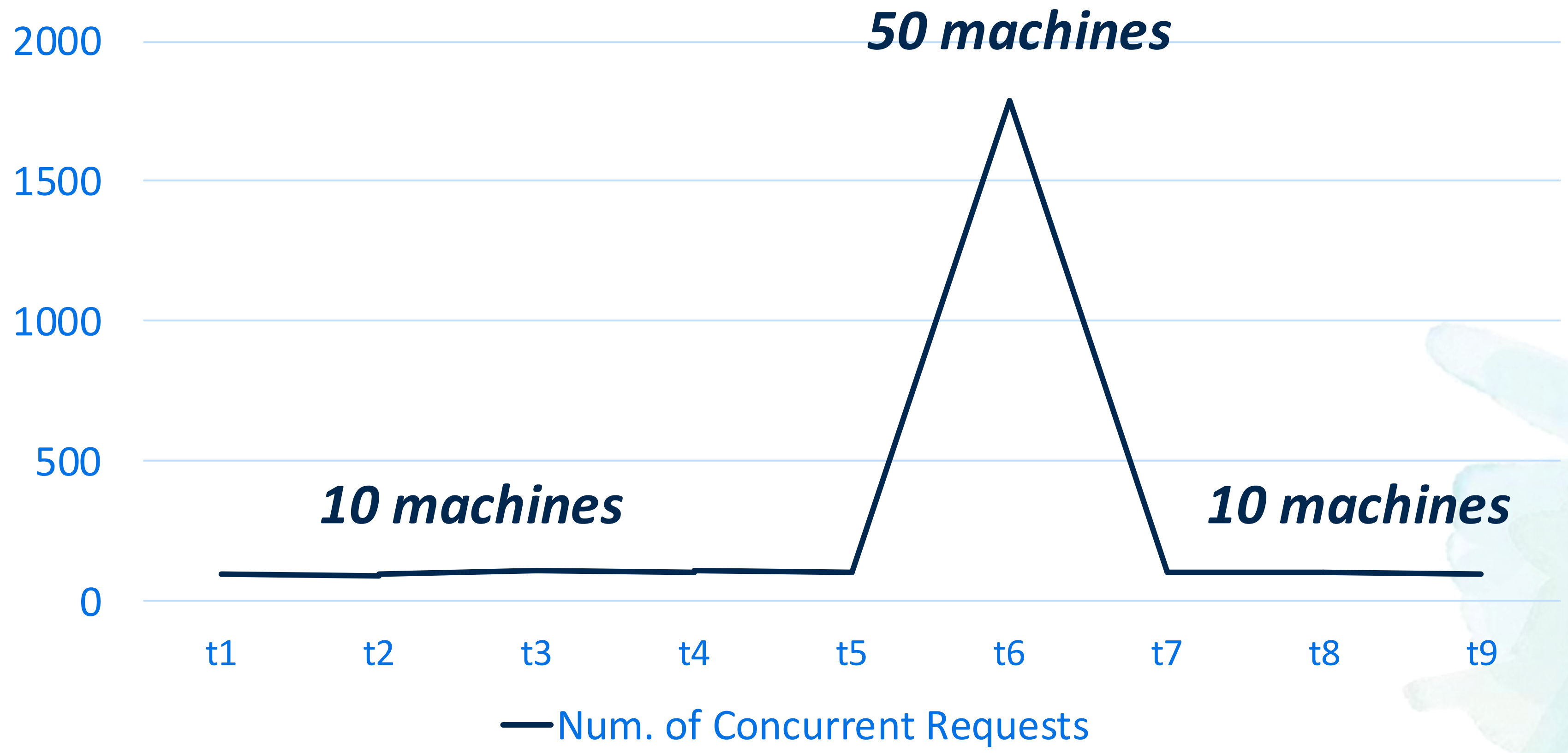
Autoscaling

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



Autoscaling

Minimum resource wastage



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
  - ProductService team should be able to update the Products Tbl schema without consulting
- Need low coupling between services
- Solution: microservices

Microservices

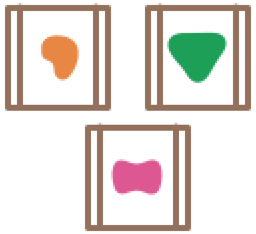
- An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API
- Independently deployable by automated processes
- Bare minimum centralized management
- Smart endpoints connected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

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

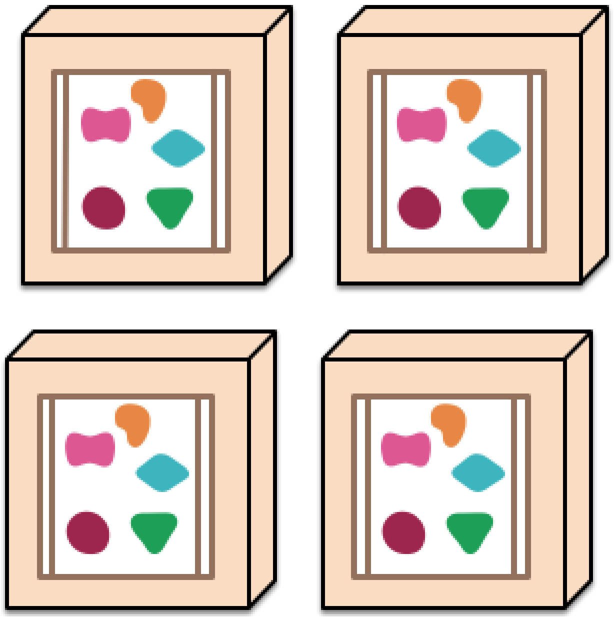
Microservices overview

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

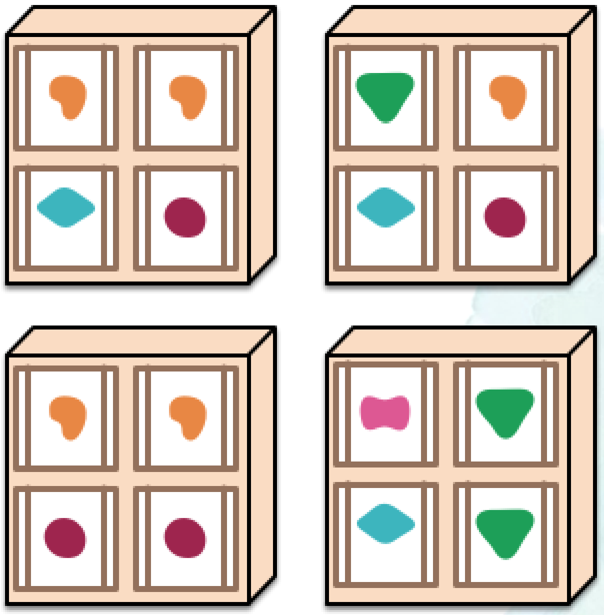
A microservices architecture puts each element of functionality into a separate service…



… and scales by replicating the monolith on multiple servers

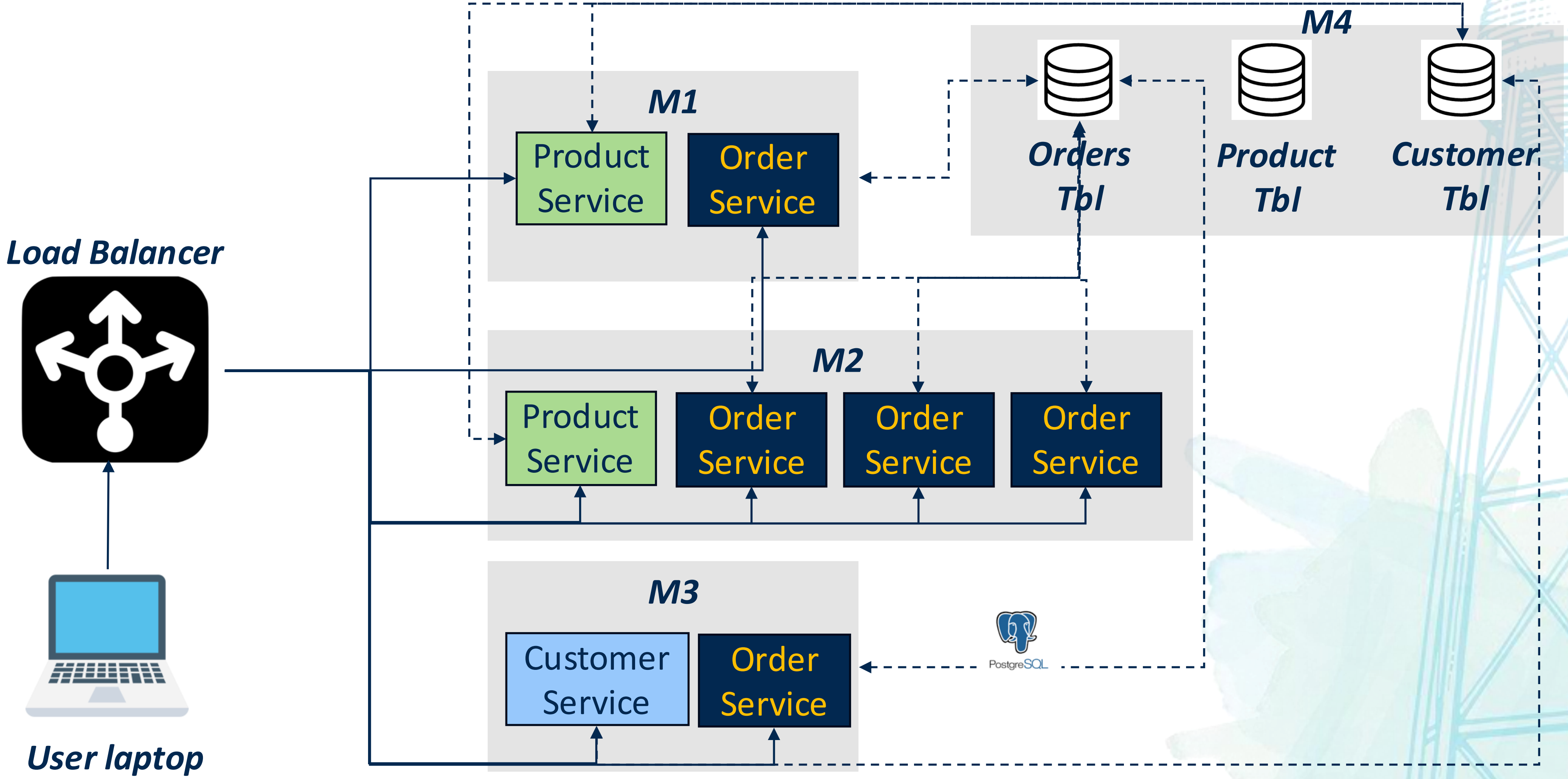


… and scales by distributing these services across servers, replicating as needed.

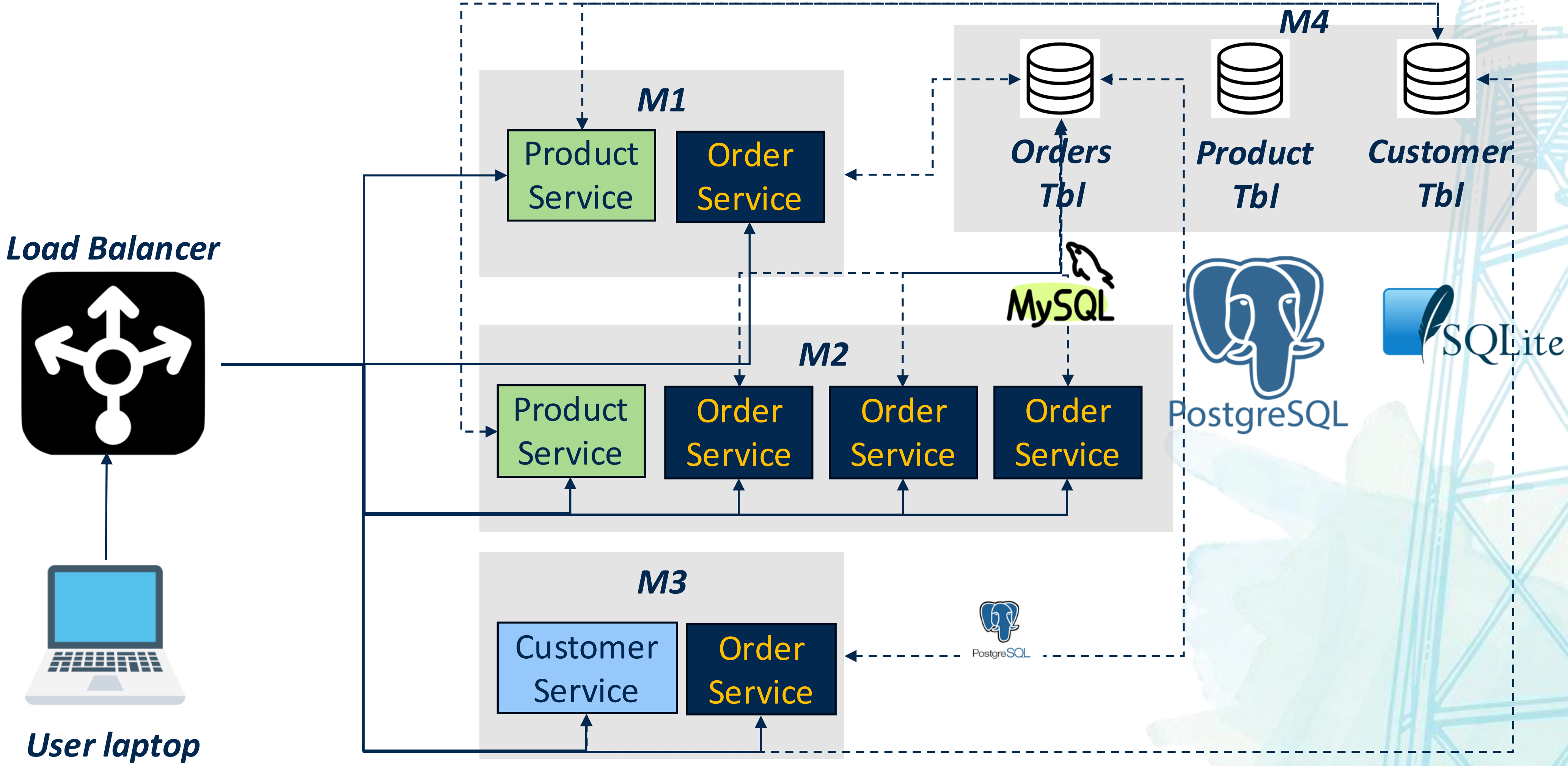


https://martinfowler.com/articles/microservices.html

Microservice architecture for online store



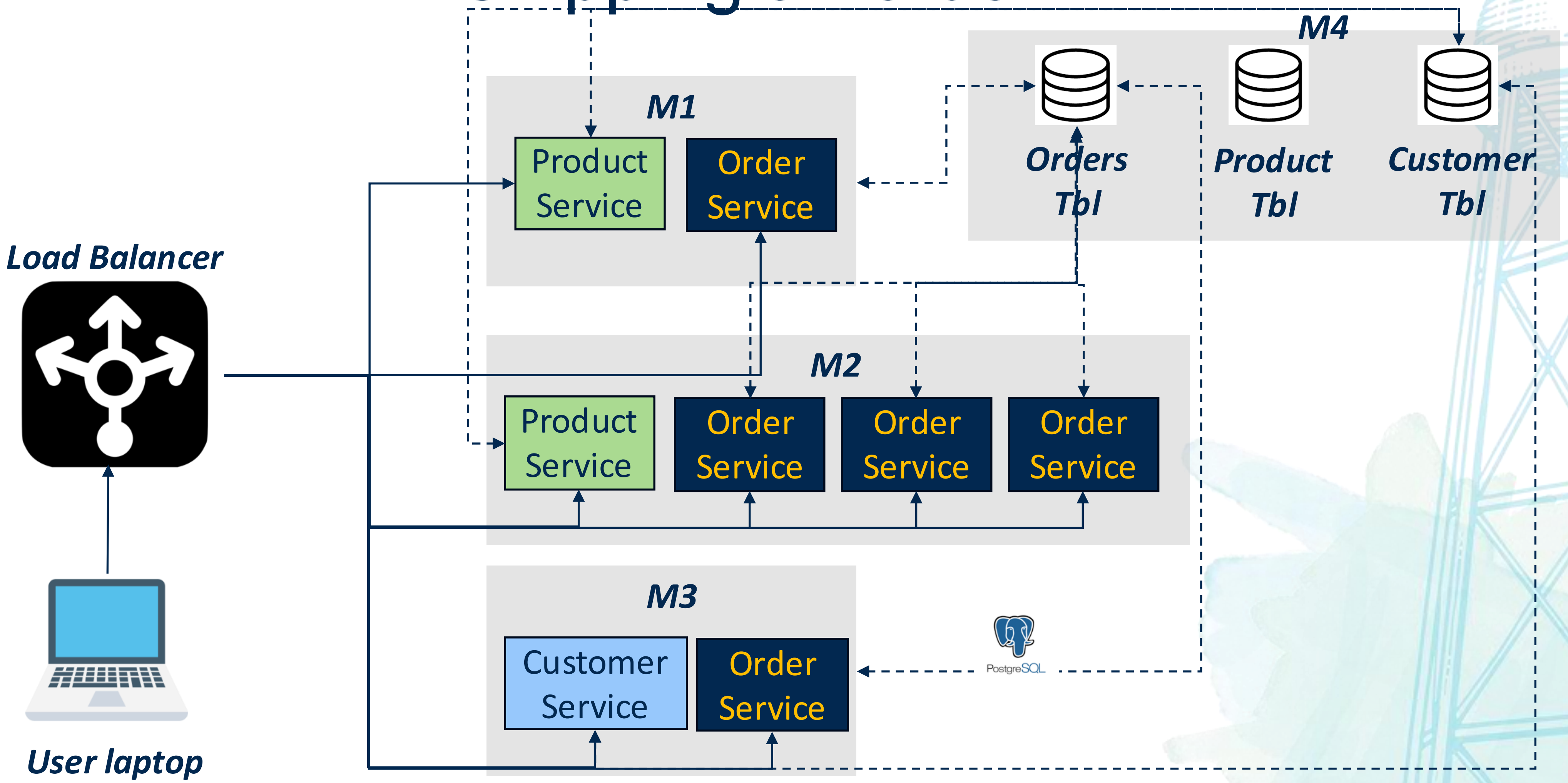
Each microservice chooses its own database



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



Shipping an order

Product MicroService

Order MicroService

Customer MicroService

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	ProductId	CustomerId
110001	1001	9001

Cannot perform joins!

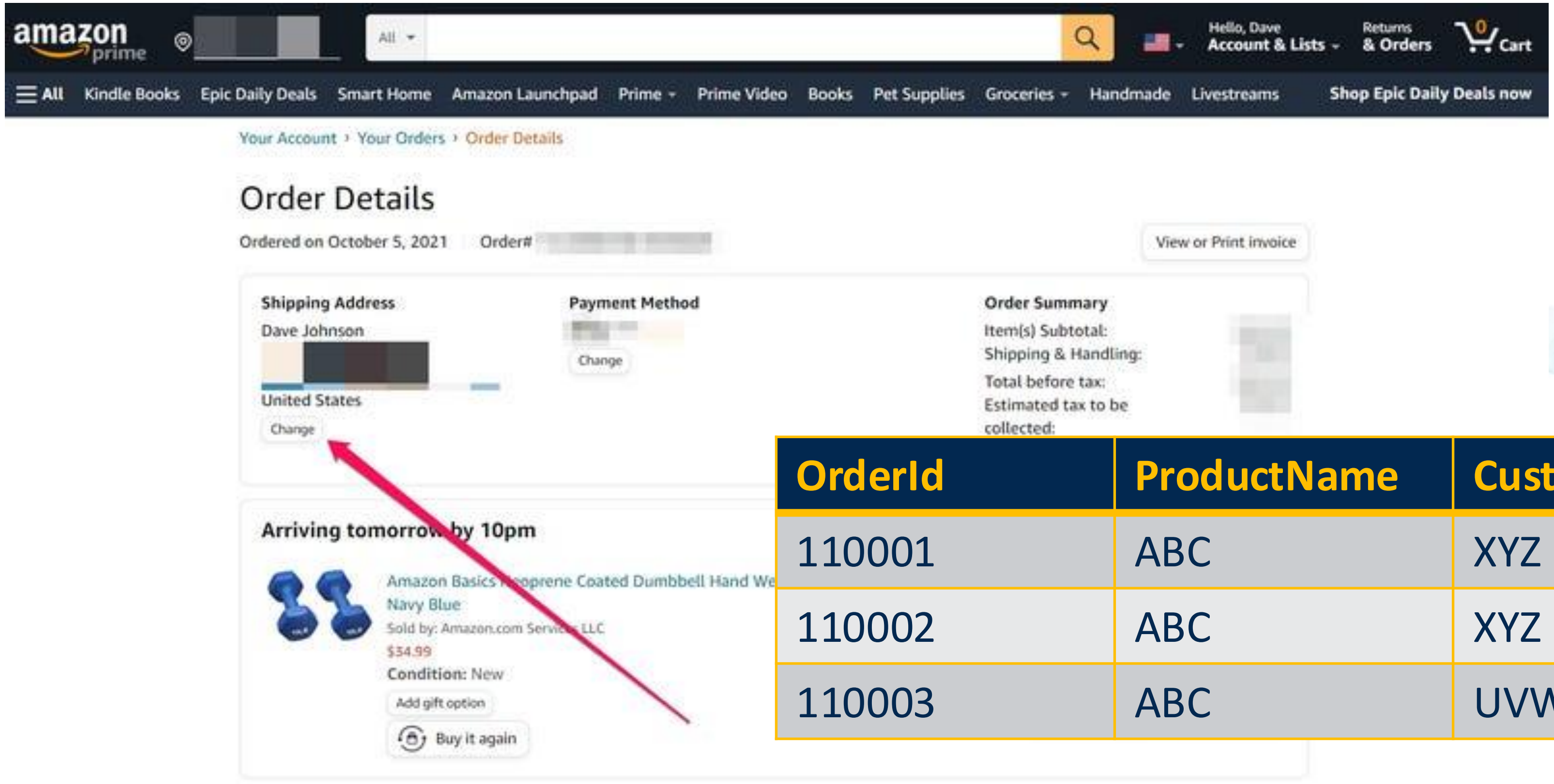
Solution – denormalization of data

- Must denormalize the data
- DO NOT want to query the Customer and Product microservices when shipping
- Will increase network overhead

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

Solution – denormalization of data

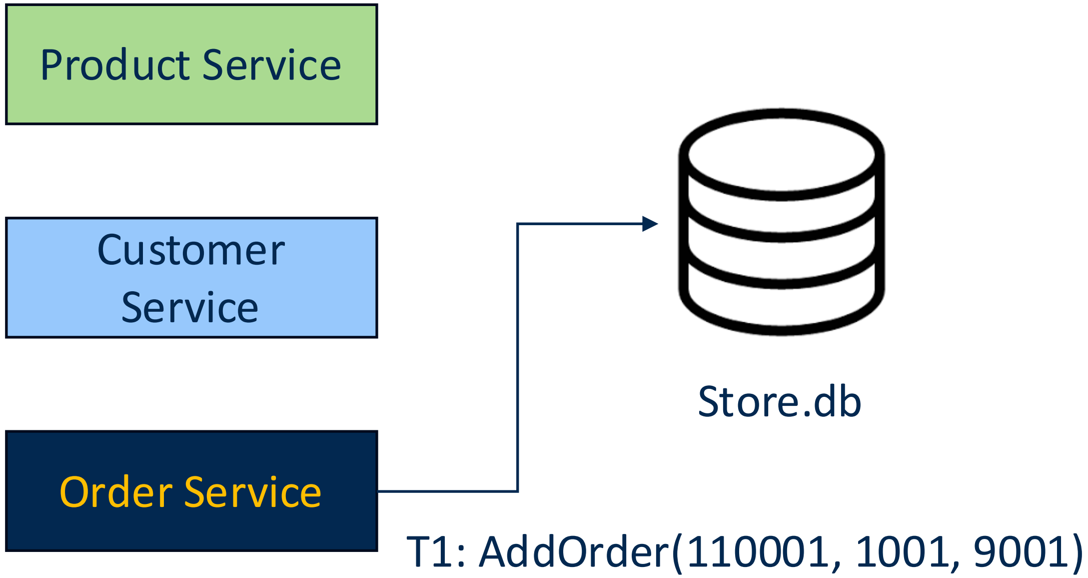
Typical solution: user confirms the Customer data on the Order HTML page



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



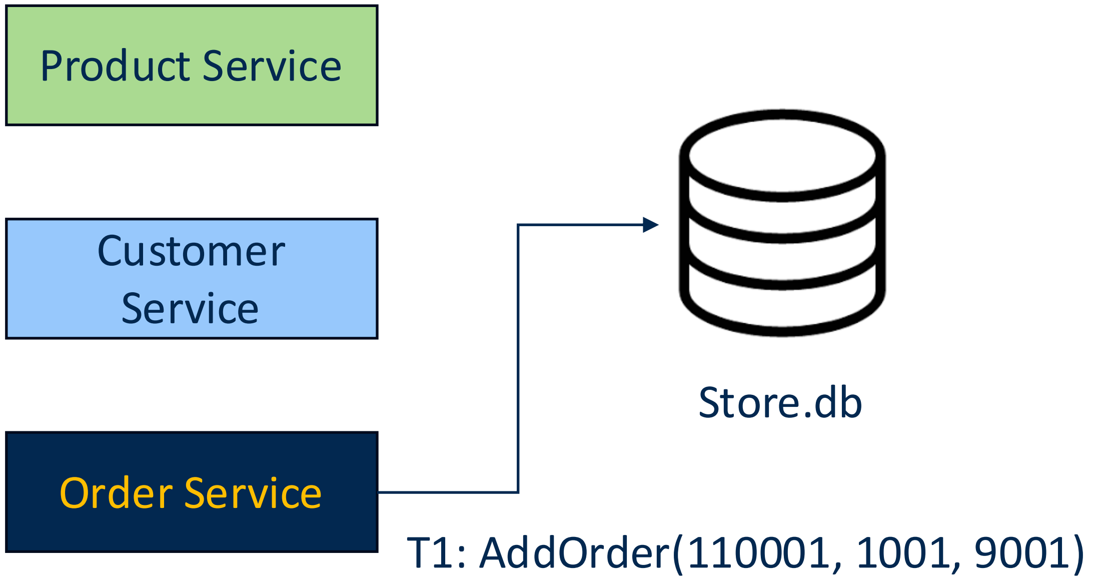
ProductId	ProductName	ProductCount
1001	ABC	10

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



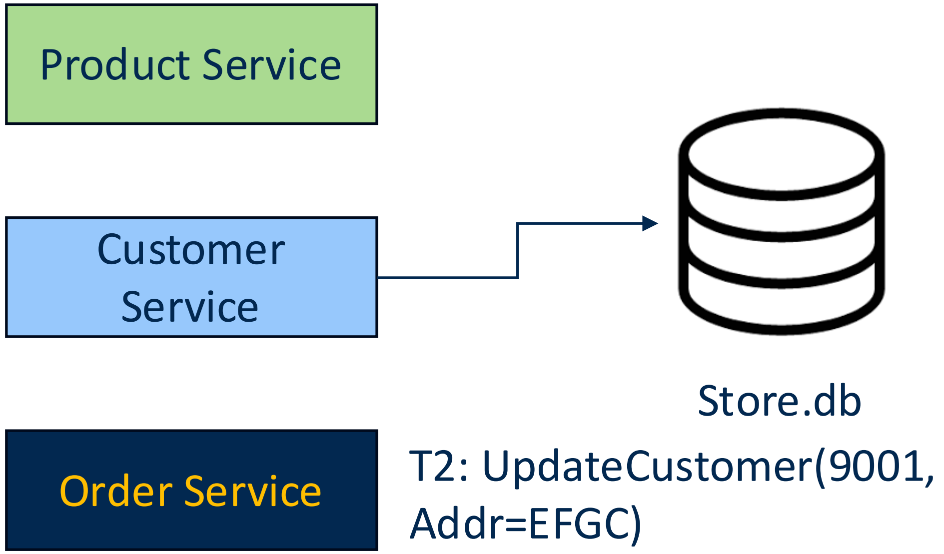
ProductId	ProductName	ProductCount
1001	ABC	10

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



ProductId	ProductName	ProductCount
1001	ABC	10

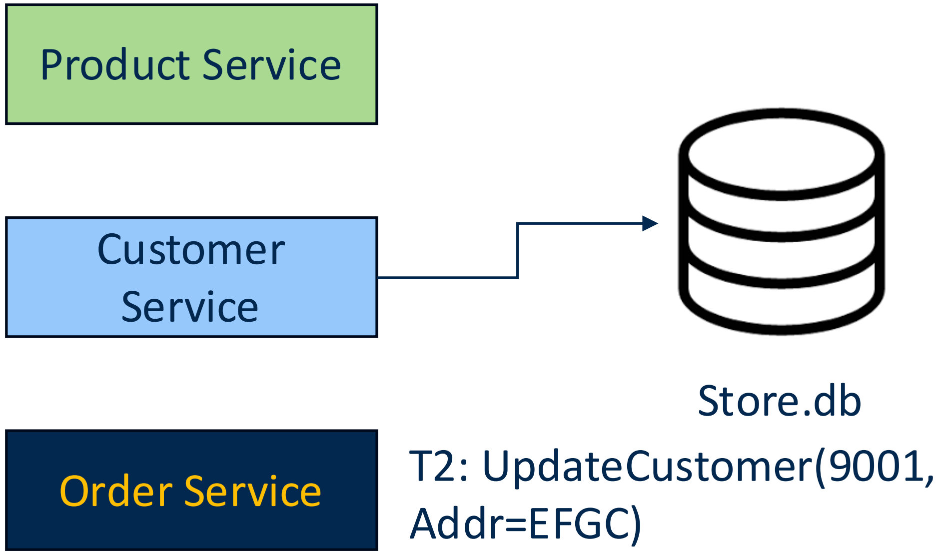
CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

OrderId	ProductId	CustomerId
110001	1001	9001

UCDAVIS

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



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

Product Service

Customer Service

Order Service

Store.db

T3: ShipOrder -> JOIN(1001, 9001) -> (ABC, UVW, EFGC)

ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

OrderId	ProductId	CustomerId
110001	1001	9001

Microservice consistency guarantees

- Microservice-oriented apps are distributed
- Enforcing strong consistency guarantees in distributed systems is very hard due to network overhead, network partition, node failures
- Typically have eventual consistency guarantees – updates are eventually visible to all components
- System must work around these limitations
- Core idea: a microservice ***owns*** some data
  - ***Notifies*** dependent microservices of change
  - ***No guarantees*** on when the updates synchronize

Working with eventual consistency

Product MicroService — Products.db

ProductId	ProductName	ProductCount
1001	ABC	10

Products tbl

Customer MicroService — Customer.db

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF

Customer tbl

Order MicroService — Orders.db

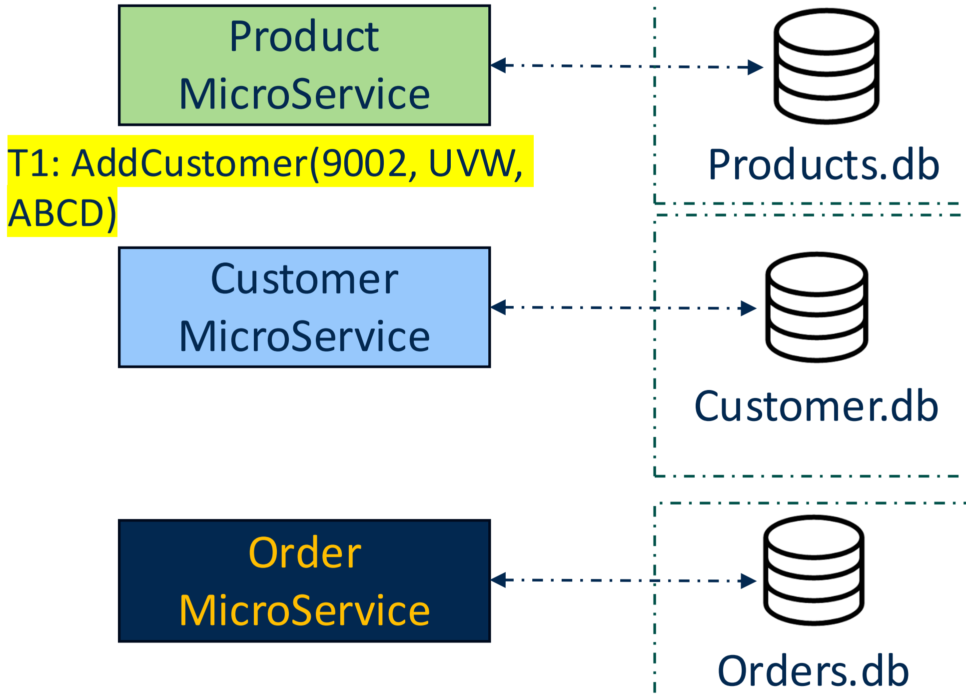
OrderId	ProductName	CustomerAddr

Order tbl

CustomerId	CustomerName	CustomerAddr

Customer Cache tbl

Working with eventual consistency



Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Order tbl

OrderId	ProductName	CustomerAddr

Customer Cache tbl

CustomerId	CustomerName	CustomerAddr

Working with eventual consistency

Product MicroService

Products.db

ProductId	ProductName	ProductCount
1001	ABC	10

Products tbl

Customer MicroService

T2: UpdateCust (9002, UVW, ABCD)

Customer.db

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Customer tbl

Order MicroService

Orders.db

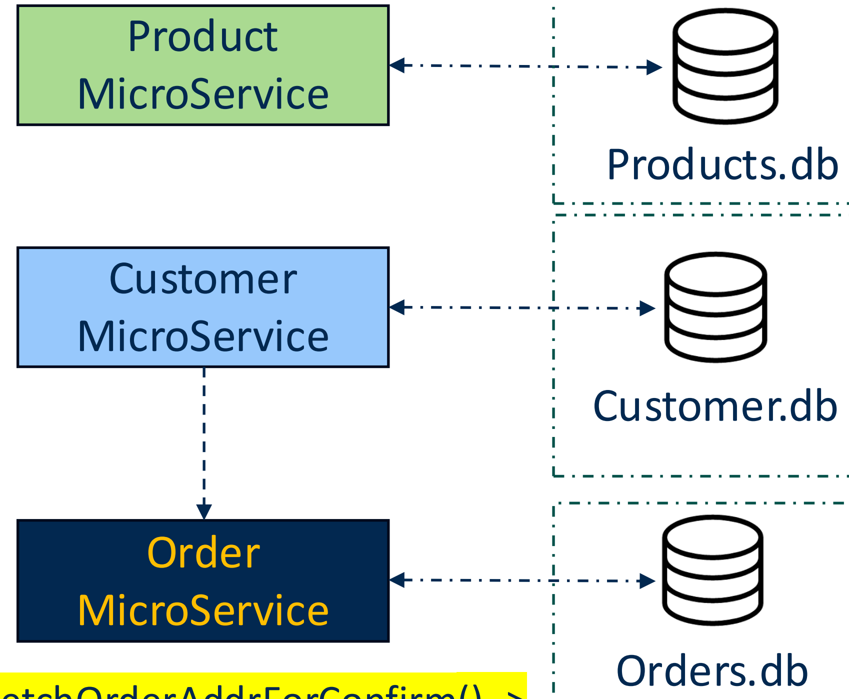
OrderId	ProductName	CustomerAddr

Order tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	ABCD

Customer Cache tbl

Working with eventual consistency



T3: FetchOrderAddrForConfirm() -> (9002, UWV, ABCD)

ProductId	ProductName	ProductCount
1001	ABC	10

Products tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Customer tbl

OrderId	ProductName	CustomerAddr

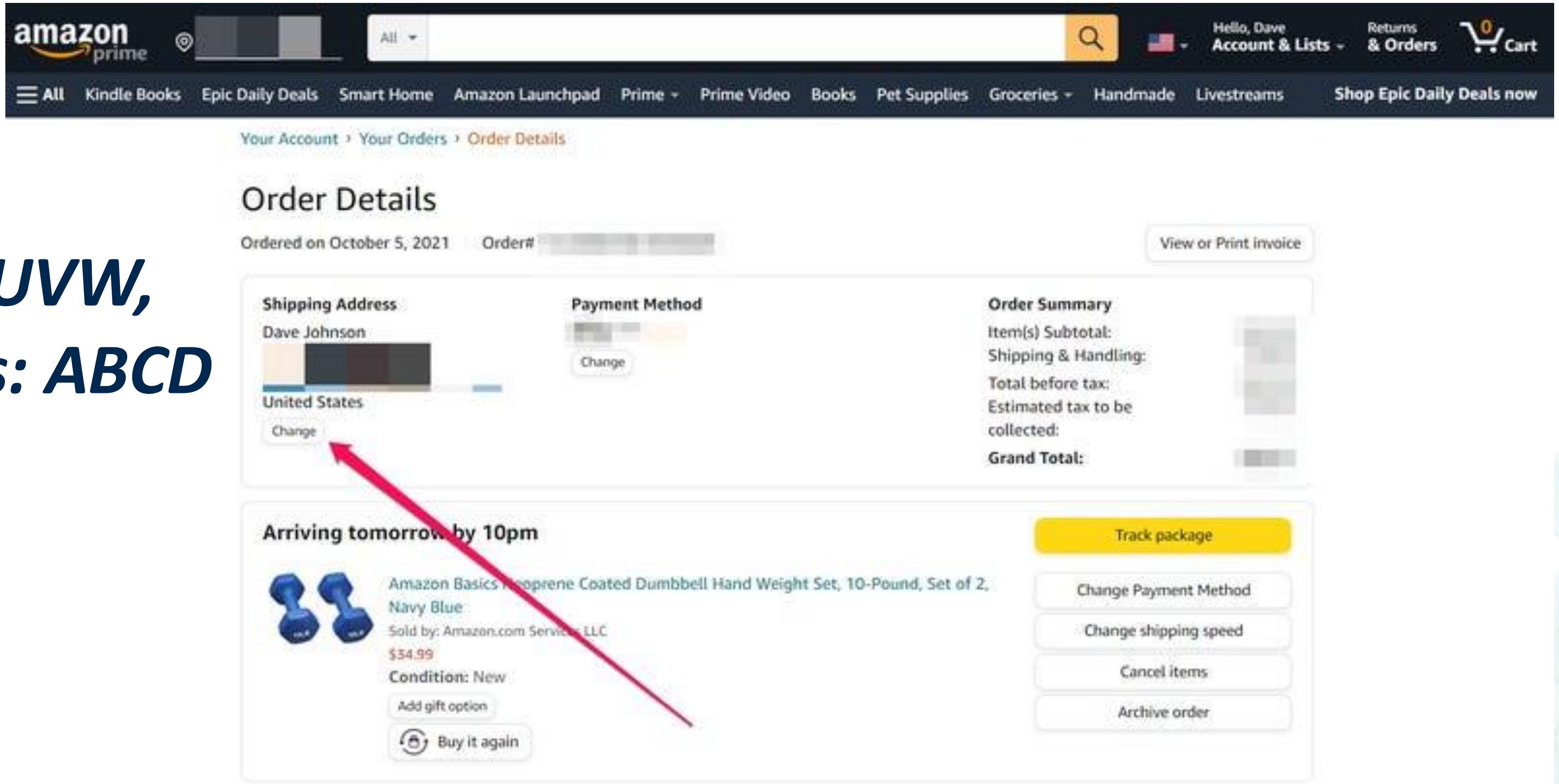
Order tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	ABCD

Customer Cache tbl

Address confirmation page

Name: UVW,
Address: ABCD



UCDAVIS

Working with eventual consistency

Product MicroService

Products.db

ProductId	ProductName	ProductCount
1001	ABC	10

Products tbl

Customer MicroService

Customer.db

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

Customer tbl

Order MicroService

T4: AddOrder(110001, ABC, ABCD)

Orders.db

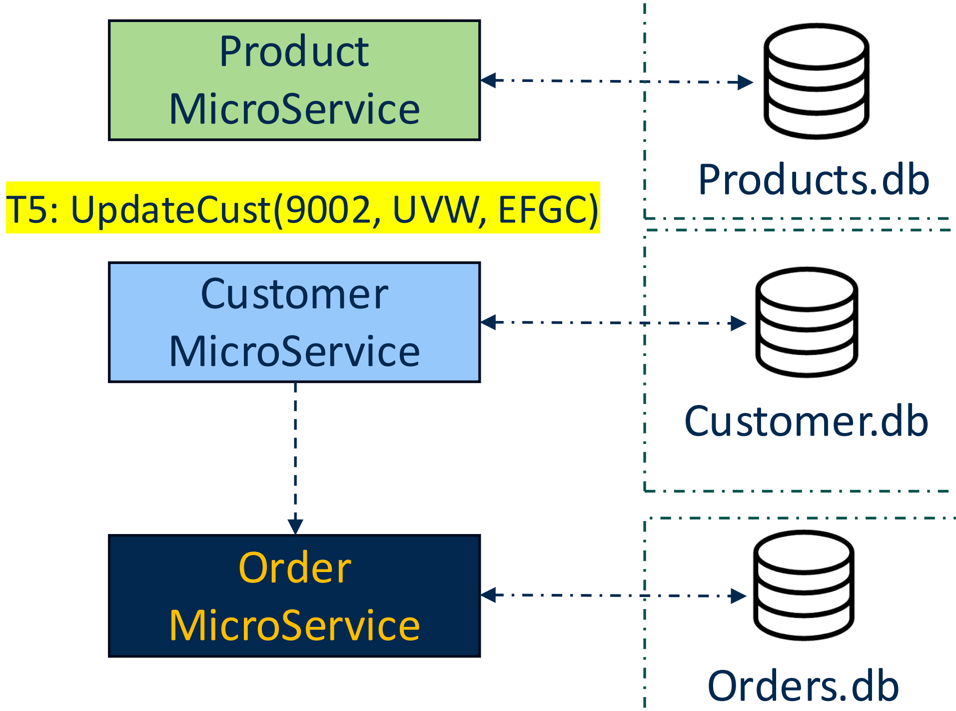
OrderId	ProductName	CustomerAddr
110001	ABC	ABCD

Order tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	ABCD

Customer Cache tbl

Working with eventual consistency



ProductId	ProductName	ProductCount
1001	ABC	10

Products tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

Customer tbl

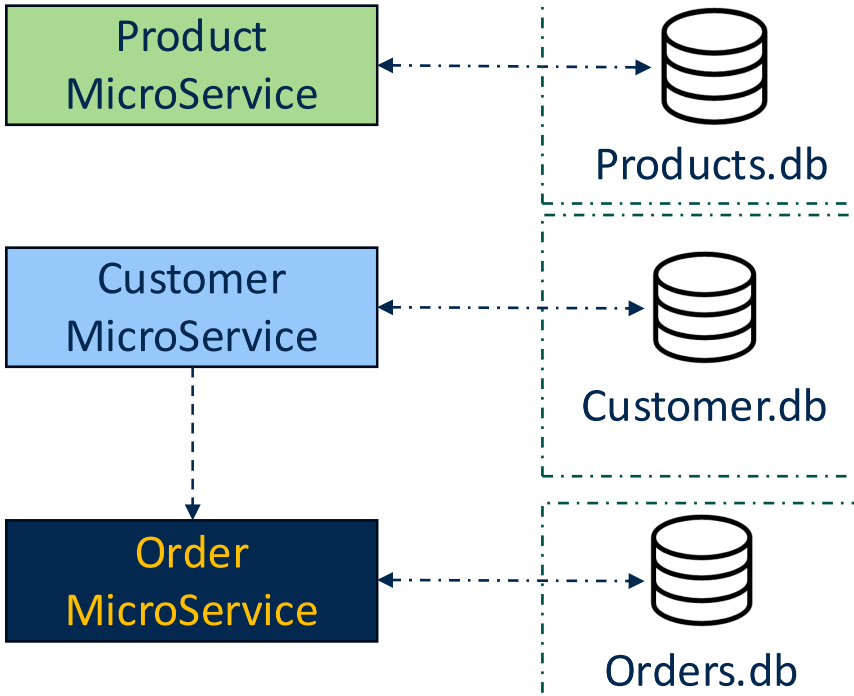
OrderId	ProductName	CustomerAddr
110001	ABC	ABCD

Order tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	ABCD

Customer Cache tbl

Working with eventual consistency



Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

Order tbl

OrderId	ProductName	CustomerAddr
110001	ABC	ABCD

Customer Cache tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	ABCD

T6: ShipOrder(11001, ABC, ABCD)

Working with eventual consistency

Product MicroService

Products.db

Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer MicroService

T7: UpdateCustCache(9002, UVW, EFGC)

Customer.db

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

Order MicroService

Orders.db

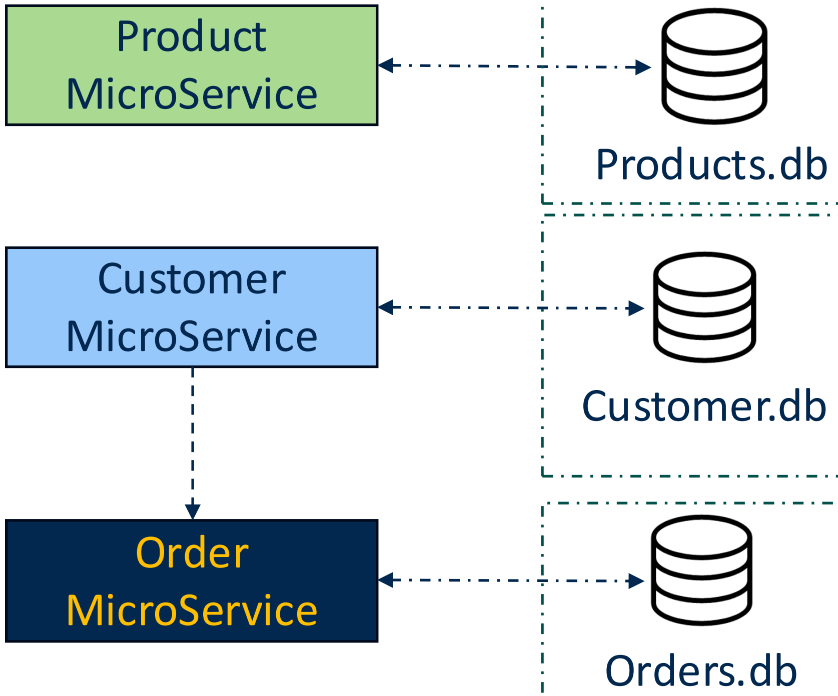
Order tbl

OrderId	ProductName	CustomerAddr
110001	ABC	ABCD

Customer Cache tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	EFGC

Working with eventual consistency



Products tbl

ProductId	ProductName	ProductCount
1001	ABC	10

Customer tbl

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

Order tbl

OrderId	ProductName	CustomerAddr
110001	ABC	ABCD

Customer Cache tbl

CustomerId	CustomerName	CustomerAddr
9002	UVW	EFGC

Subsequent orders load updated customer address

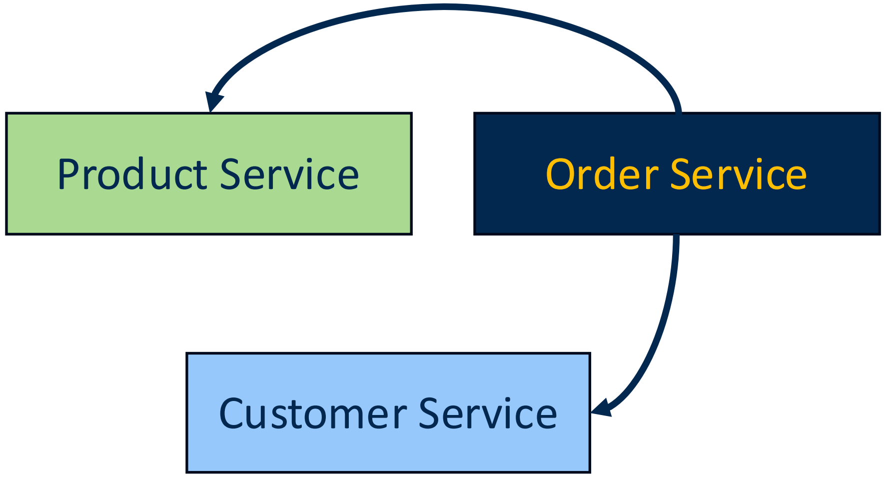
Microservice design concerns

- Solution is generally use-case dependent
- General goal - limit network communication

Network communication cost

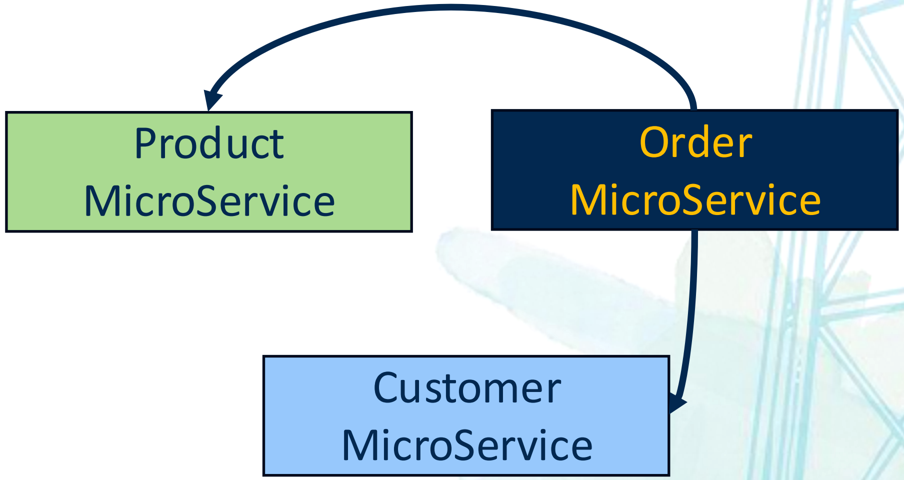
Monolithic applications

Function calls (few nanoseconds)



Microservices

Network RPC calls (few microseconds)

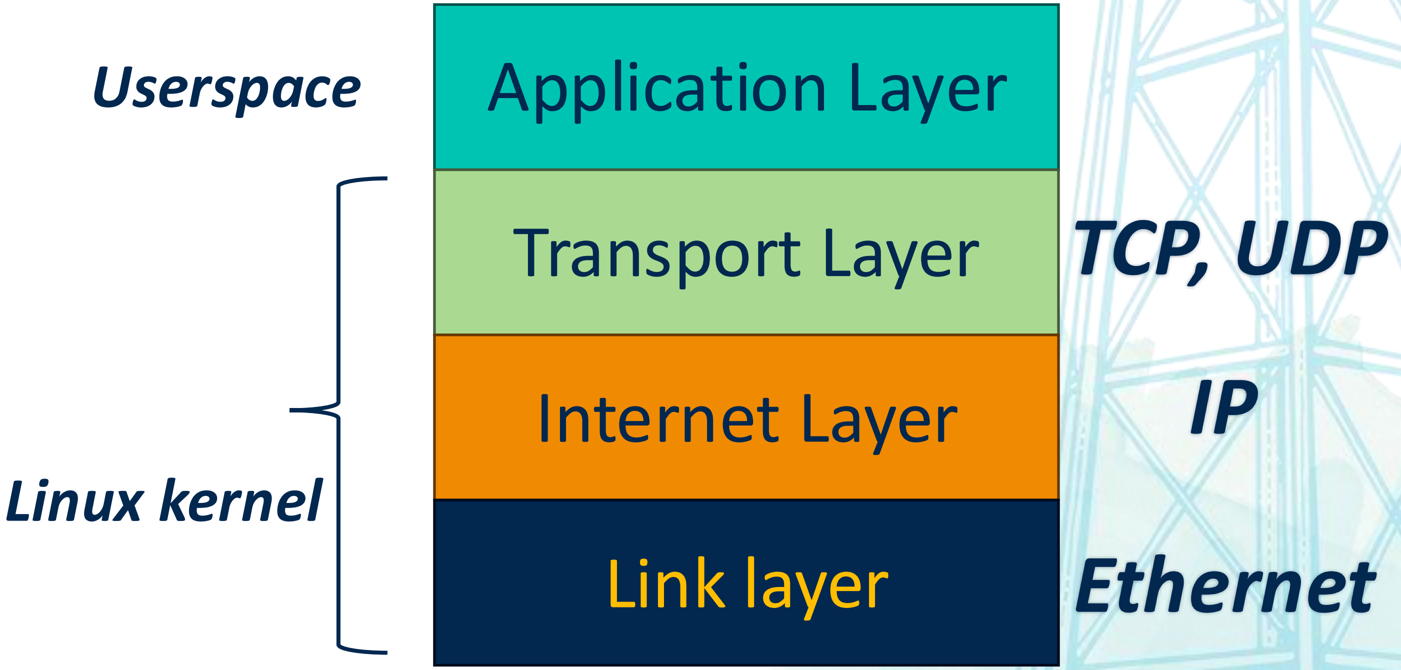


Network overhead decomposition

- 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
- Kernel overhead

Kernel overhead for network comm.

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

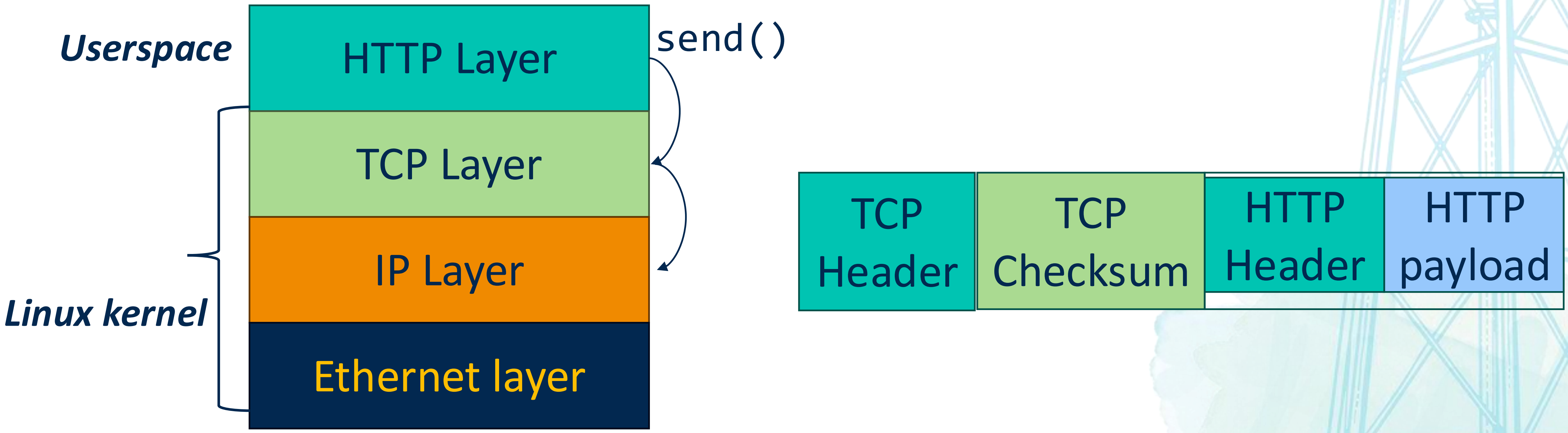


Each layer has a protocol

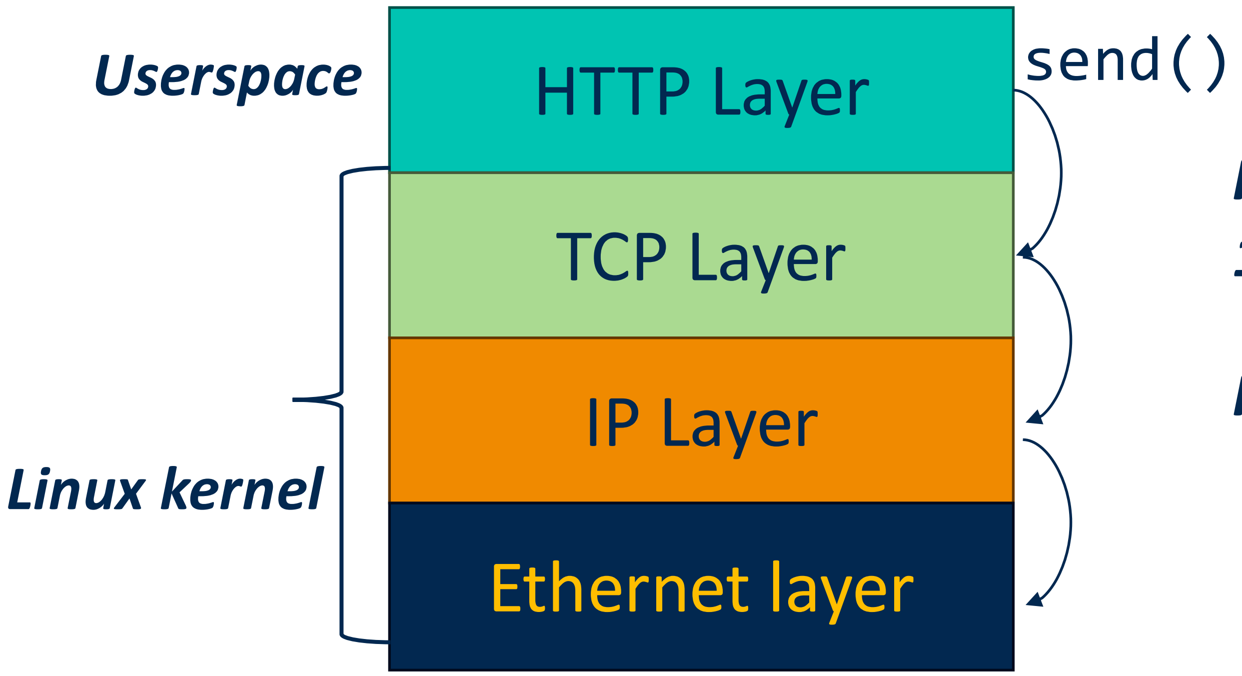
- Every layer/protocol has a fixed message format
  - Header
  - Payload
  - [optional] Checksum

- As the packet traverses through the layers, packets are rewrapped

Life of a packet

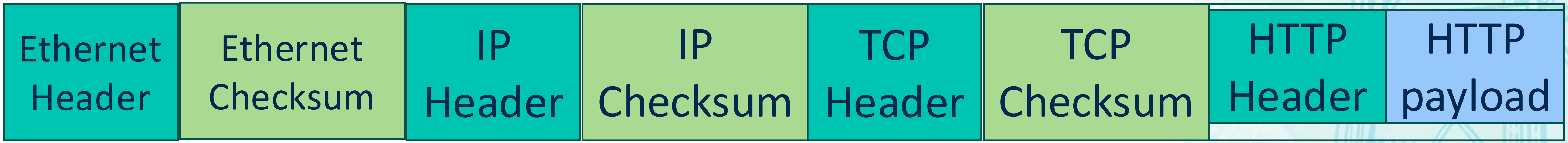


Life of a packet



Packet encapsulation process can take 100+ microseconds

Reading 6 will discuss an alternative



Microservice pros

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

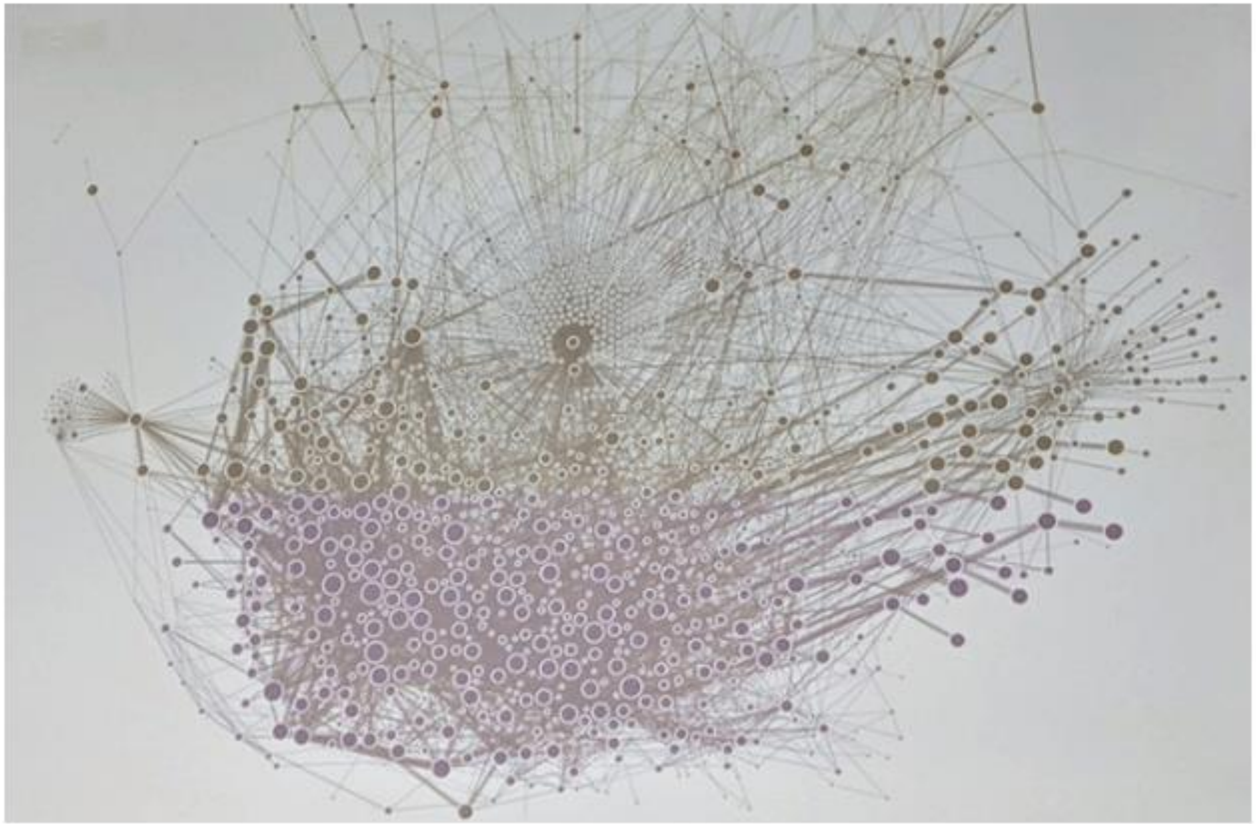
Microservice cons

- 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
- Throughput – number of operations performed in unit time
  - Rps – requests per second
  - Ops – operations per second
- Microservices increase latency compared to monolithic operation
- Microservices significantly improve throughput
- Note: throughput of a system is max when utilization -> 100% (memory, CPU, network), but latency spikes dramatically due to ***queuing effect***

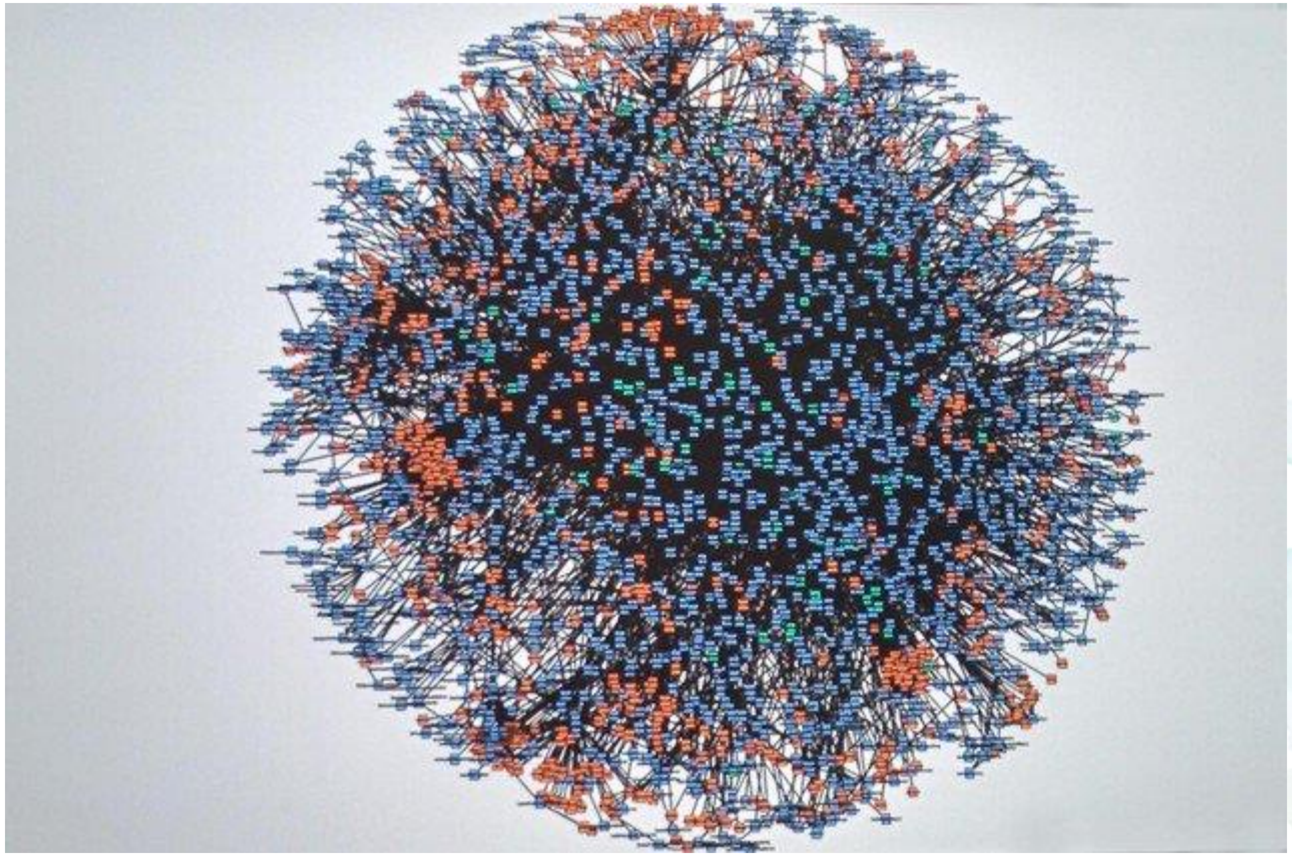
Microservices at Uber (2019)



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?

Spring Boot overview

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

URL

```
POST /users HTTP/1.1
Content-Type: application/json{
   "name": "John Doe",
   "email": john.doe@example.com
}
```

Post “body”

UCDAVIS

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

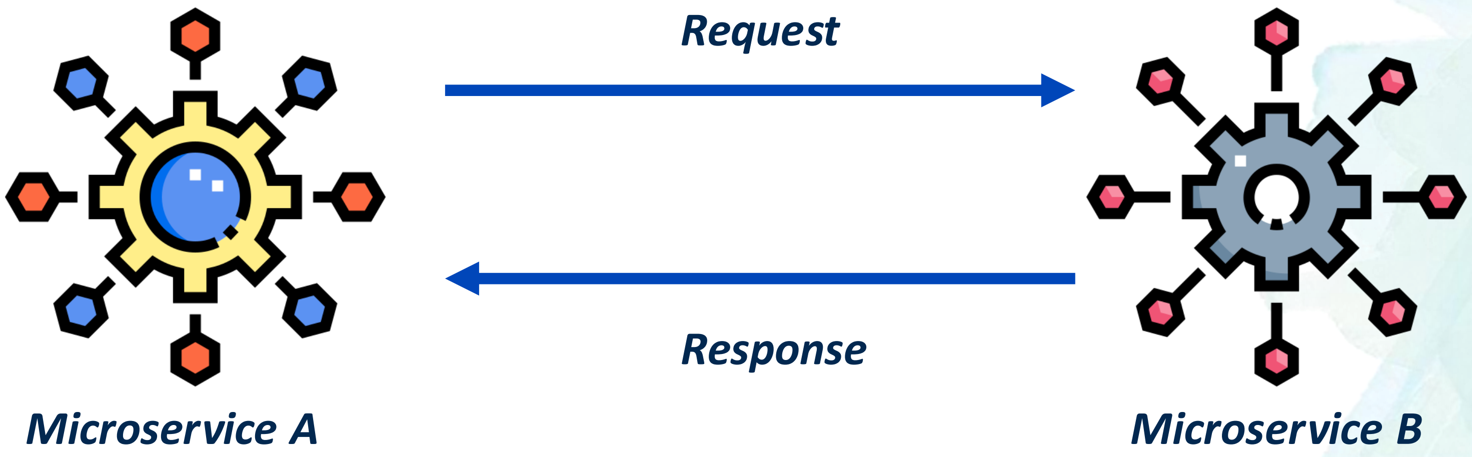
- Uses reflection to first look up all classes with @RestController annotation
- Then automatically creates Servlets out of the methods annotated with @GetMapping, @PostMapping, etc.
- Uses reflection to parse the request parameters into class objects annotated with @RequestBody
- Generates the WAR file 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 "";
    }
}
```

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?***

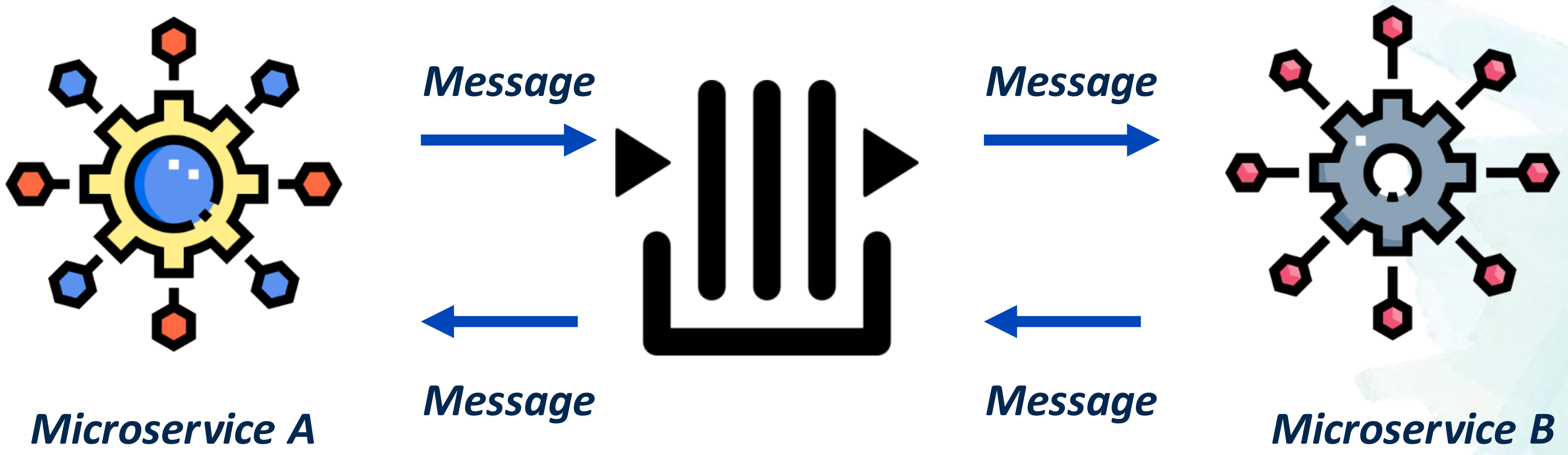
JSON and REST APIs

- 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

RabbitMQ

APACHE kafka™

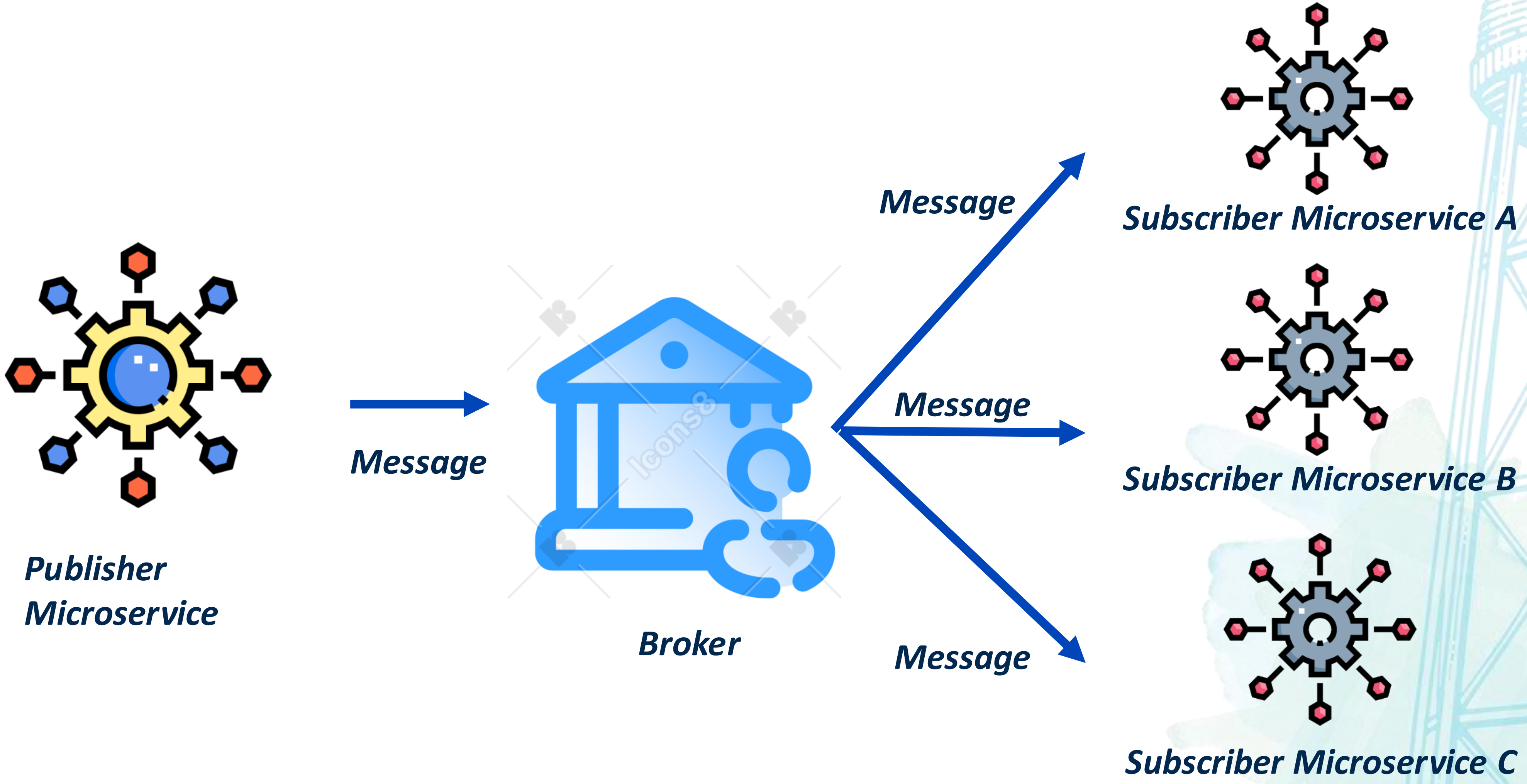
APACHE ACTIVEMQ®

amazon SQS

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

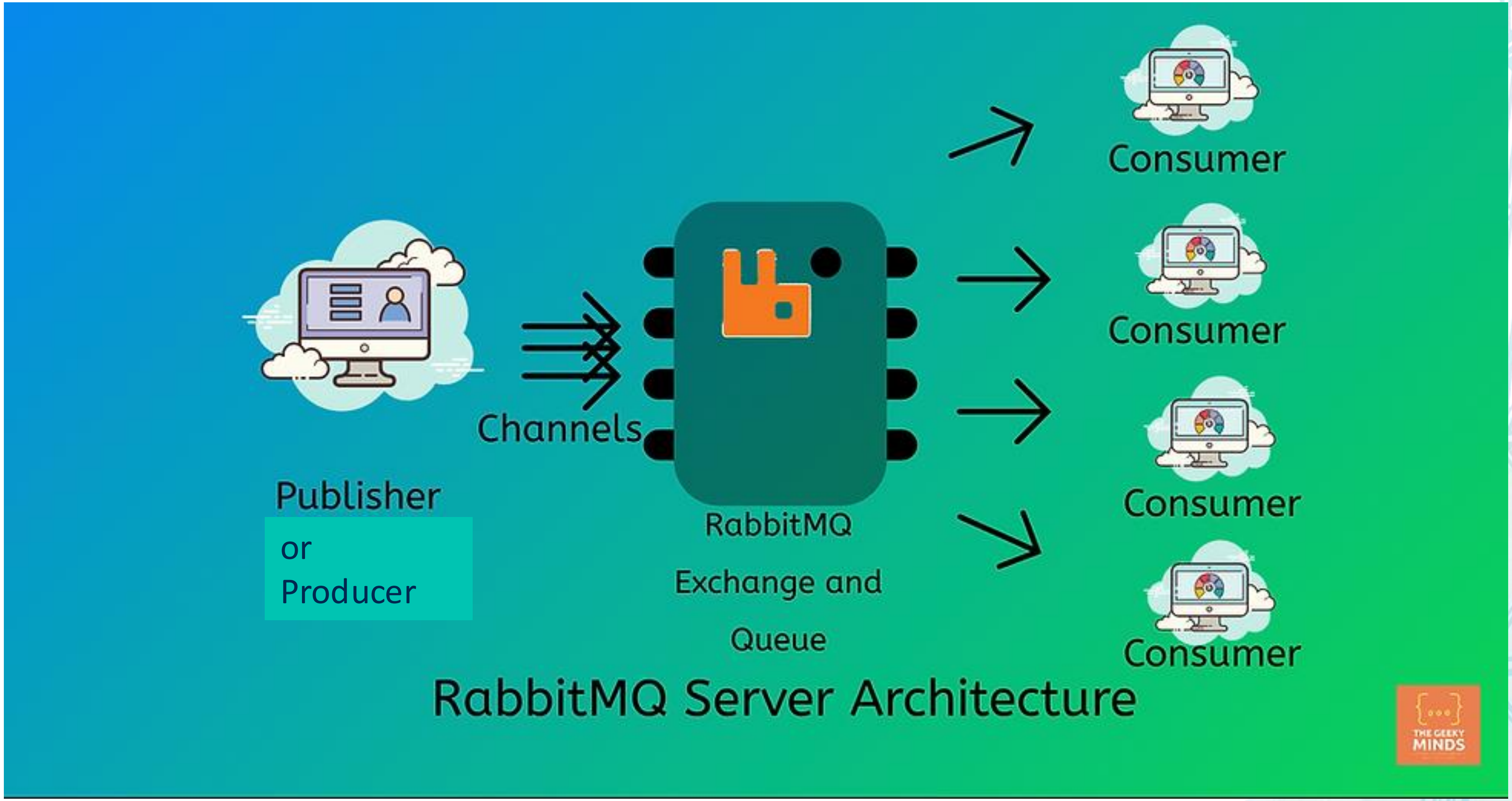
Google Cloud Pub/Sub

Redis Pub Sub

APACHE kafka™

RabbitMQ

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();
        }
    }
}
```

Can mix and match!!

- Pipeline architecture using microservices with message queues deployed on a serverless architecture
- Pub-sub architecture with microservices using containers
- Pipeline architecture in one part of the system, with a pub-sub in another
- … and so on
- Pick what is right for your software system!