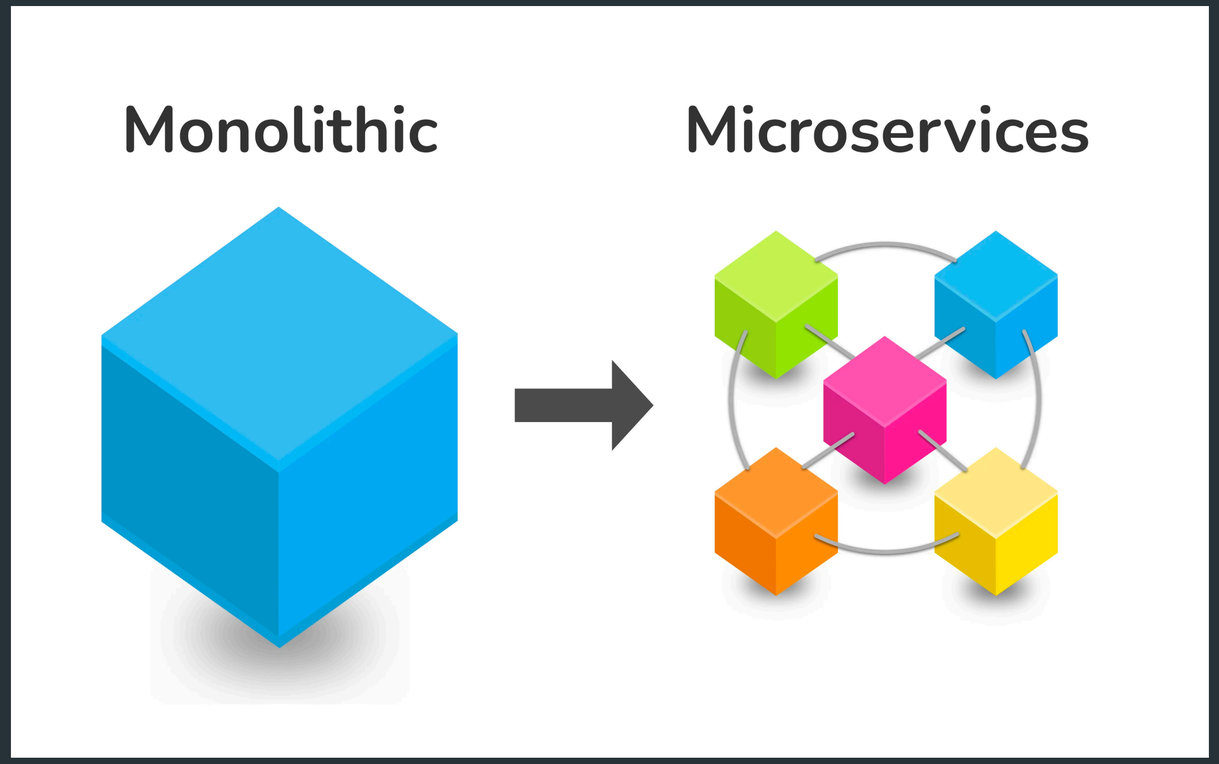
Lecture #9
Microservices

WSMT2023

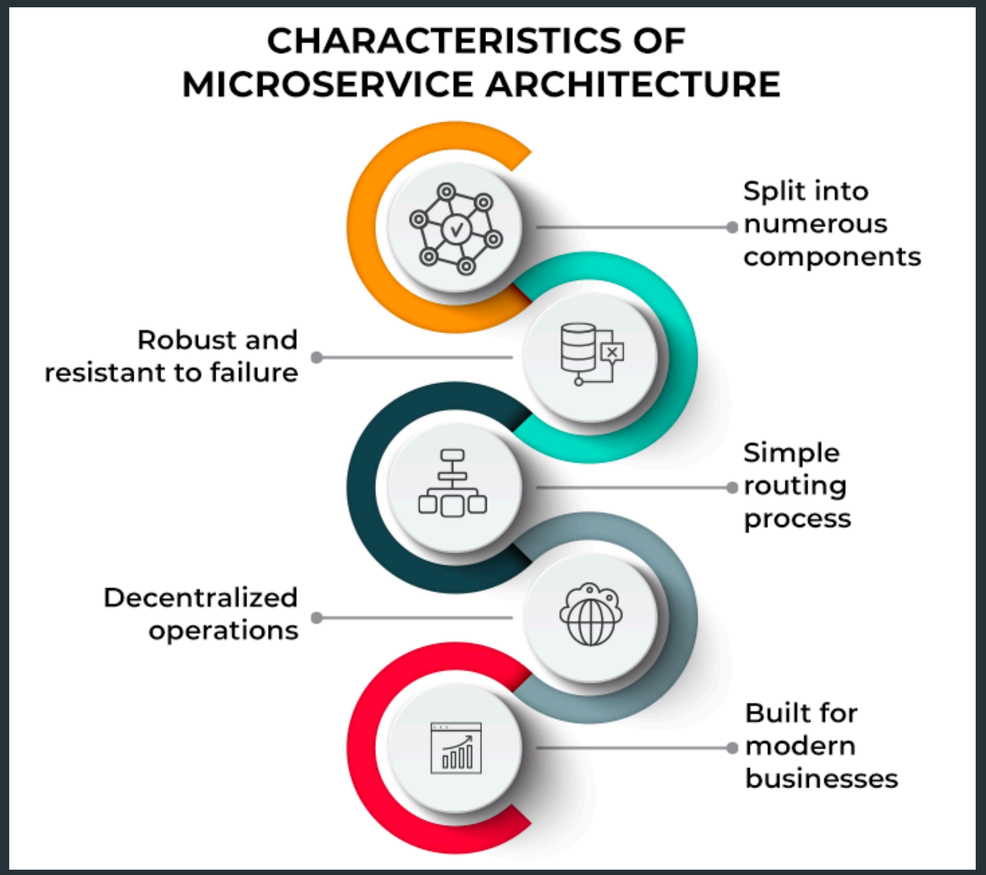
Microservices at a Glance

- Microservices is a software architecture style that structures an application as a suite of small, independent services.
- Each service is self-contained and performs a single function.
- Services communicate with each other using well-defined APIs.



Key Concepts of Microservices

- Independent services
- Loose coupling
- Well-defined APIs

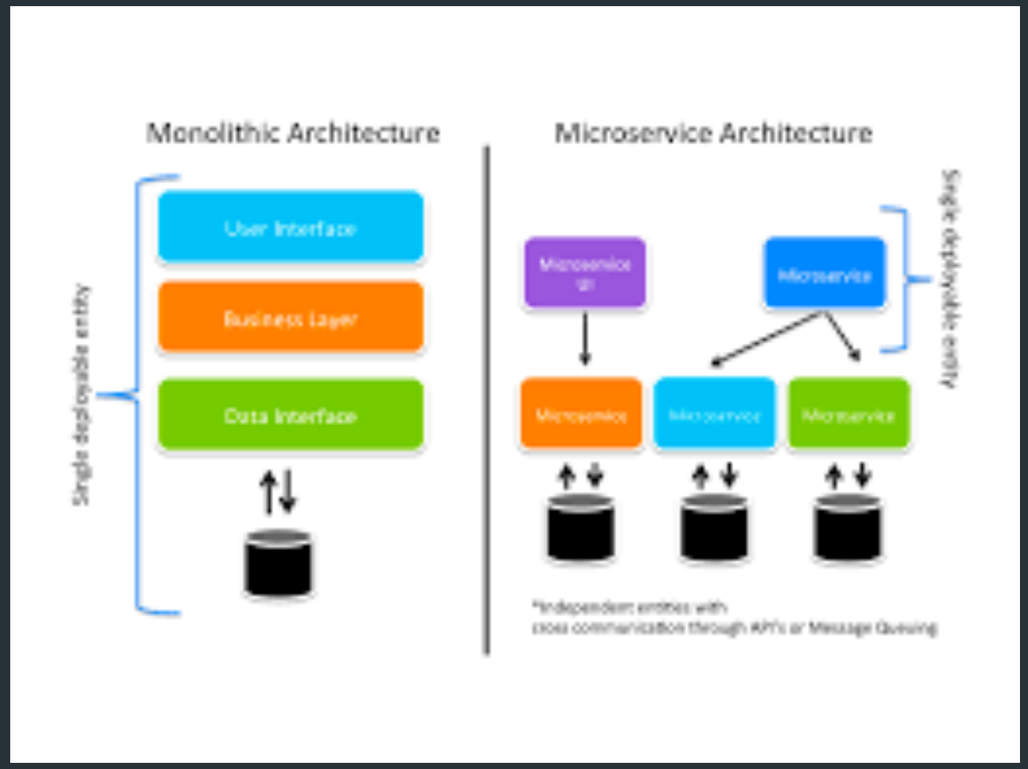


Example

```
import requests

def get_time():
    url = "https://api.example.com/time"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()["time"]
    else:
        raise Exception("Error retrieving time")
```

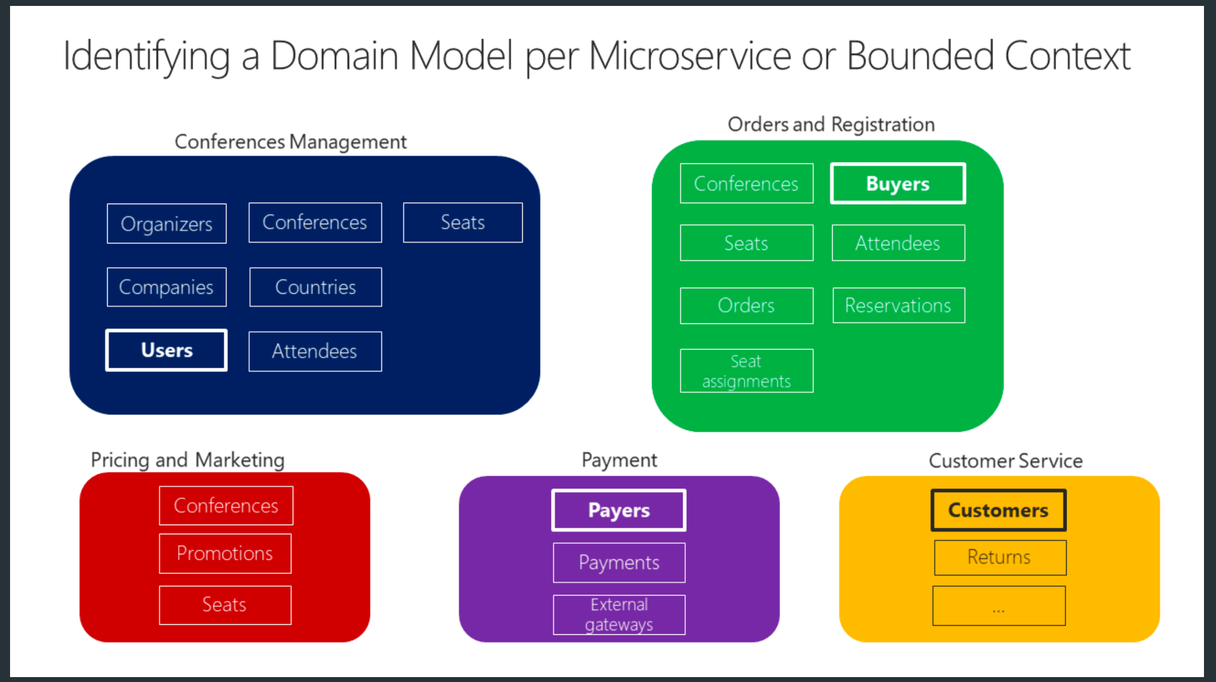
Independent Deployability

- Independent deployability is one of the key concepts of microservices architecture.
- It means that each microservice can be deployed independently of the other microservices.
- This allows for greater agility and flexibility in development and deployment.



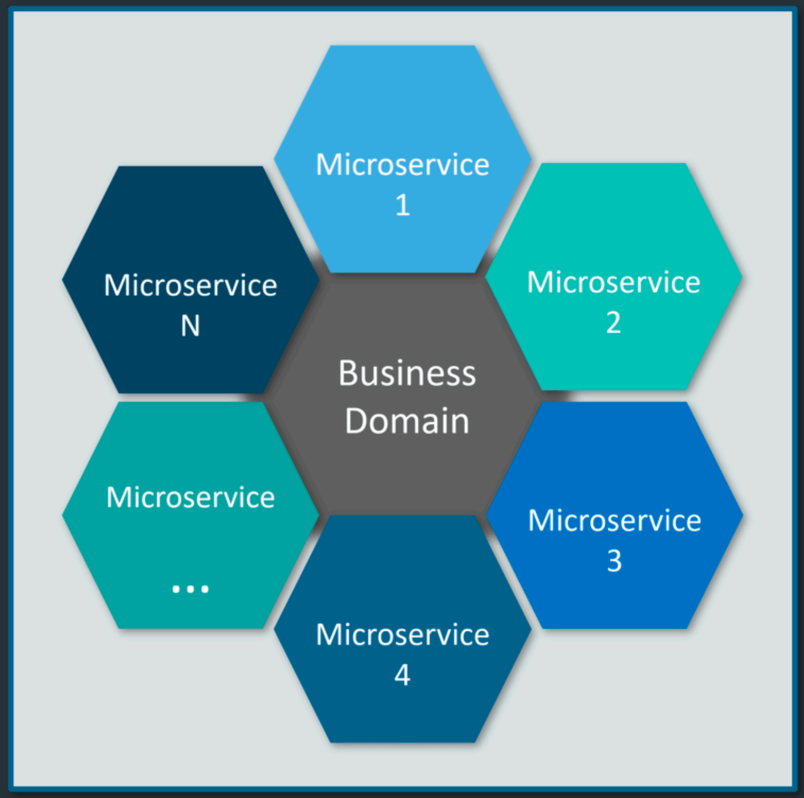
Modeled Around a Business Domain

- Identify the business domains
- Group related services together
- Use a consistent naming convention



Identify the business domains

- Customer management
- Product management
- Order management
- Shipping and fulfillment



Group related services together

- Create a customer account
- Update customer information
- Delete a customer account

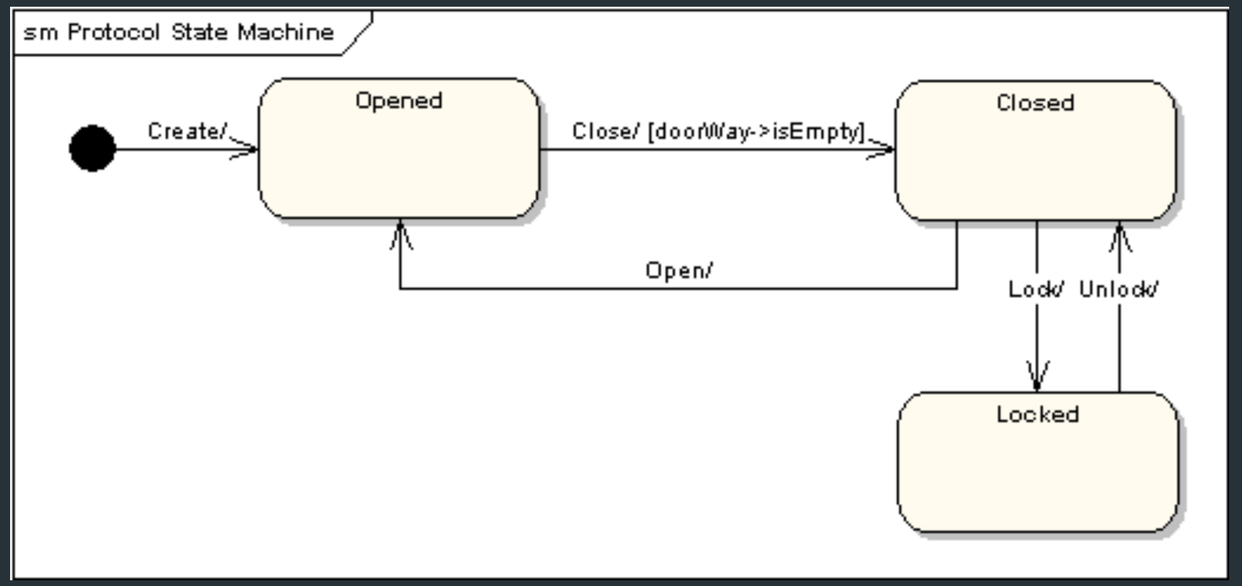
Use a consistent naming convention

Naming conventions in Java

NAMING CONVENTIONS	APPLICATION	EXAMPLES
Lower Camel Case	variables and methods	firstName timeToFirstLoad indexNumber
Upper Camel Case	classes, interfaces, annotations, enums, records	TomcatServer RestController WriteOperation
Screaming Snake Case	constants	INTEREST_RATE MINIMUM_SALARY EXTRA_SAUCE
lower dot case	packages and property files	java.net.http java.management.rmi application.properties
kebab case	not recommended	landing-page.html game-results.jsp 404-error-page.jsf

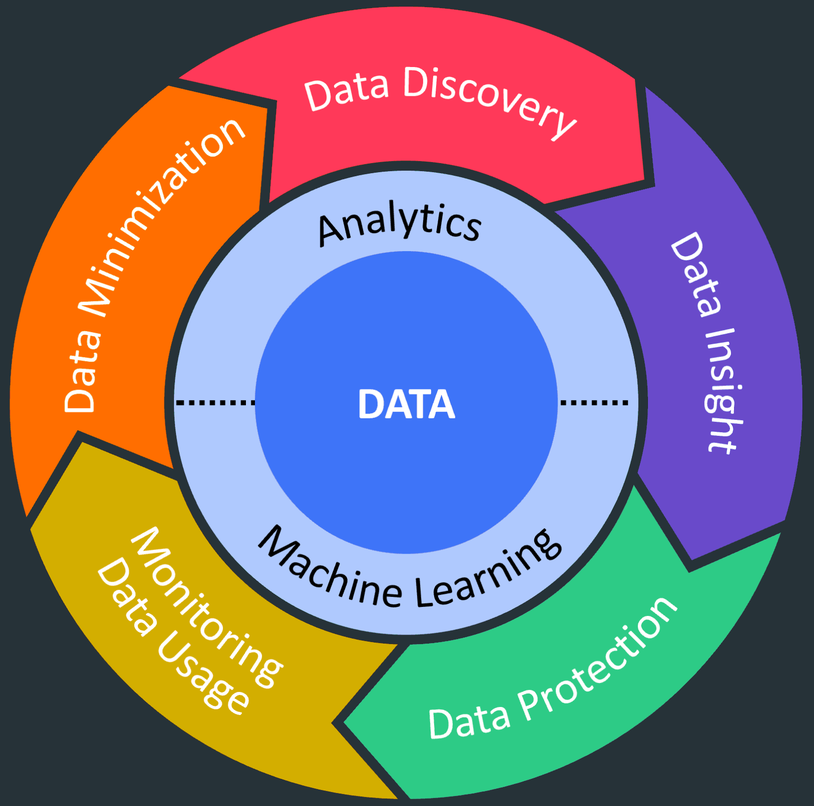
Microservices should own their own state

- Each microservice should be responsible for managing its own data. This can help to improve the scalability and resilience of the application.



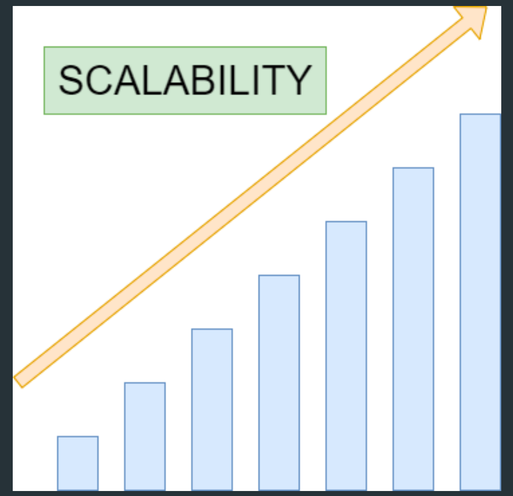
Responsible for managing its own data

- Each microservice should have its own unique identifier, and each piece of data should be stored in a way that is specific to that microservice.



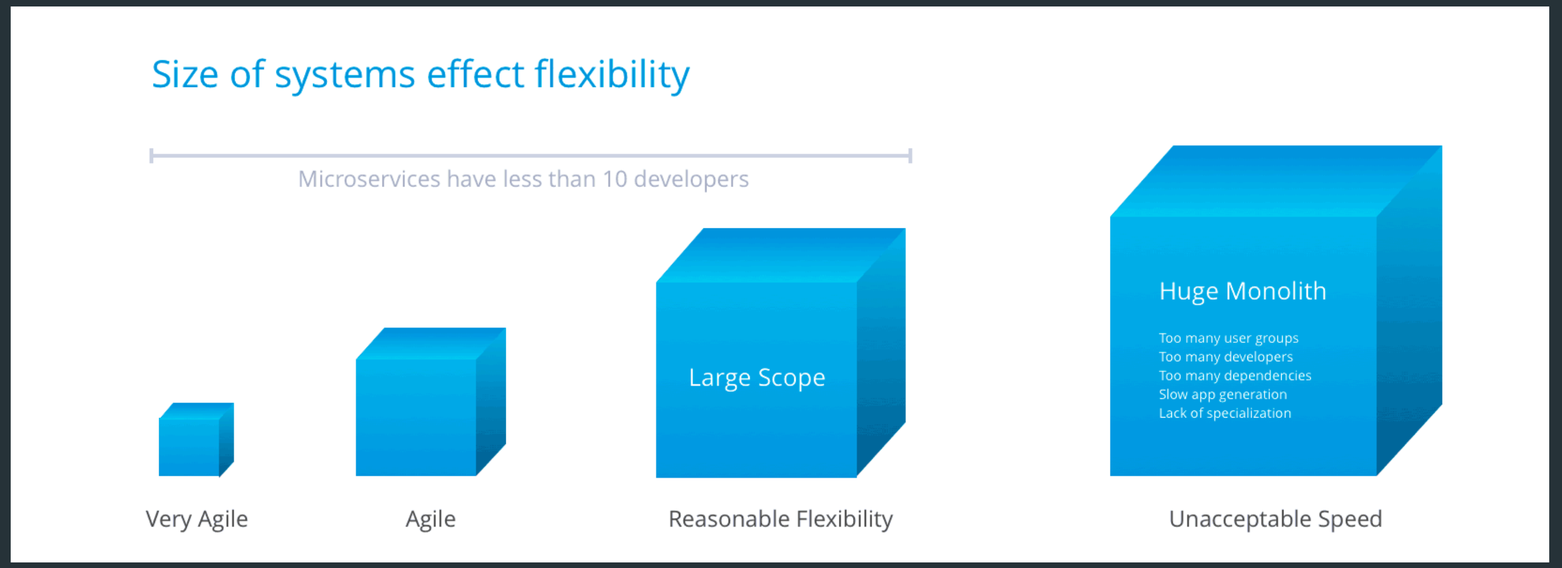
Scalability and Resilience

- By owning their own state, microservices can be scaled independently of each other. This means that if one microservice becomes overloaded, it can be scaled up or down without affecting the other microservices.



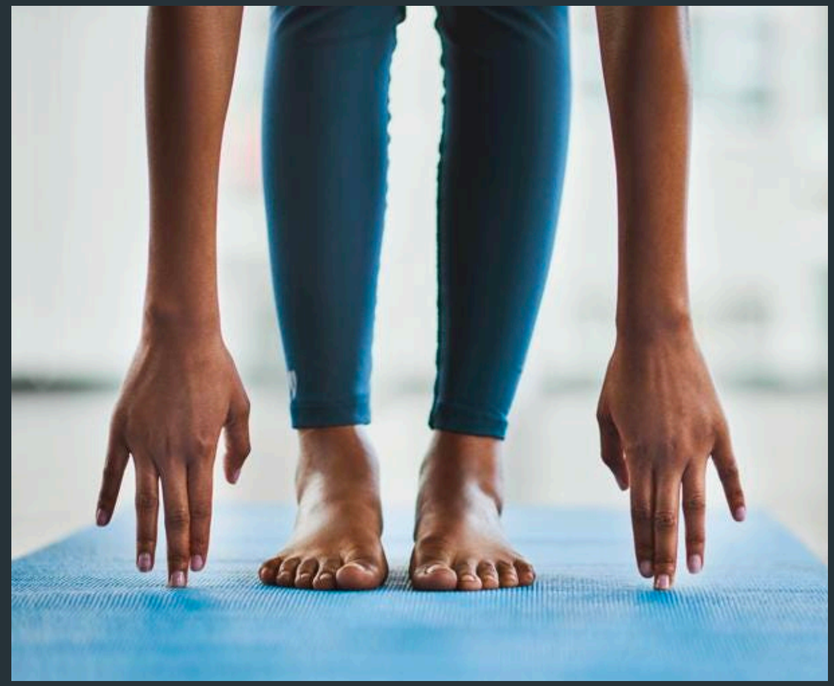
Size

- Microservices should be small
- This means that they should be focused on a single business domain or function
- This can help to improve the scalability, agility, and resilience of the application



Flexibility

- Microservices are flexible
- This means that they can be easily adapted to changes in requirements
- This can help to reduce the cost of ownership



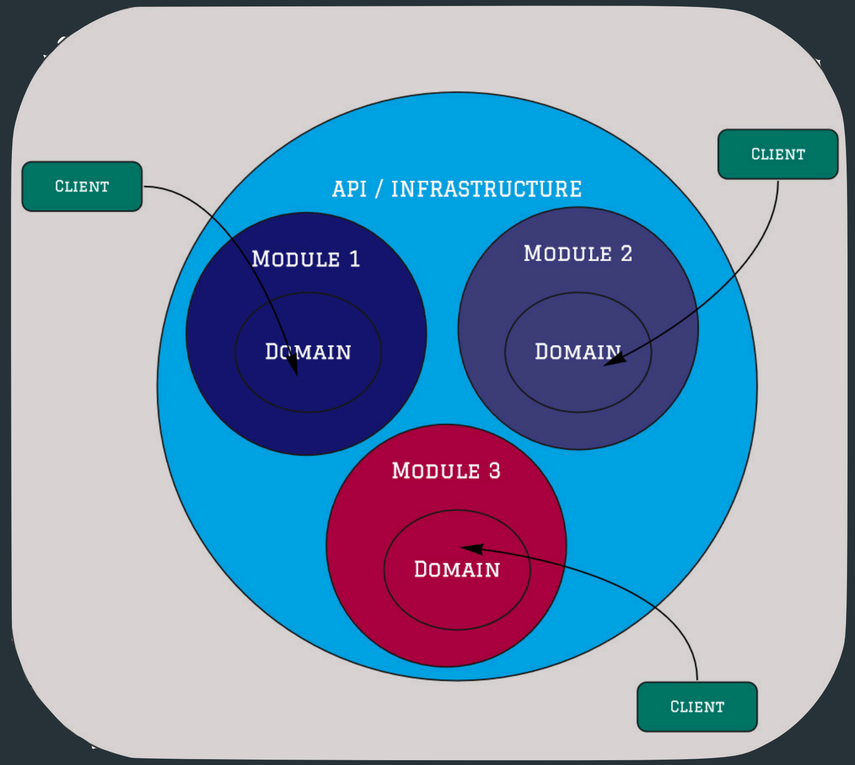
Monolith

The Single Process Monolith

- A single process monolith is a software application that is developed as a single, large codebase.
- The entire application is deployed as a single unit.
- Single process monoliths are easy to develop and deploy, but they can be difficult to scale and maintain.

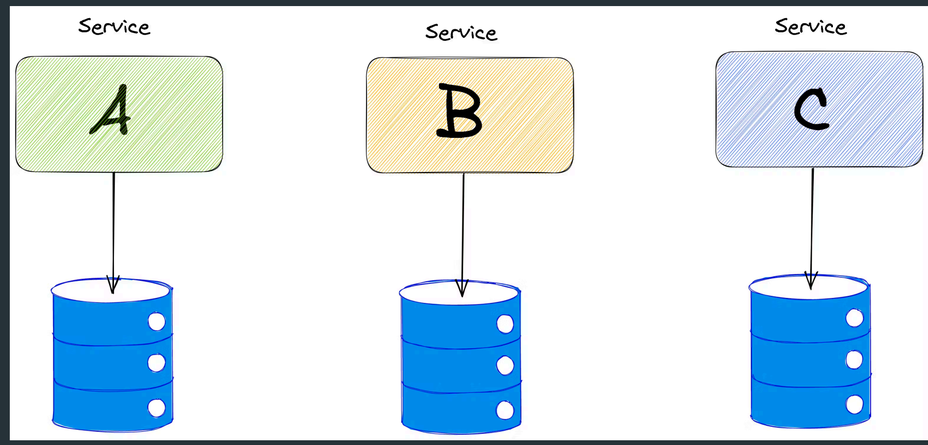
The Modular Monolith

- A modular monolith is a software application that is developed as a single, large codebase, but is divided into smaller, independent modules.
- Each module can be developed, deployed, and scaled independently of the other modules.
- Modular monoliths are a good compromise between the simplicity of a single process monolith and the scalability and flexibility of microservices.



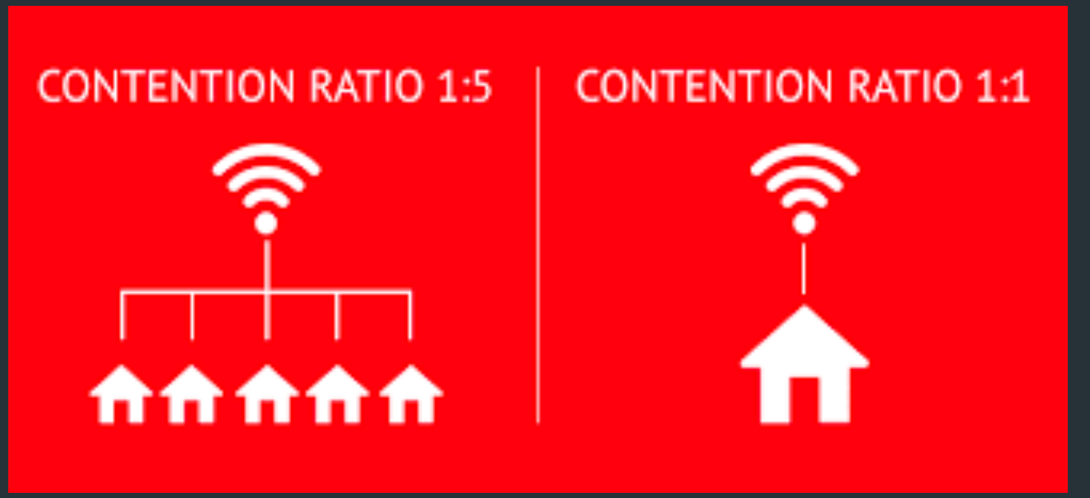
The Distributed Monolith

- A distributed monolith is a software application that is developed as a single, large codebase, but is deployed across multiple servers or machines.
- Each server or machine in a distributed monolith runs a copy of the entire application.
- Distributed monoliths are a good choice for applications that need to be scalable and available, but do not need to be as flexible as microservices.



Monoliths and Delivery Contention

- Monoliths are a single codebase, which means that multiple teams can be working on it at the same time.
- This can lead to delivery contention, where teams are competing for access to the codebase.
- Delivery contention can lead to delays in development and deployment, as well as quality issues.



Advantages of Monoliths

- Simplicity
- Efficiency
- Cost-effectiveness

Monoliths vs. Microservices. Main features

Monoliths	Microservices
Simple deployment	Scalability
Limited agility	Error isolation
Low complexity	High complexity
Limitation of technologies	Flexibility
Simple testing	Loosely coupled system

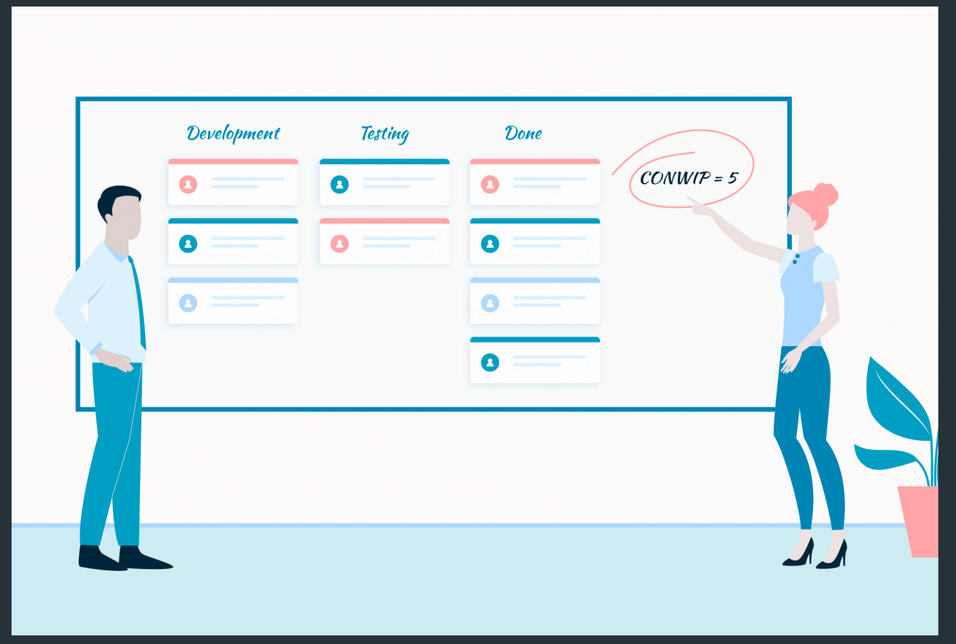
Splitting the Monolith

Splitting the Monolith

- Identify the services
- Design the services
- Migrate the services

Identify the services

- Here are some questions that can be used to identify services:
  - What are the different business capabilities of the application?
  - What are the different functional areas of the application?
  - What are the different teams that are responsible for developing and maintaining the application?



Design the services

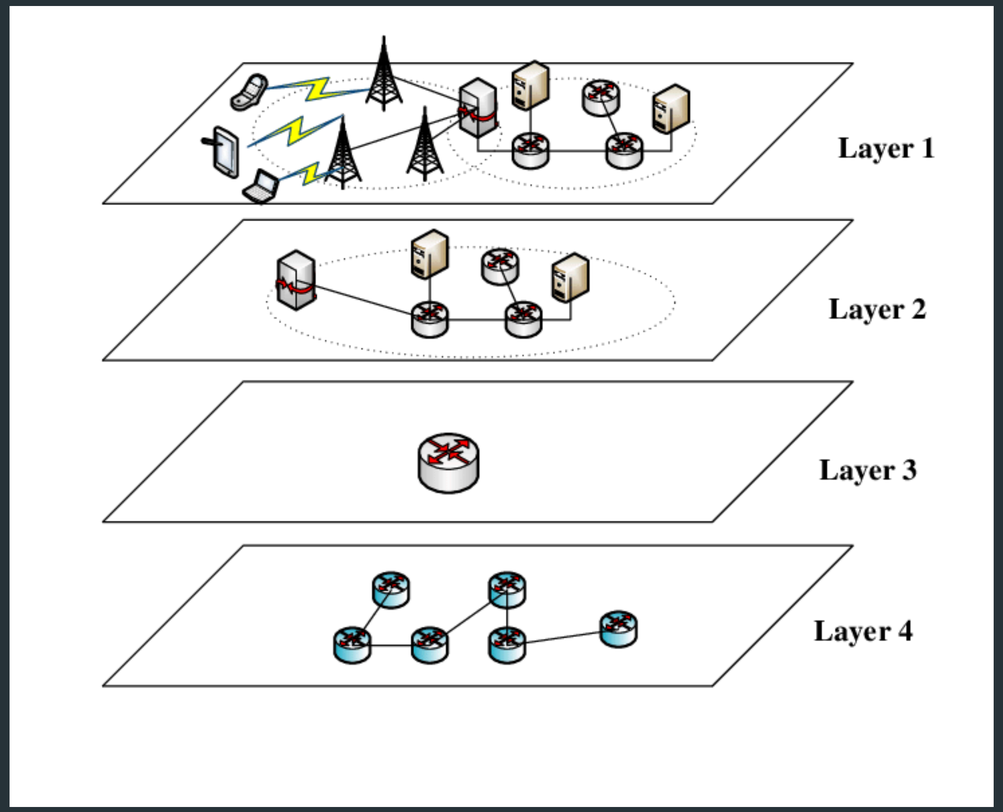
- Here are some factors to consider when designing the services:
  - Cohesion
  - Avoid coupling
  - Consider the scalability

Migrate the services

- Here are some factors to consider when migrating the services:
  - Plan the migration carefully
  - Test the migration thoroughly
  - Communicate with the users

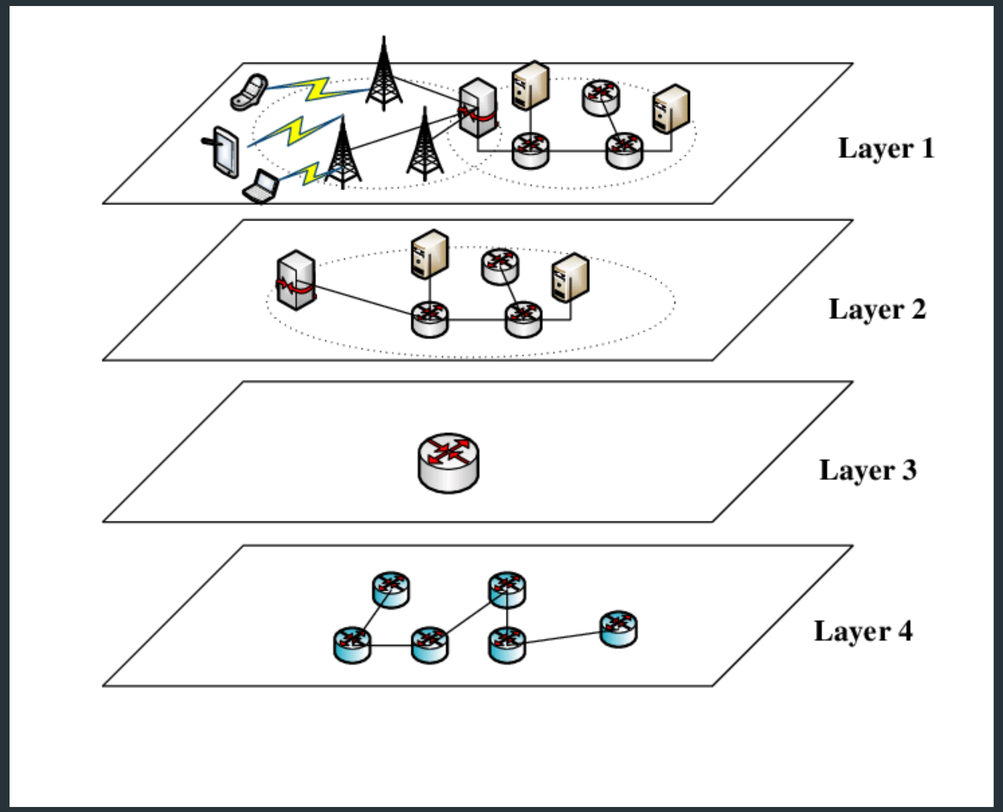
Decomposition by Layer

- Identify the layers
- Decompose each layer
- Refactor the code



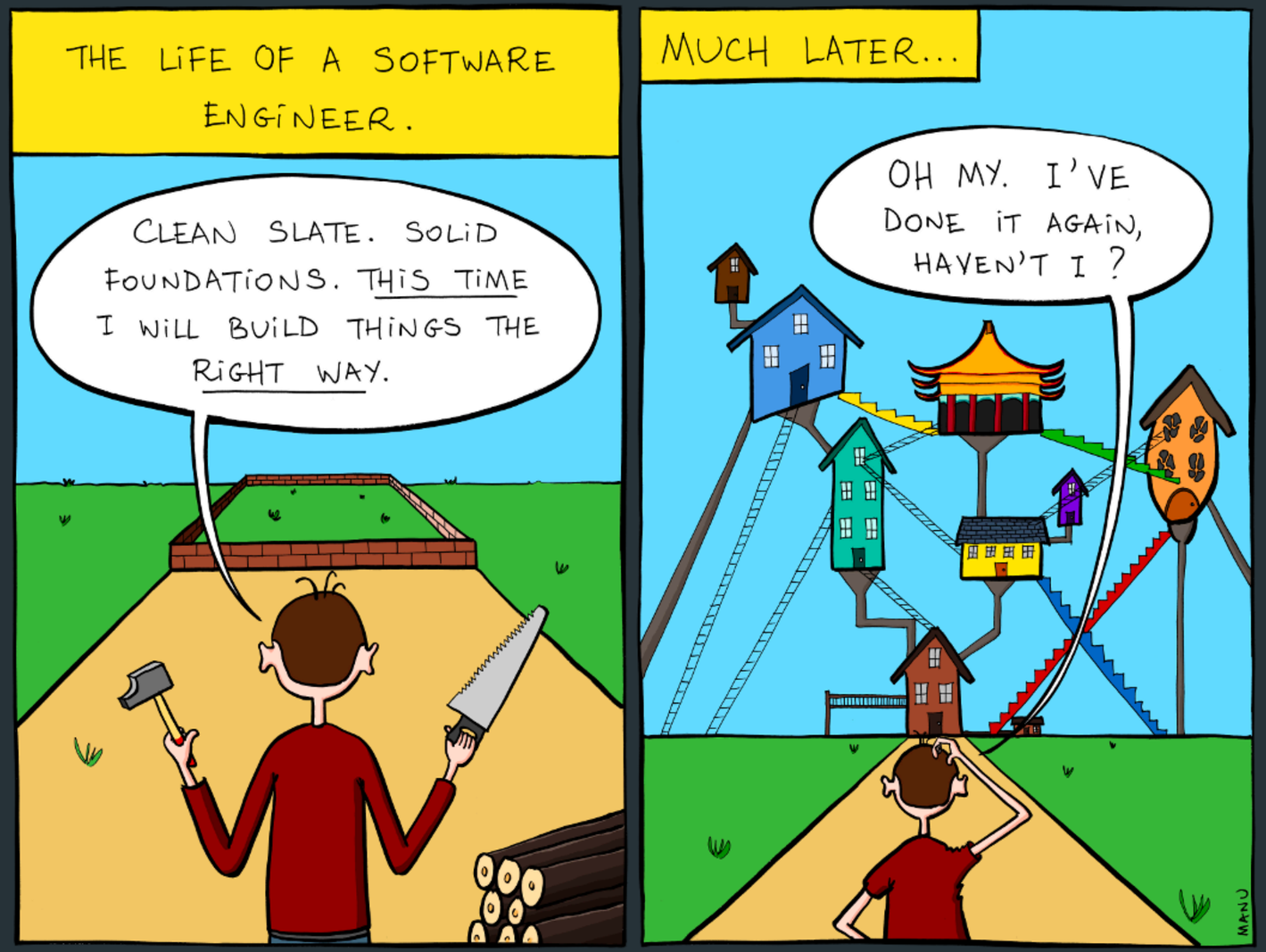
Identify the layers - Questions

- Here are some questions that can be used to identify layers:
  - What are the different logical functions of the application?
  - What are the different types of data that the application uses?
  - What are the different types of users of the application?



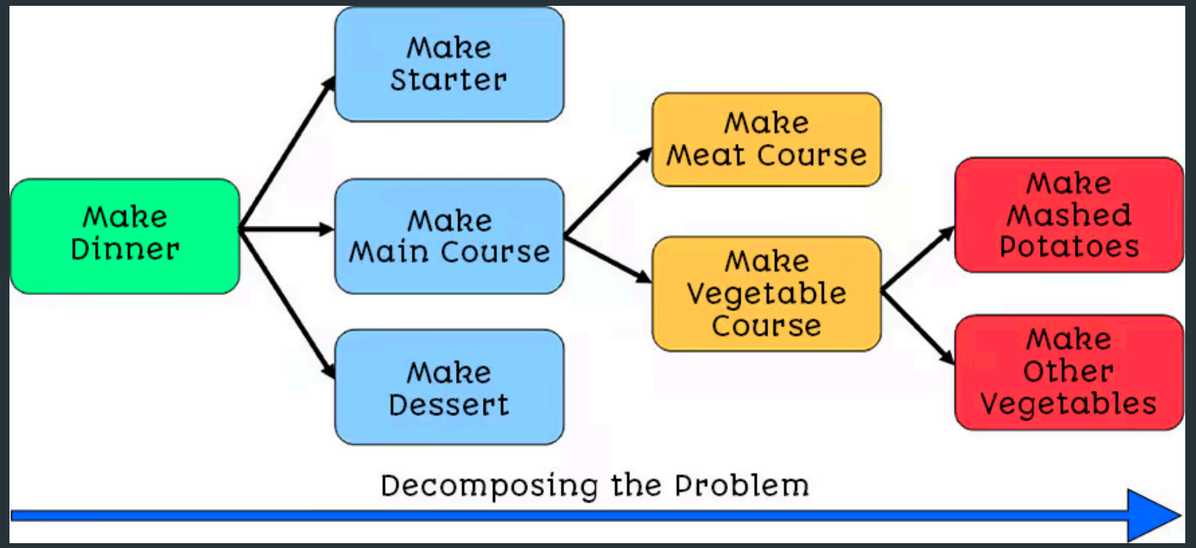
Refactor the code

- Use a consistent coding style
- Use automated tools
- Test the code



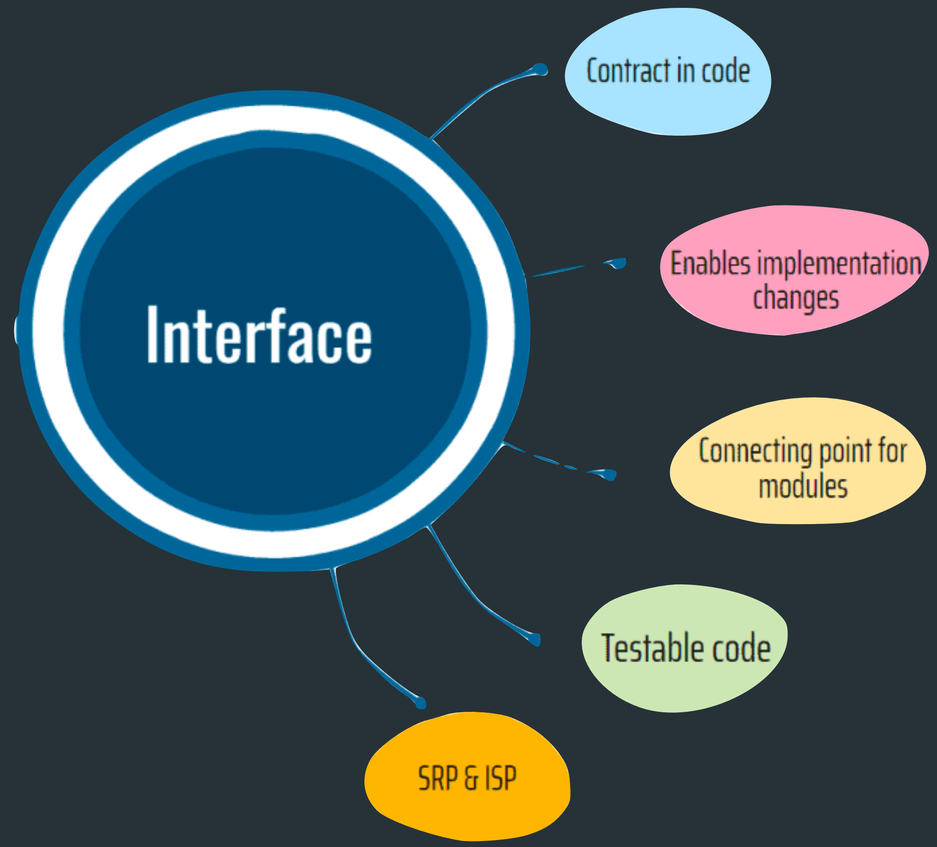
Code First Decomposition

- Define the interfaces
- Implement the interfaces
- Test the services



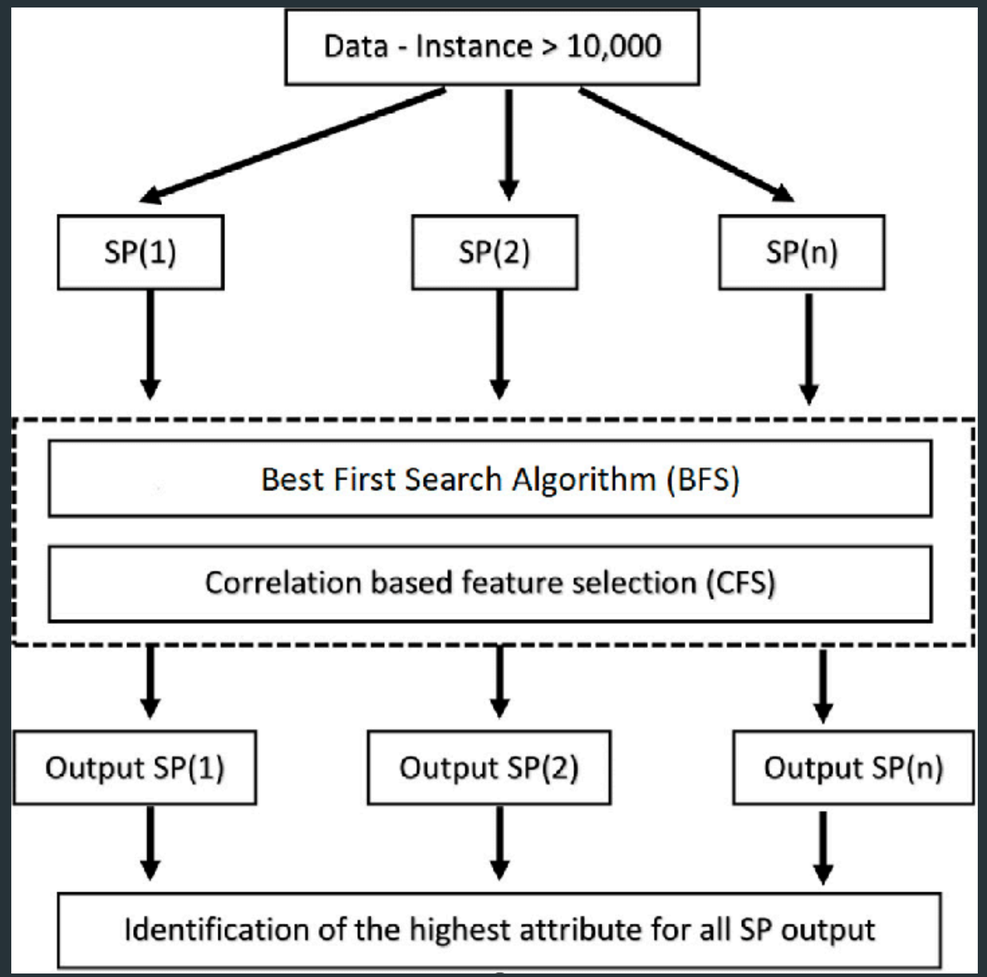
Define the interfaces

- What are the different operations that the service will need to perform?
- What are the different types of data that the service will need to use?
- What are the different security requirements for the service?



Data First Decomposition

- Identify the data
- Design the services
- Migrate the data

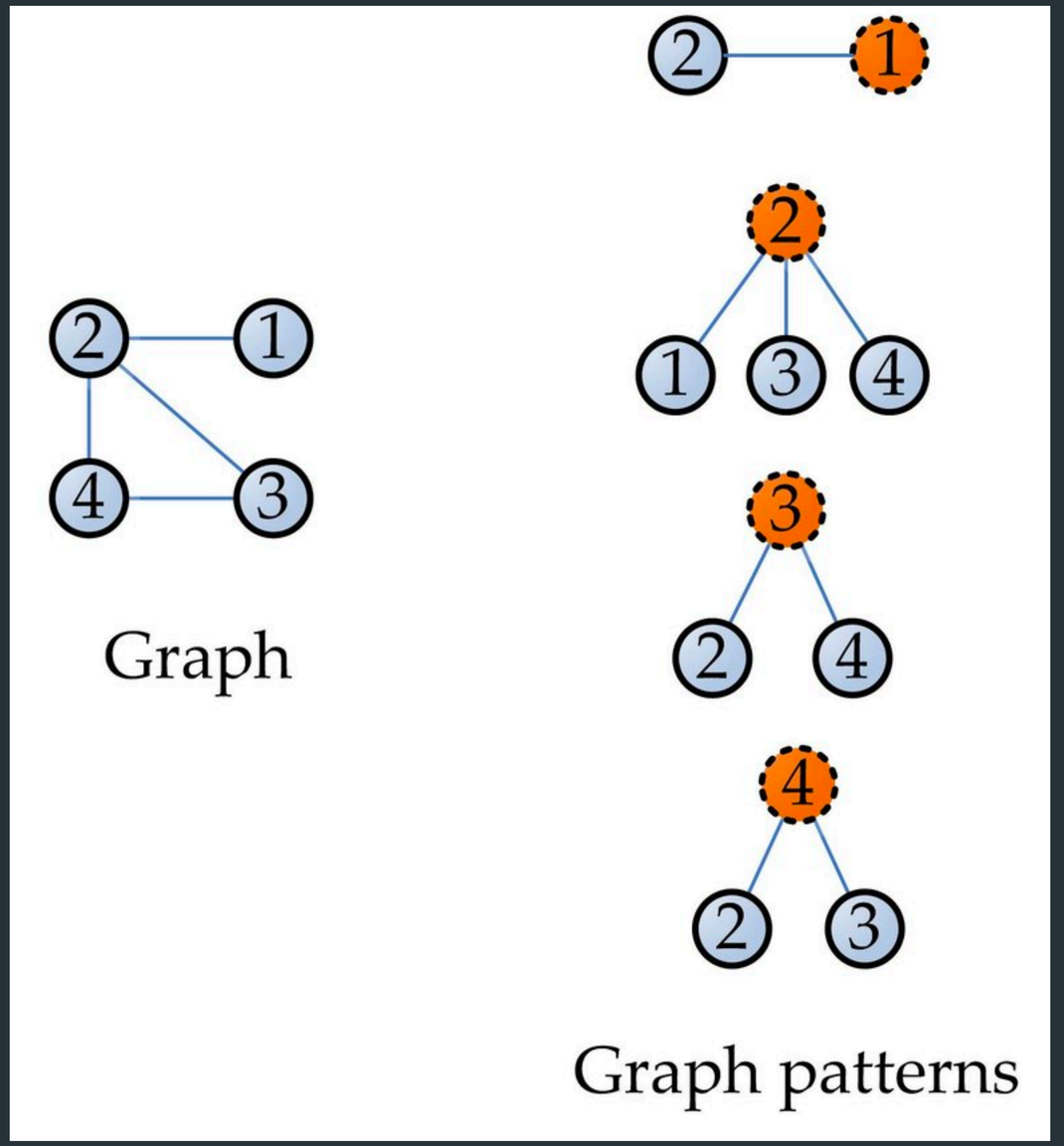


Identify the data

- What are the different types of data that the application uses?
- What are the different relationships between the different data types?
- What are the different sources of data?
- What are the different sinks for data?

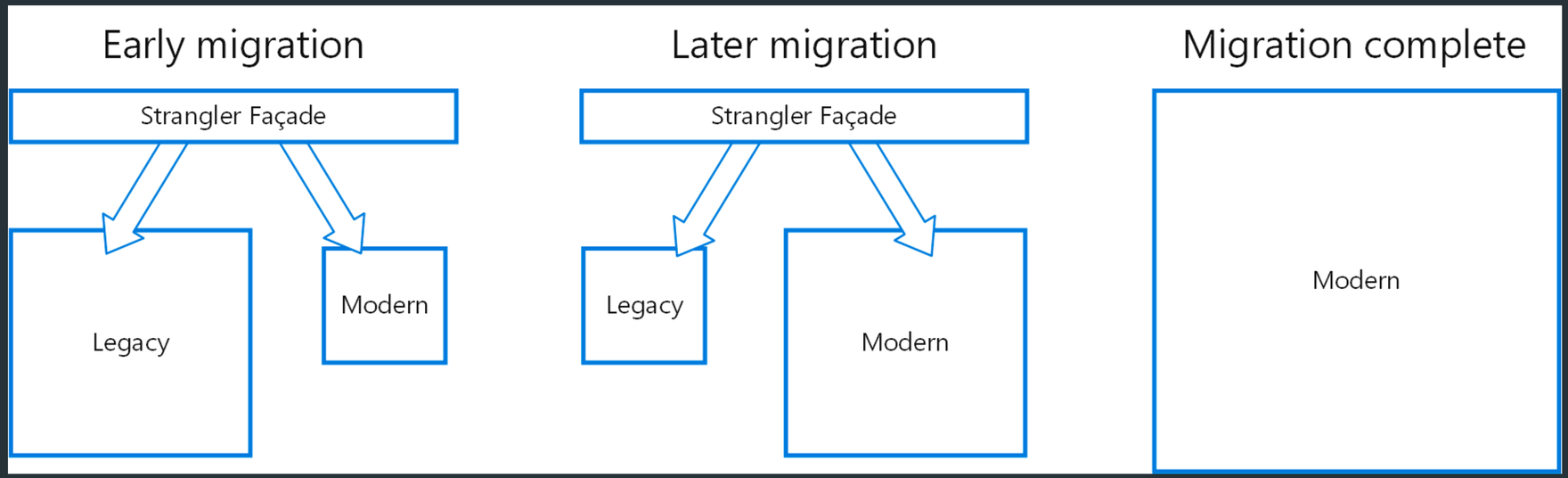
Useful Decompositional Patterns

- Functional decomposition
- Data decomposition
- Process decomposition



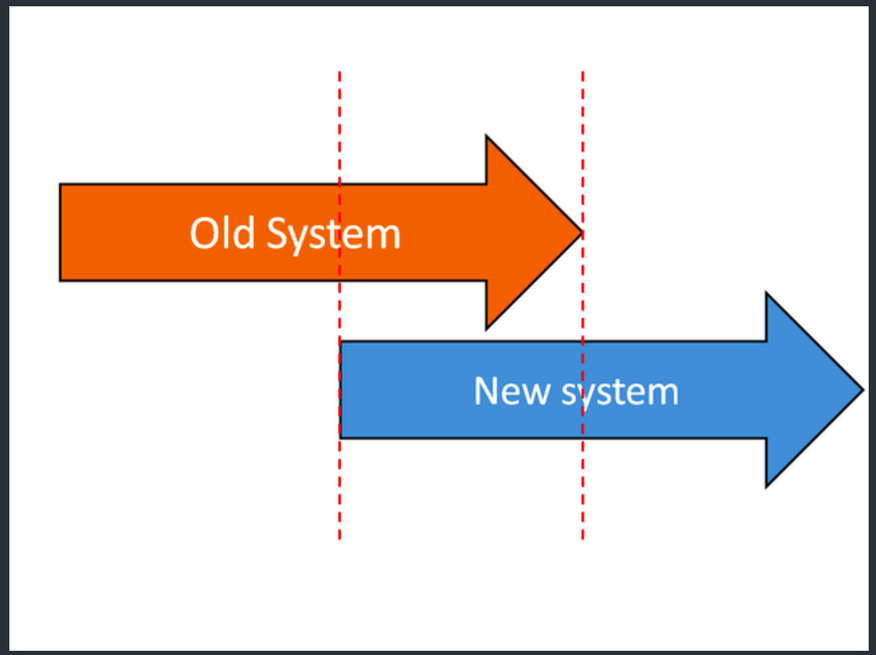
Strangler Fig Pattern

- Add a new service to replace an existing feature in the monolith
- Gradually shift traffic to the new service
- Retire the old service when it is no longer needed



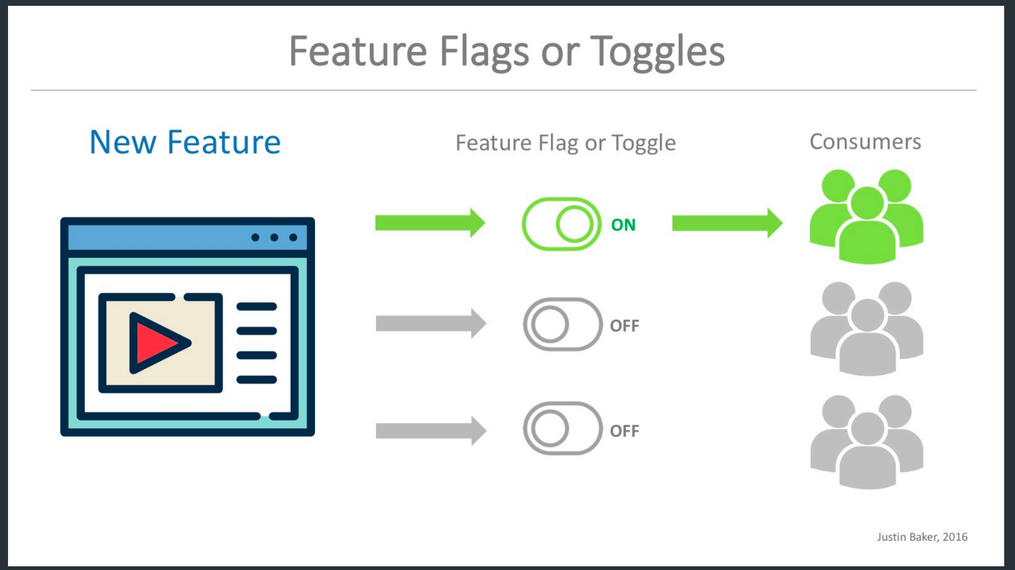
Parallel Run Pattern

- Run both the old and new implementations in parallel.
- Compare the results to ensure they are equivalent.
- Only one implementation is considered the source of truth at any given time.



Feature Toggle Pattern

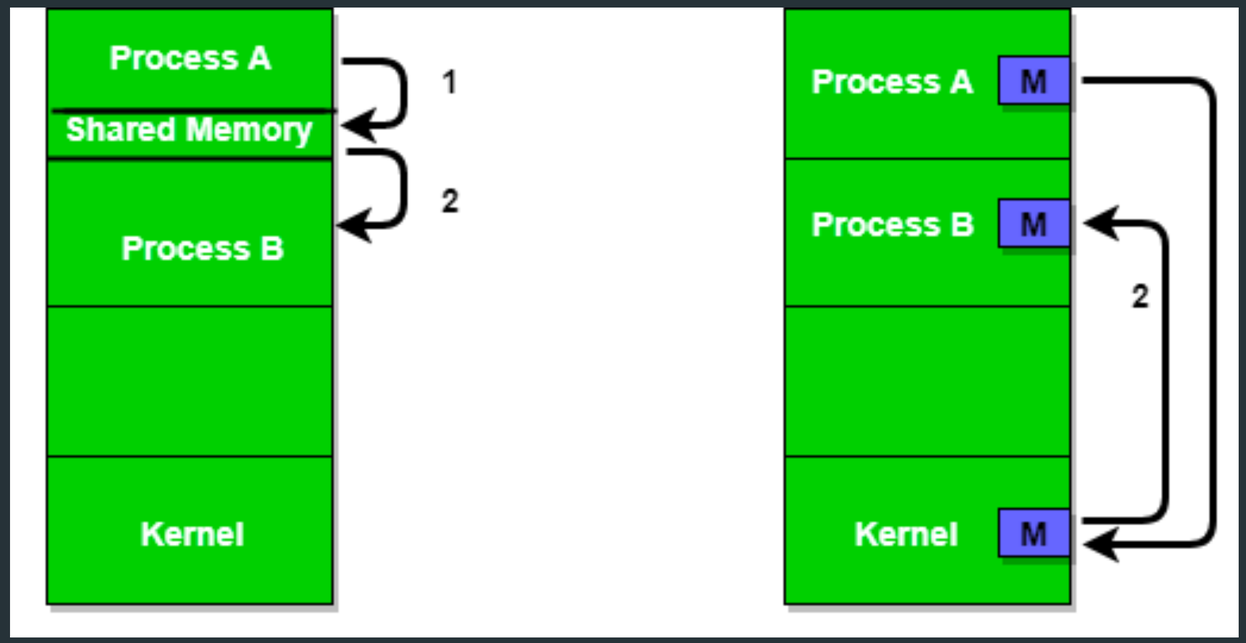
- Create a feature toggle to enable or disable the new implementation.
- Use the feature toggle to control the rollout of the new implementation.
- Monitor the results of the rollout and make adjustments as needed.



Communication Styles

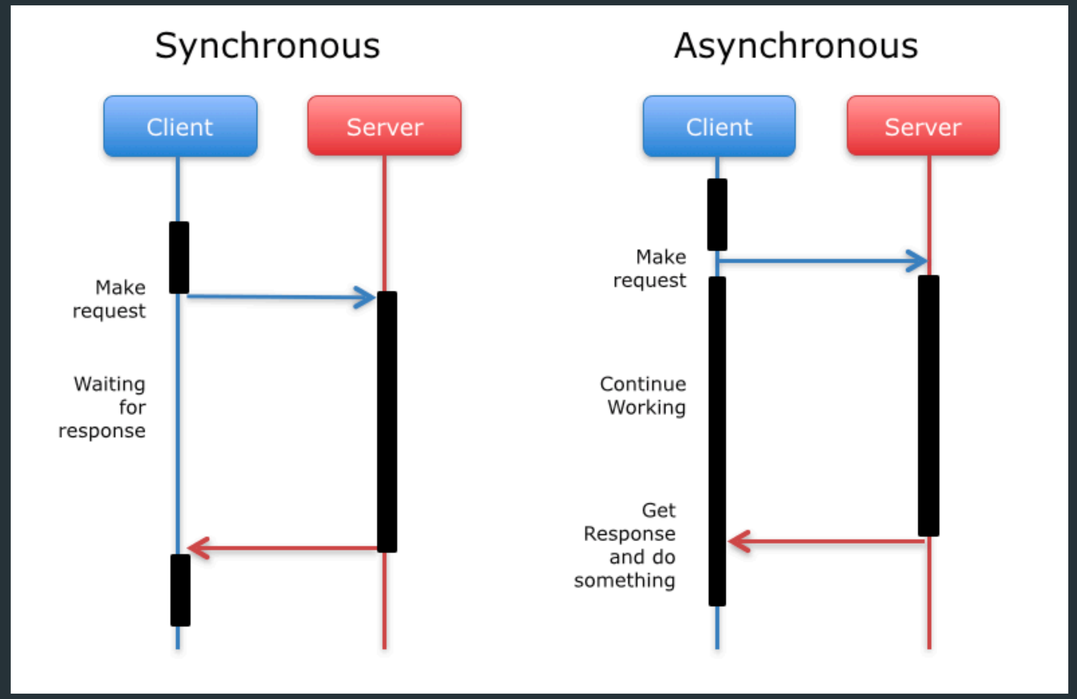
From In-Process to Inter-Process

- In-process communication:
  - Pros: Fast, efficient, and easy to use
  - Cons: Difficult to scale, not fault-tolerant, and not secure
- Inter-process communication:
  - Pros: Scalable, fault-tolerant, and secure
  - Cons: Slower, more complex, and more difficult to use



Synchronous Blocking

- Pros:
  - Simple and easy to implement
  - Provides strong guarantees about the order and consistency of messages
- Cons:
  - Inefficient and can lead to performance bottlenecks
  - Not scalable or fault-tolerant

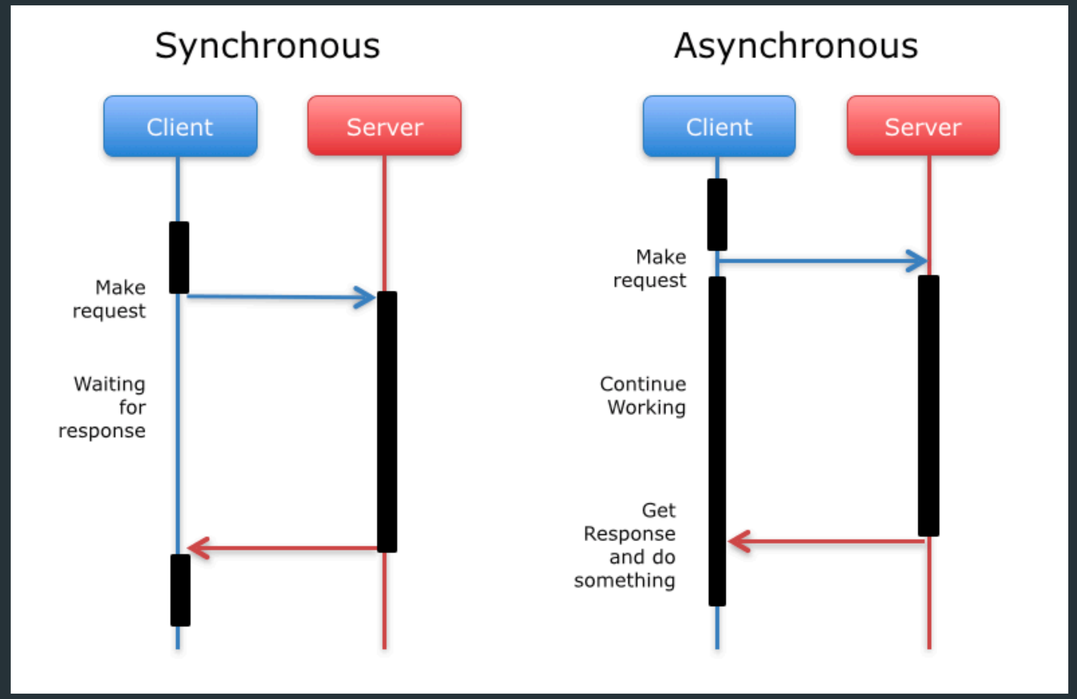


Synchronous Blocking

```
// In the first microservice

// Make a request to the second microservice

response = await secondMicroservice.call();

// Use the response

// In the second microservice

// Receive the request from the first microservice

response = processRequest(request);

// Send the response back to the first microservice

await firstMicroservice.reply(response);
```

Asynchronous Nonblocking

- Pros:
  - Scalable and fault-tolerant
  - Efficient and can improve performance
- Cons:
  - More complex to implement
  - Can be difficult to debug

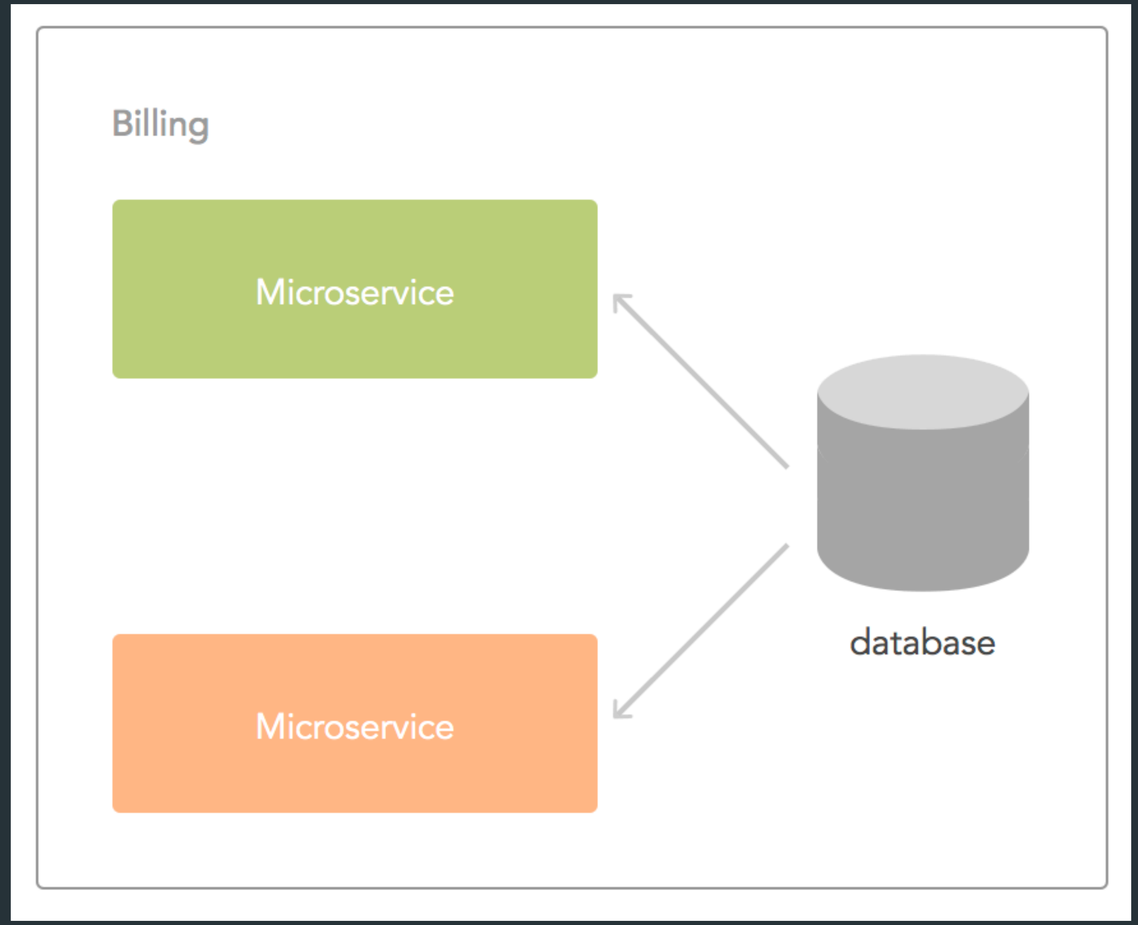


Asynchronous Nonblocking

```
// In the first microservice

// Make a request to the second microservice

secondMicroservice.call(request, (err, response) => {
  if (err) {
    // Handle the error
  } else {
    // Use the response
  }
});

// In the second microservice

// Receive the request from the first microservice

request = processRequest(request);

// Send the response back to the first microservice

secondMicroservice.reply(response);
```

Communication Through Common Data

- Pros:
  - Simple and easy to implement
  - Scalable and fault-tolerant
- Cons:
  - Can be inefficient
  - May not be suitable for all applications

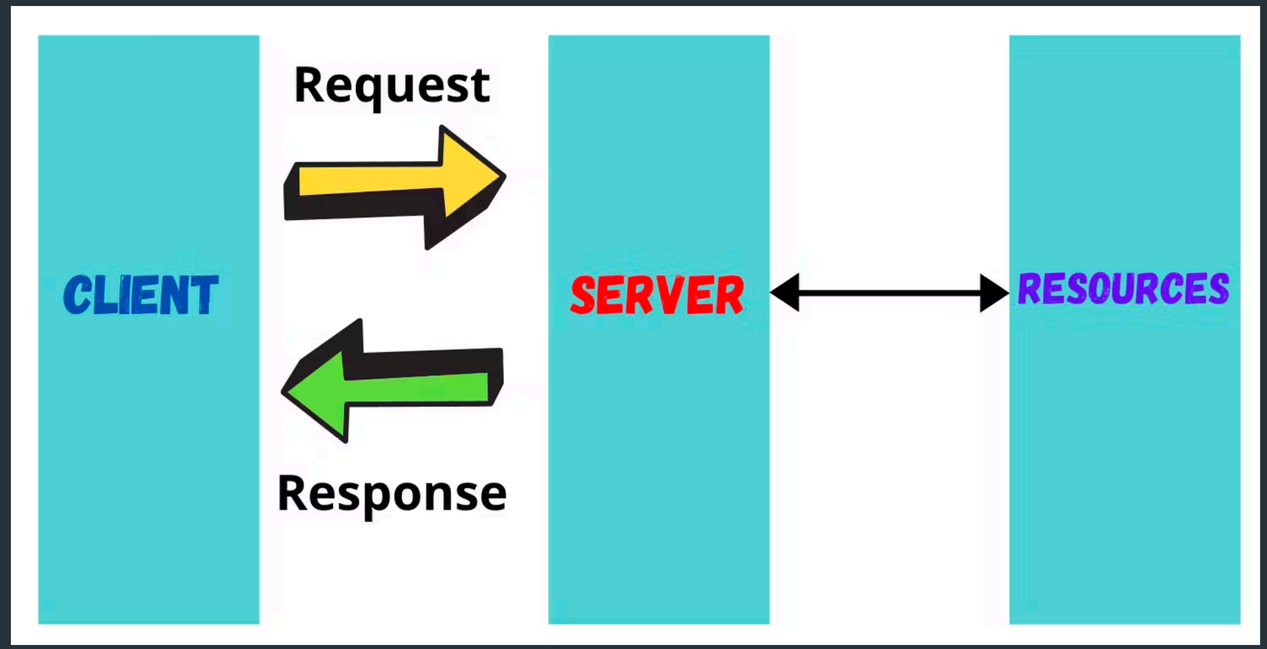


Communication Through Common Data

```
// In the first microservice

// Update the common data store

database.save(new User(username, password));

// In the second microservice

// Poll the common data store for updates

user = database.find(username);

// Use the user
```

Request-Response Communication

- Pros:
  - Simple and easy to implement
  - Provides strong guarantees about the order and consistency of messages
- Cons:
  - Inefficient and can lead to performance bottlenecks
  - Not scalable or fault-tolerant

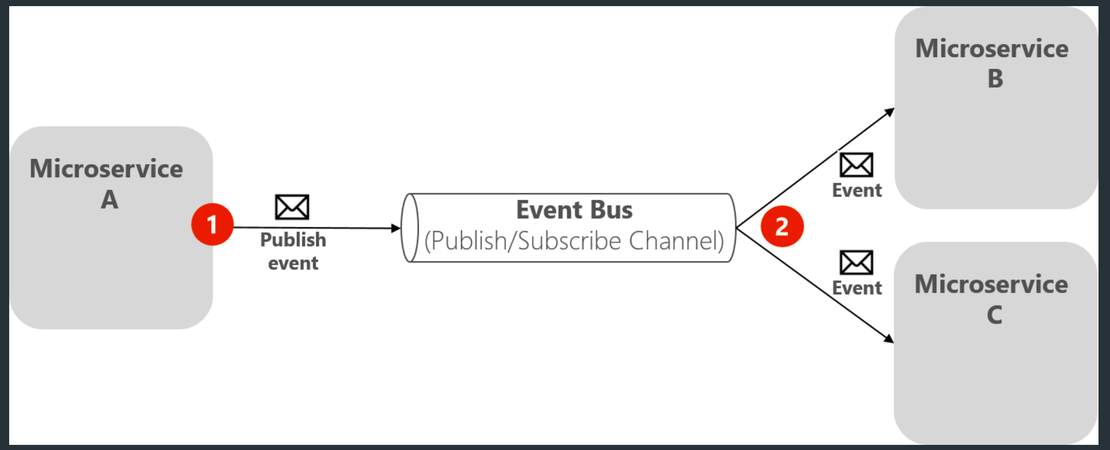


Request-Response Communication

```
// In the first microservice

// Make a request to the second microservice

response = await secondMicroservice.call();

// Use the response

// In the second microservice

// Receive the request from the first microservice

response = processRequest(request);

// Send the response back to the first microservice

await firstMicroservice.reply(response);
```

Event-Driven Communication

- Pros:
  - Simple and easy to implement
  - Provides strong guarantees about the order and consistency of messages
- Cons:
  - Inefficient and can lead to performance bottlenecks
  - Not scalable or fault-tolerant

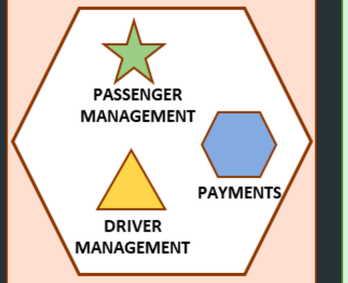


Event-Driven Communication

```
// In the first microservice

// Publish an event

eventBus.publish("order-placed", order);

// In the second microservice

// Subscribe to the event

eventBus.on("order-placed", (order) => {
  // Process the order
});
```

Implementing Microservices Communication: Technology Choices

Monolith



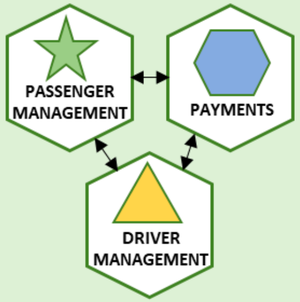
Benefits

Simple to Develop.
Simple to Deploy.
Simple Testing.

Challenges

Hard to scale.
Too Large/Complex.
Redeploy the entire application on each update.

Microservices



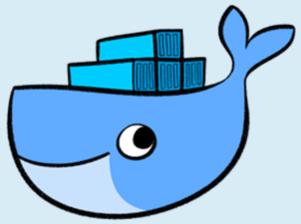
Benefits

Better Scaling.
New technologies.
Frequent Releases.

Challenges

Slower Deployment.
Service Discovery.
Complex Configs.
Load Balancing.
Central Logging.

Docker



Benefits

Faster Deployment.
Faster start time.
Lightweight Image.

Challenges

No Storage Option.
No Autoscaling.
No health-checks.
Networking is hard.

Kubernetes

Benefits

Storage options.
Auto-scaling.
Self-healing.

Challenges

Hard to install.
No Building Image.
No built in Jenkins.
Separate install for Dashboard.

Openshift

Benefits

OOTB Webconsole.
OOTB Build Images.
Built in Jenkins.
Build Images and store in Registry.

Challenges

Requires additional API Management.
Limited installation options.

Istio

Benefits

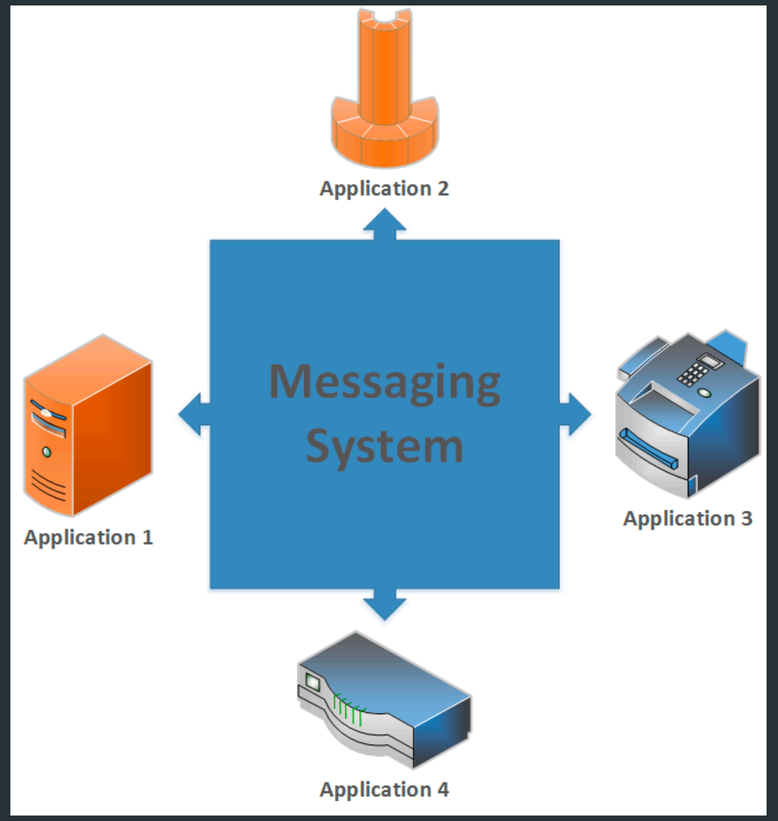
Tracing.
Circuit Breaker.
Canary Release.
Dark Launches.
Telemetry.

Challenges

Added Complexity.
Adds Overhead.
Required Expertise.

Implementing Microservices Communication: Technology Choices

- Messaging systems
- API gateways
- Direct calls



Popular Messaging Systems

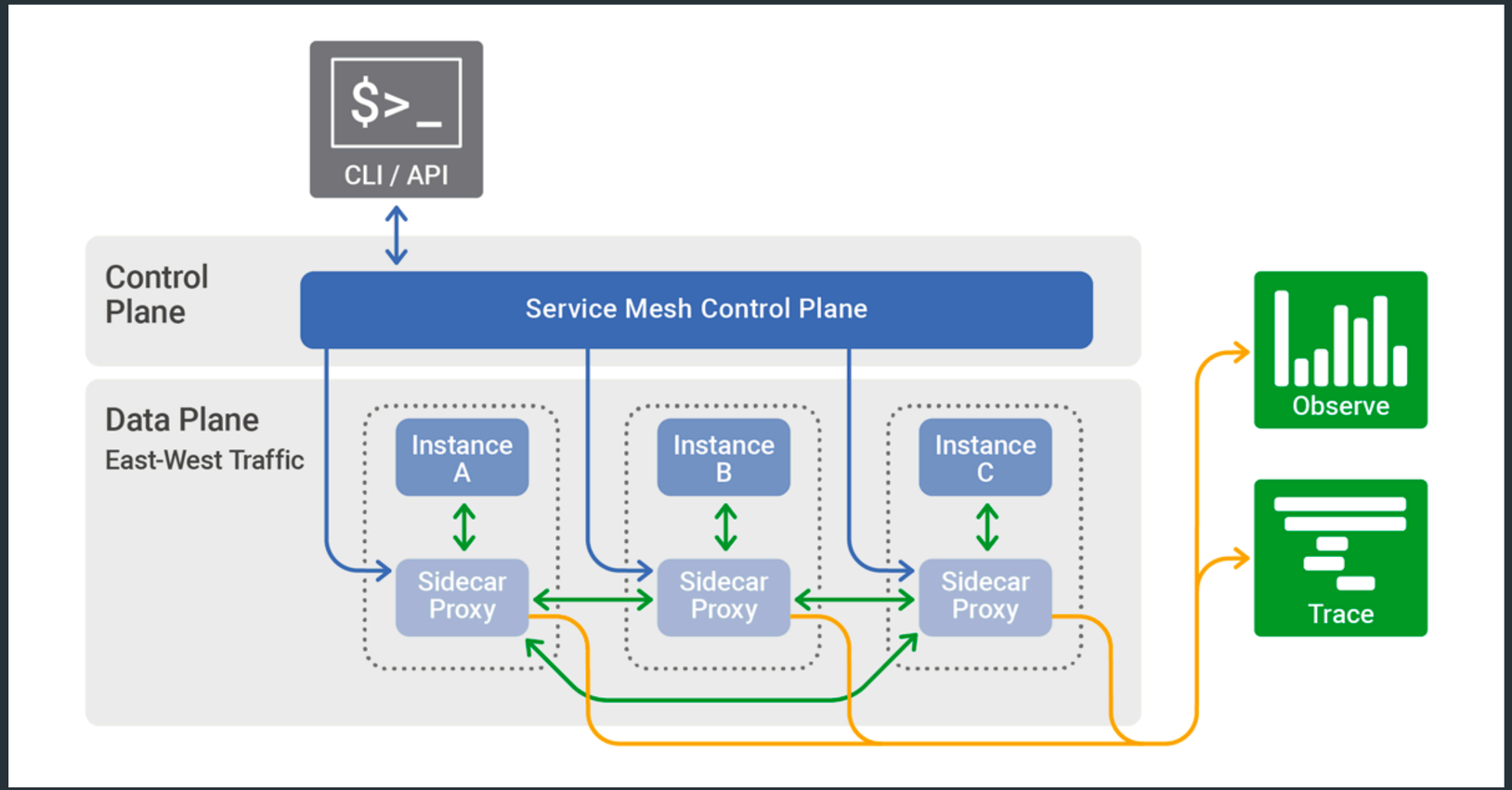
- RabbitMQ
- ActiveMQ
- Kafka

API gateways

- Kong
- Apigee
- Tyk

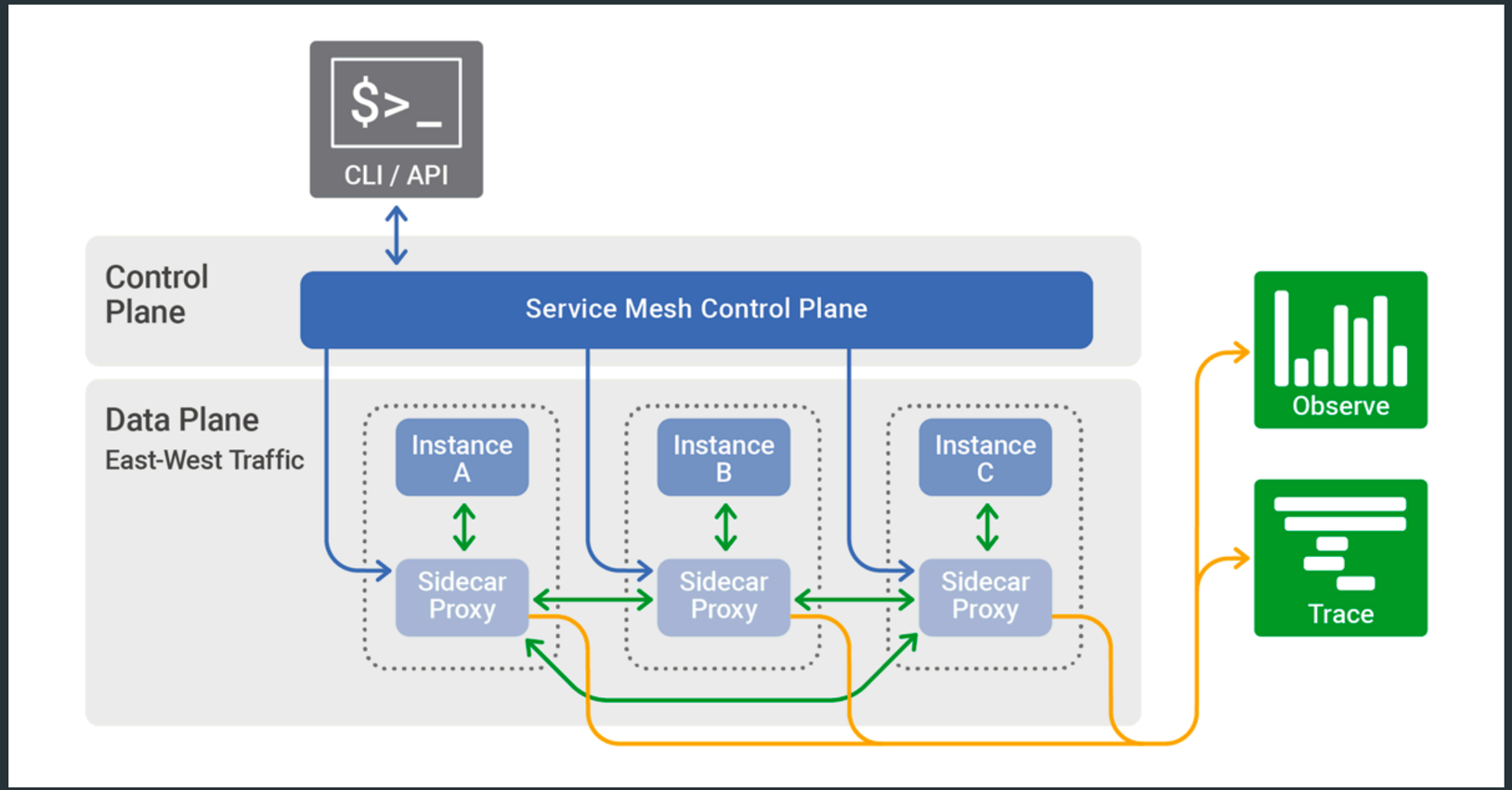
Service Meshes

- What is a service mesh?
  - A service mesh is a layer of software that sits between microservices and handles communication between them.
  - It provides features such as load balancing, retries, and circuit breaking.



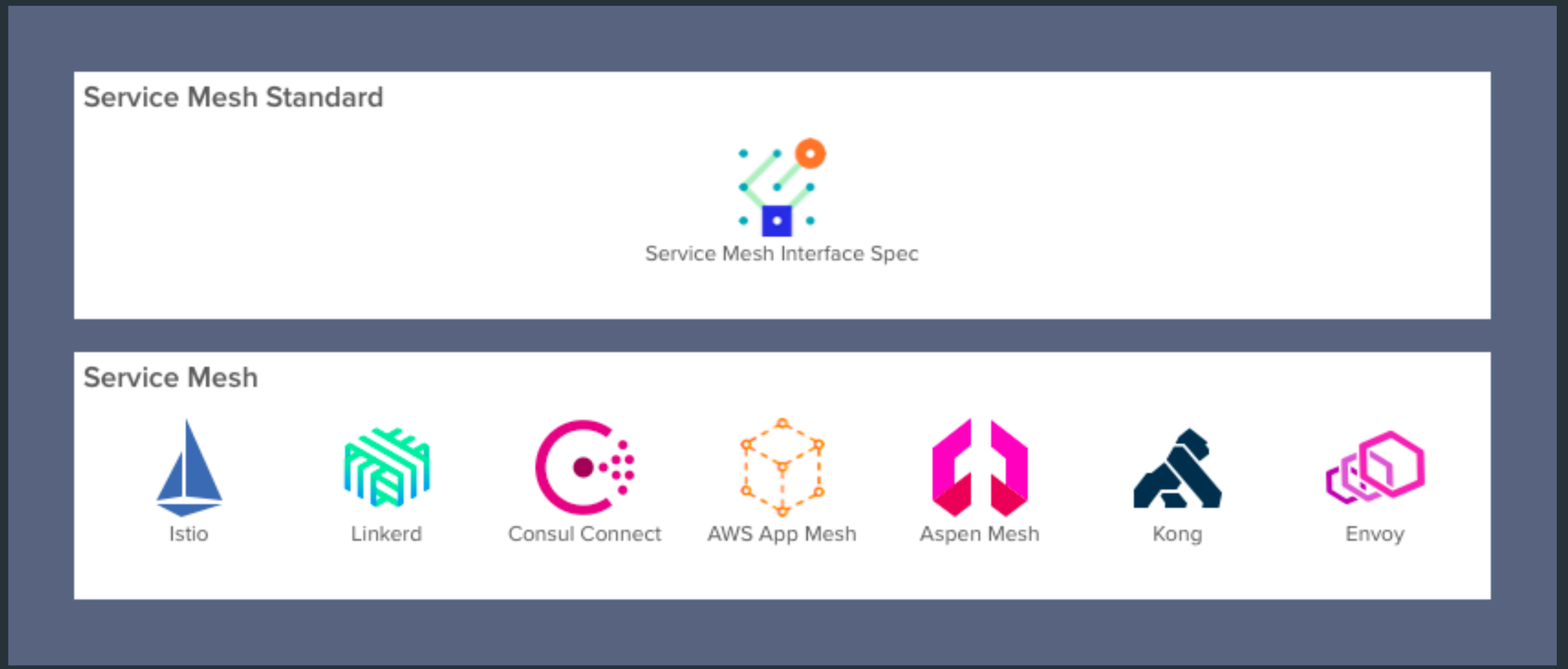
Benefits of using a service mesh

- Improved reliability
- Improved performance
- Reduced complexity
- Increased security



Popular service meshes

- Istio
- Linkerd
- Consul



Communication with GraphQL

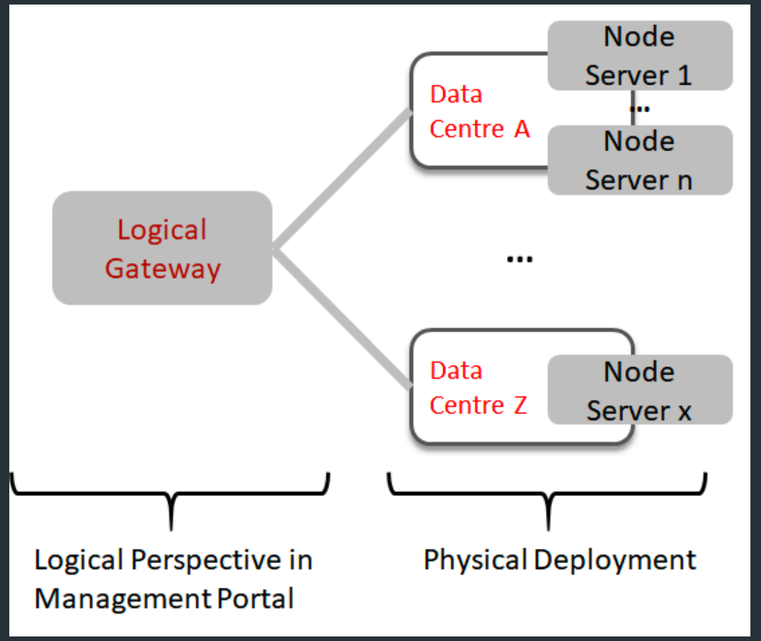
- GraphQL is a query language for APIs that provides a uniform way to access data from different sources.
- GraphQL can be used to implement microservices communication by defining a schema that describes the data that is available from each service.
- GraphQL clients can then use this schema to query the data that they need, without having to know the details of how the data is stored or accessed.

```
query {
  users {
    name
    email
  }
}
```

Deployment

From Logical to Physical

- When deploying microservices, it is important to consider the following factors:
  - Scalability: Your deployment solution should be scalable so that you can easily add or remove microservices as needed.
  - Resilience: Your deployment solution should be resilient so that it can handle failures without affecting the availability of your application.
  - Security: Your deployment solution should be secure so that your microservices are protected from attack.

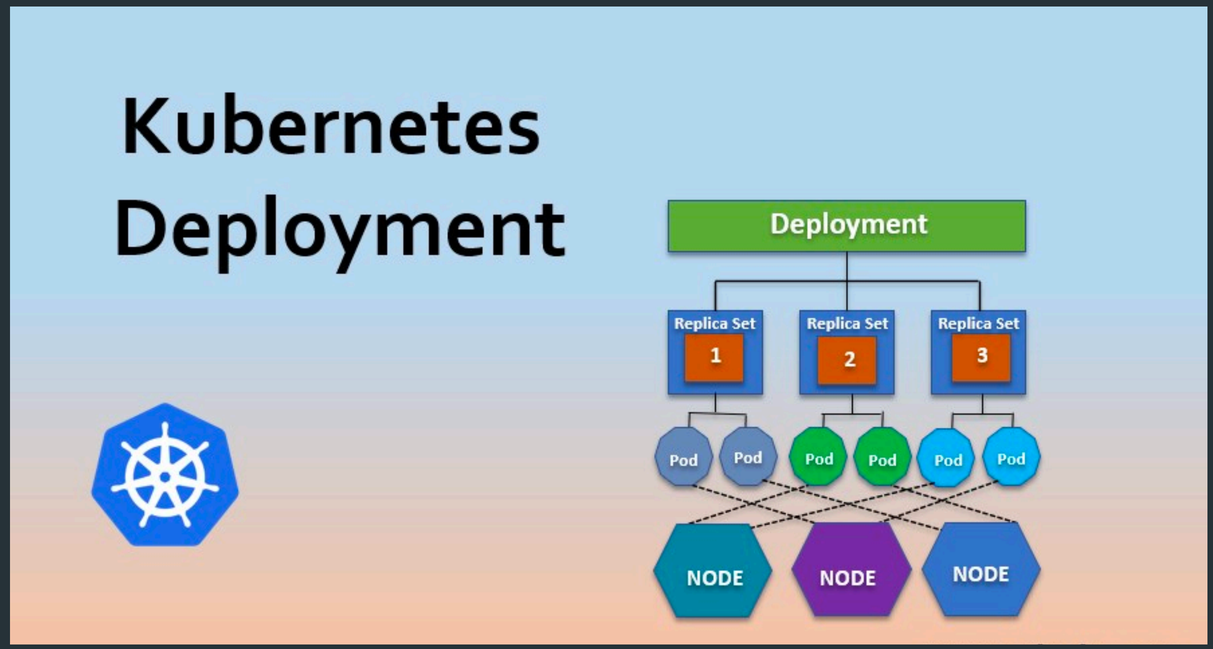


Example

```
#The script first creates a new deployment for the microservice.
kubectl create deployment my-microservice --image=my-microservice:latest

#The script then creates a new service for the microservice.
kubectl create service my-microservice --type=LoadBalancer

#The script finally waits for the microservice to be deployed and ready to use.
kubectl wait --for=condition=ready deployment/my-microservice
```

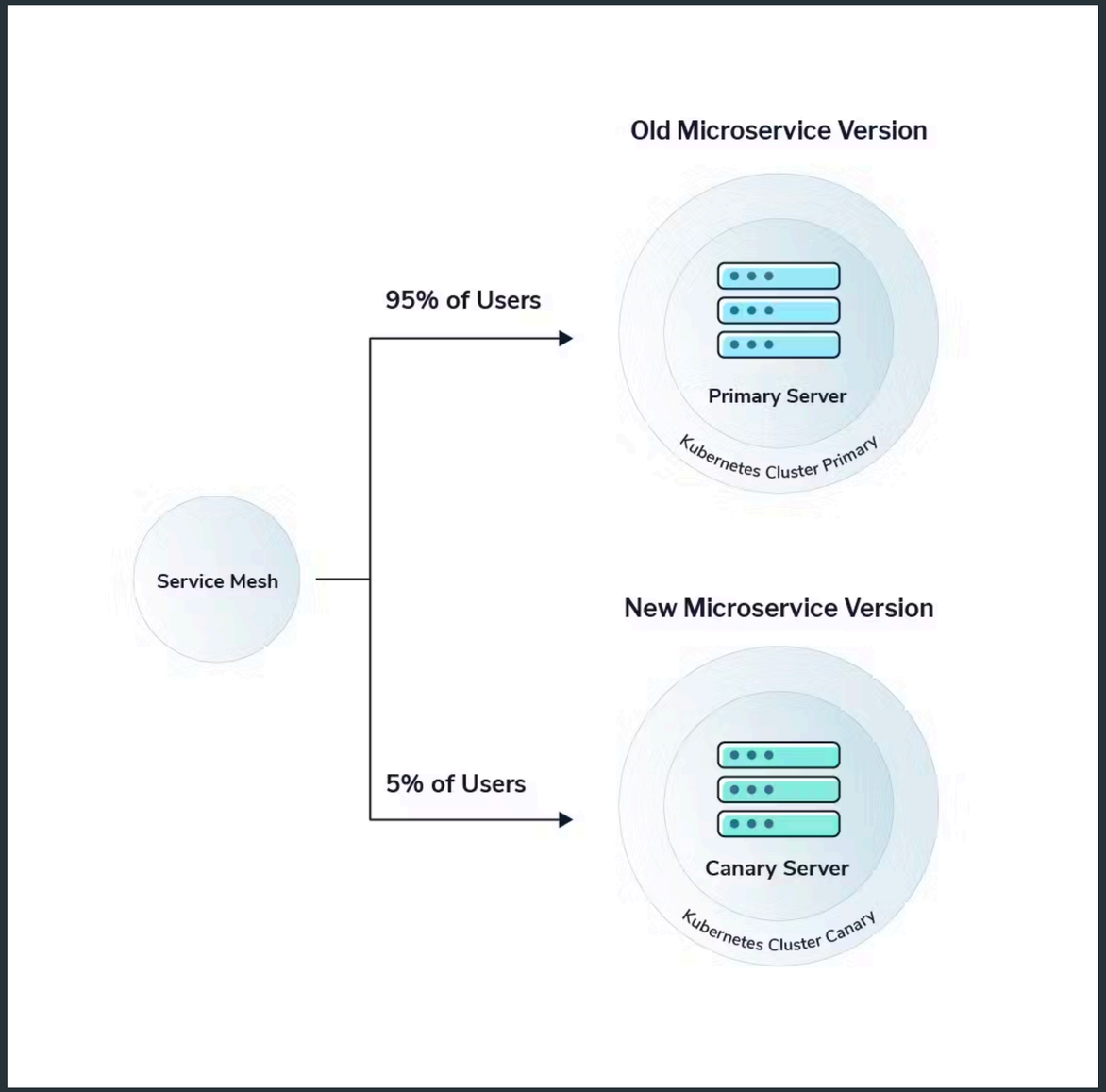
Popular Tools

- Kubernetes
- Docker Swarm
- AWS Elastic Beanstalk
- Azure App Service



Progressive Delivery for Microservices

- Progressive delivery is a software delivery strategy that gradually deploys new features to users, allowing you to monitor and control the rollout.
- Progressive delivery can be used to deploy microservices by gradually rolling out new versions of each service to a subset of users.
- This allows you to test new features in production without impacting all of your users.



Lecture outcomes

- Key Concepts
- Monolith
- Splitting the Monolith
- Communication Styles
- Technology Choices
- Deployment