Deloitte.

Microservices

Deloitte Technology Academy (DTA)

Agenda

</>

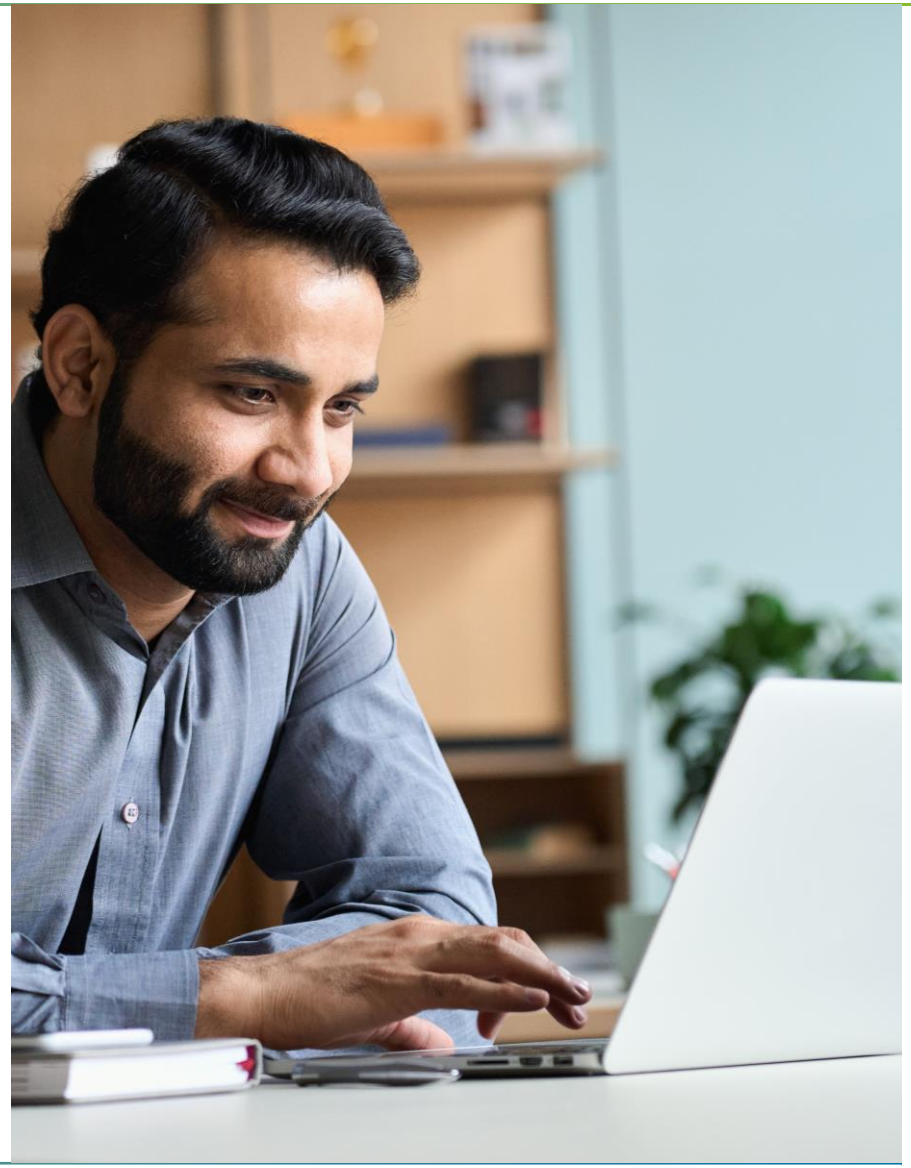
Topics	Descriptions	Duration
Introduction	Introduction to Monolithic Architecture, Drawbacks of Monolithic Architecture, Service Oriented Architecture (SOA), and Issues with SOA	X hours XX mins
Microservices	Evolution and Characteristics of Microservices, Benefits and Challenges, Microservices Overview, Architecture Comparison, Characteristics, and Challenges	X hours XX mins
Transaction	Monolithic Transactions and Microservice Transactions	X hours XX mins

Learning Objectives

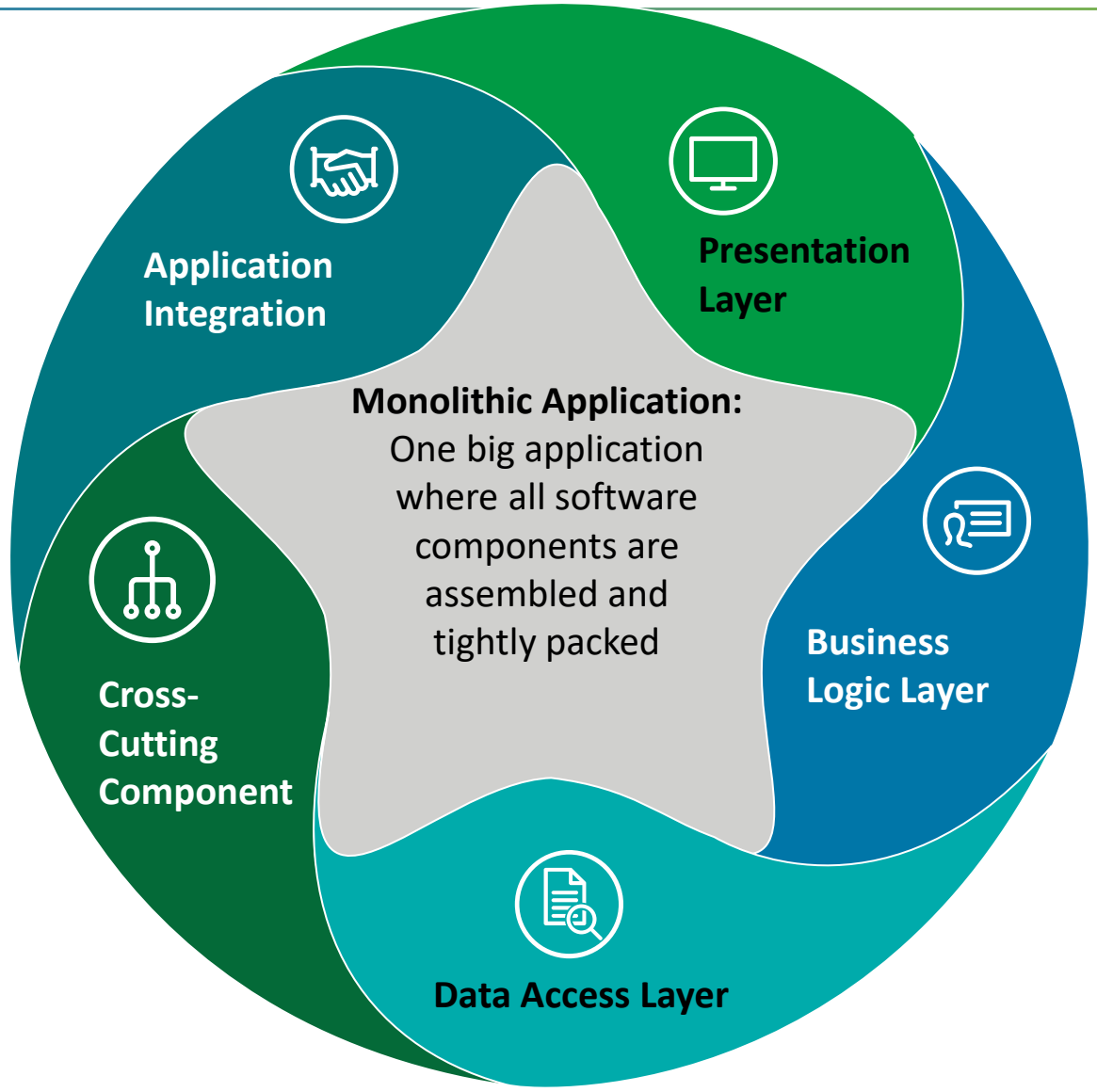
By the end of this session, you will be able to:

- Explain Microservice architecture and how it is different from the way enterprise applications were previously developed
- Understand Monolithic architecture, benefits and disadvantages
- Compare Monolithic vs Microservice
- SOA and its components/principles
- Benefits and challenges of microservices
- Understand communication between microservices

Introduction



Monolithic Architecture



Benefits of Monolithic Architecture

< / >

Simple to Develop

At the beginning of a project, it is much easier to go with monolithic architecture.

Simple to Deploy

You have to just copy the packaged application to a server.

Simple to Test

You can perform end-to-end testing by simply launching the application and testing the UI.

Simple to Scale

You can scale the instances horizontally by running multiple copies behind a load balancer.

Drawbacks of Monolithic Architecture

< / >

Difficult to maintain if the application is too large and complex to understand entirely, and it is a challenge to make changes fast and correctly.

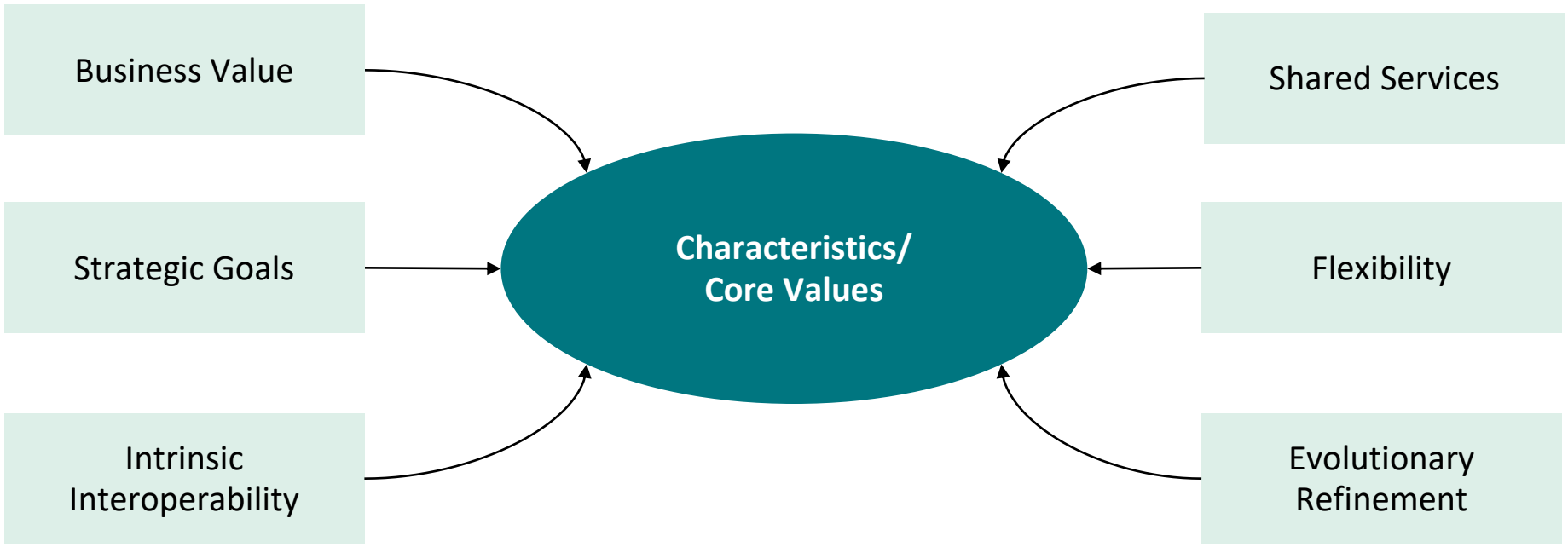
You need to redeploy the entire application on each update, and the size of the application can slow down the start-up time.

It affects reliability, as a bug in any module (e.g., memory leak) can potentially bring down the entire process.

It can be challenging to scale when different modules have conflicting resource requirements.

It is difficult to adopt new and advanced technologies since changes in languages or frameworks can affect the entire application.

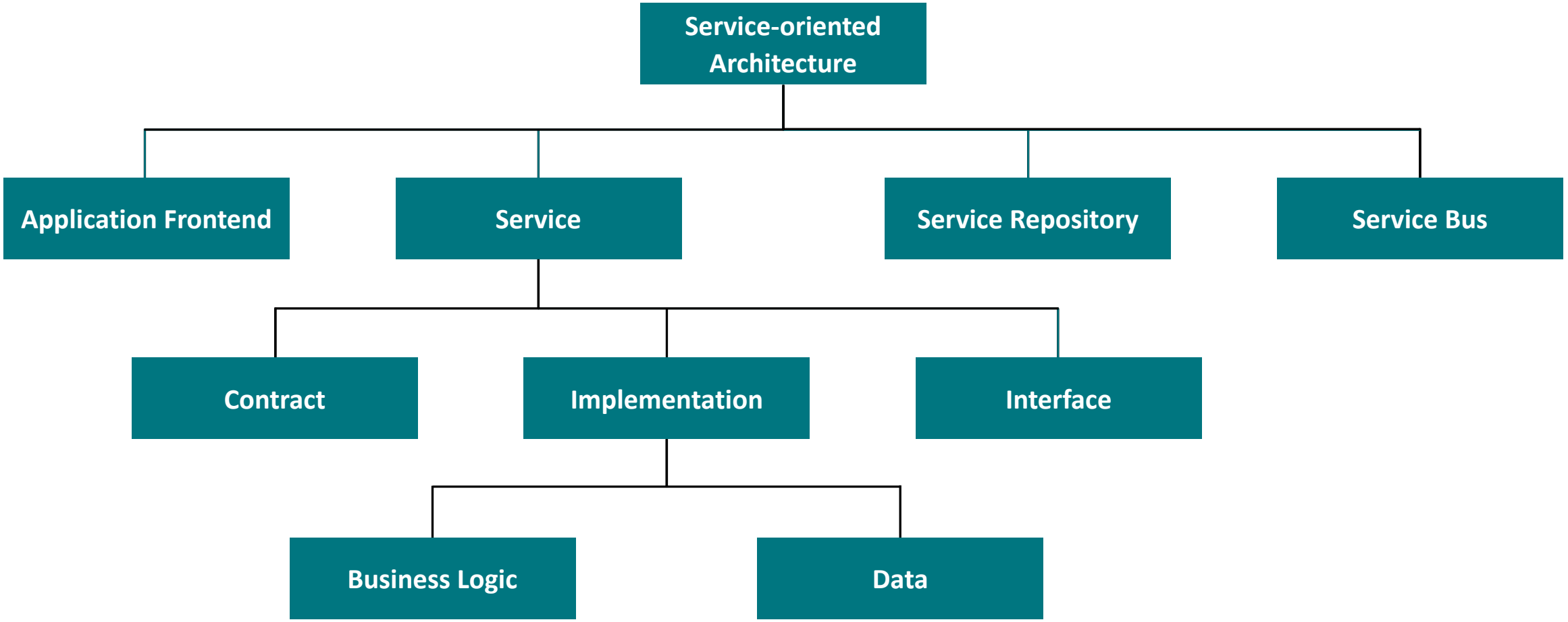
Service Oriented Architecture (SOA)



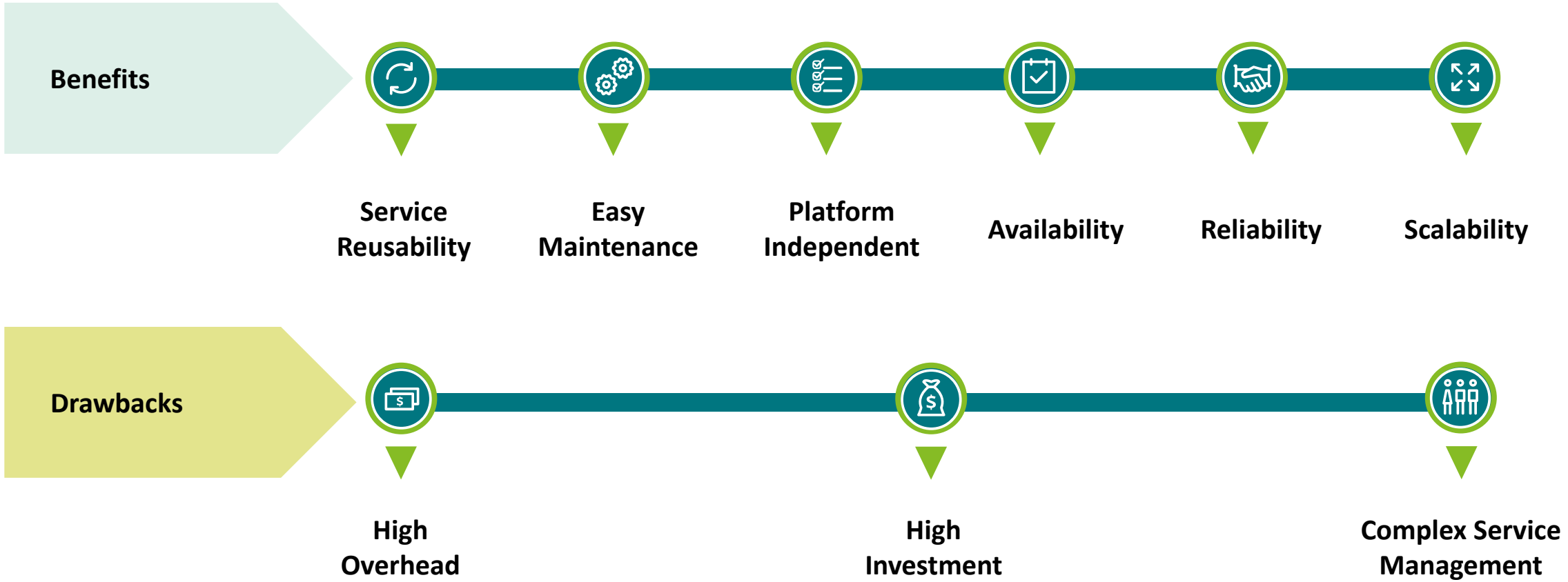
Principles

Standardized Service Contract	A service contract is specified through one or more service description documents.
Loose Coupling	By designing services as self-contained components, they can minimize dependencies on other services to maintain relationships.
Abstraction	A service is completely defined by description documents and service contracts. Their logic is hidden within the implementation through encapsulation
Reusability	Services can be reused more effectively since they are designed as components. This reduces development time and the associated costs.
Autonomy (Independent)	The logic encapsulated in services is under control and there is no need to know about their implementation from a service consumer point of view.
Discoverability	Services can be effectively discovered through defined description documents that constitute supplemental metadata. An effective means for utilizing third-party resources is provided by Service discovery.
Composability	Using services as building blocks, sophisticated, and complex operations can be implemented. Service orchestration and choreography provide solid support for composing services and achieving business goals.

Components of SOA



Benefits and Drawbacks

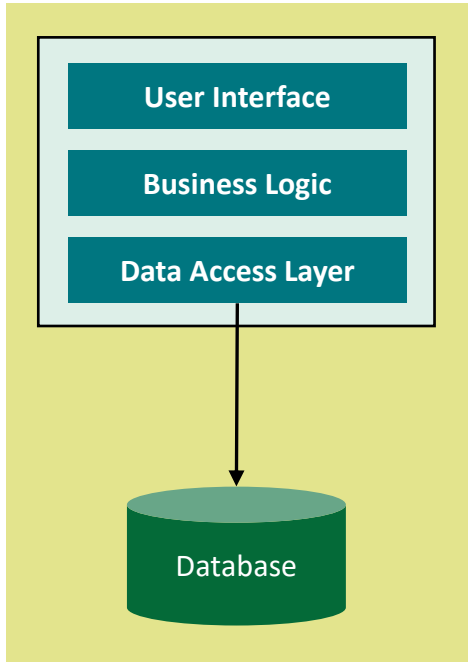


Monolithic vs. Microservice Architecture

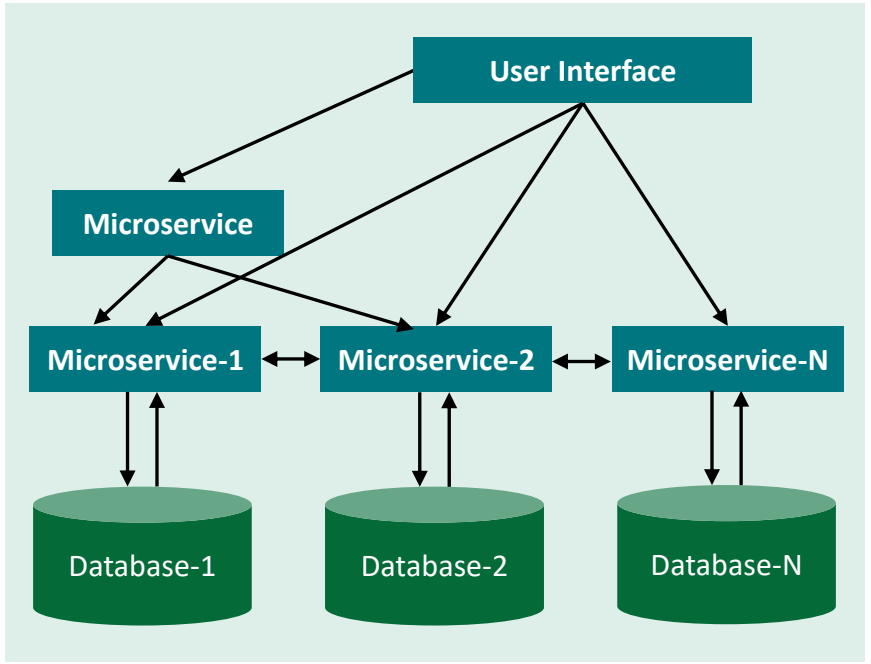
< / >

Points to infer from the adjacent figure:

- Each microservice has its own database and business layer.
- Other services are not affected by changes in one microservice.
- The use of lightweight protocols such as REST, HTTP, or messaging protocols is how these services communicate with one another.



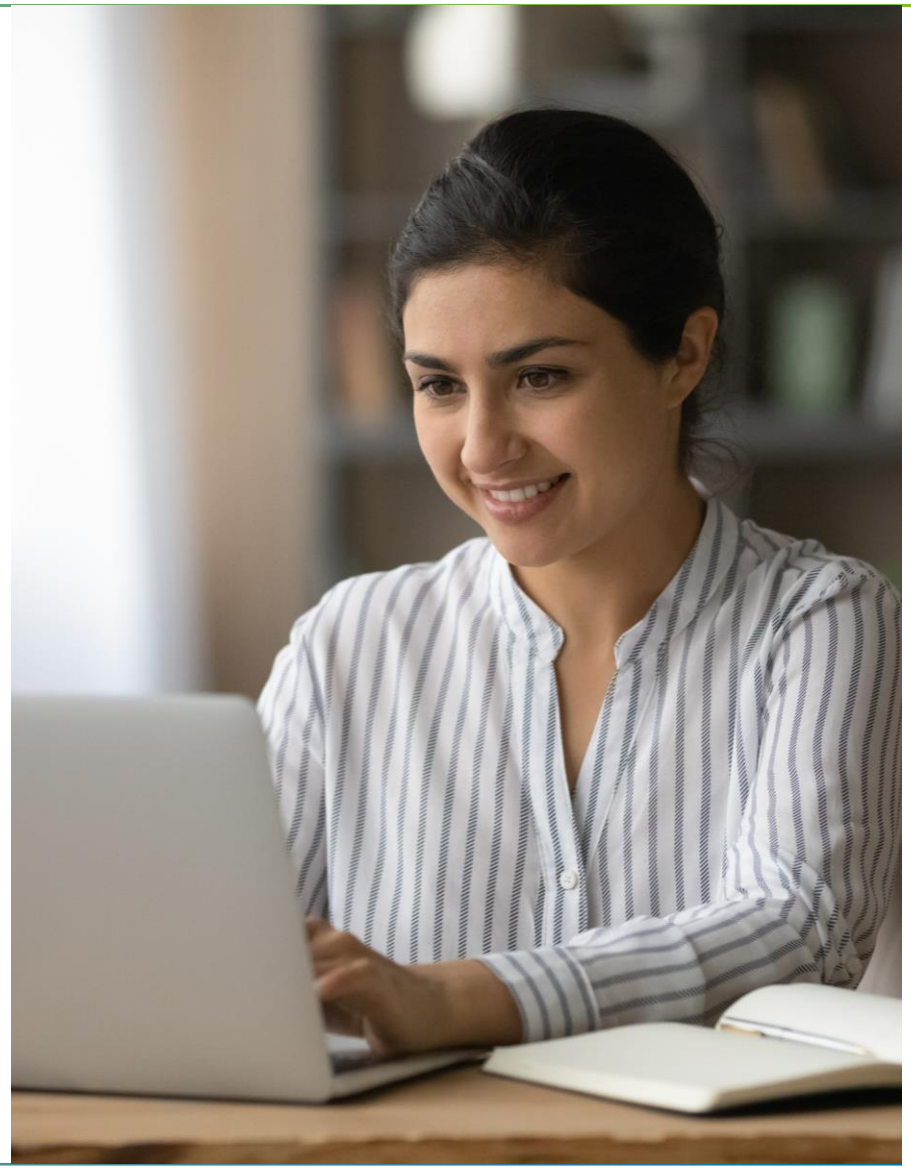
Monolithic Architecture



Microservice Architecture

</ />

Microservices



What is Microservices?

- To satisfy modern business demands for software development microservices has an architecture style and approach.
- It is an evolution from previous architecture styles—Monolithic and SOA.
- Microservices is a style of distributed architecture with several loosely coupled services providing functionality through collaboration.
- Typically, these services tend to be smaller and take one single responsibility, and hence they are developed over a few weeks rather than months.

Evolution

</>

Monolithic Architecture

- One big application with everything assembled together
- Deployment causes entire application outage
- Difficult to maintain as it requires an understanding of the entire application
- Constrained with using a single development stack

Service Oriented Architecture

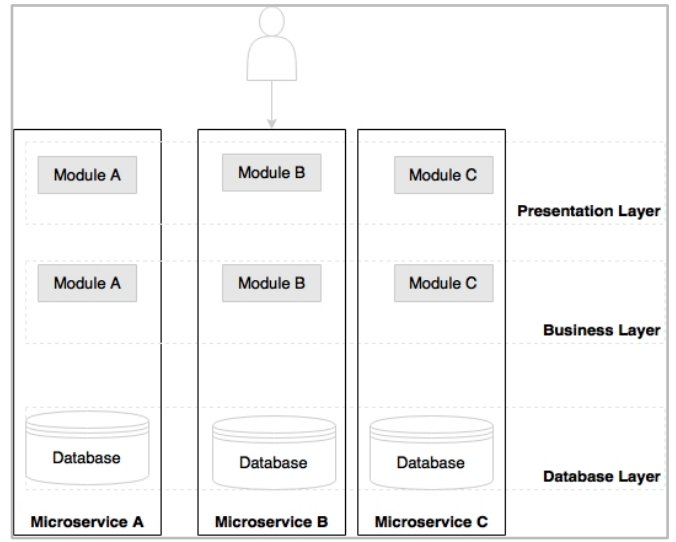
- Application logic decoupled into services
- Services are loosely coupled
- Service producer, and consumer role
- Using different technology stacks services can be implemented
- Relatively easier to maintain, and troubleshoot
- Scalable by attaching more servers
- Costly in terms of human resources, development, and technology

Microservice Architecture

- Functionality is broken into most basic level
- Increased speed of delivery
- Adaptable to agile, and DevOps model
- Lesser turnaround time, and cost
- Services can be independently deployed, maintained, and scaled as per requirement
- Platform independent
- System resilient

Characteristics

Lightweight	**Polyglot Architecture**	**Distributed and Dynamic**
Loose Coupling	**Service Abstraction**	**Service Reuse**
Statelessness	**Discoverable**	**Fault Tolerant**



Microservice Architecture

Benefits

Productivity

- Part of the functionality associated with microservice architecture is the ability to easily understand when compared to an entire monolithic app.
- A better pick is Microservices if you plan on development team expansion.

System Resilience

- The entire application will not be affected if some of the services go down.

Evolutionary Design

- No need to rewrite your whole application.
- Add new features, such as microservices, and plug them into your existing application.

Small Codebase

- Each microservice deals with one concern (SoC) only—this result in a small codebase, which means easier maintainability.

Auto Scale

- If a service will handle the larger load, you have the freedom to scale only the loaded service.

Easy to Deploy

- You do not have to redeploy the entire application, only the codebase needed.

Real-Time Use Cases

Netflix

- Implementation of the microservice architecture is best shown in Netflix.
- Due to an increased demand for its service, Netflix moved to a microservice from a Monolithic Architecture in 2009. Netflix engineers were forced to create a microservice that offered the best Internet television network through an open-source technology, since microservice did not exist then.
- Through moving to Microservices, Netflix was able to support its 10 billion hours of TV series and movies and 193 million subscribers through thousands of code sections being deployed daily by the company's developers.

Uber

- Uber launched with a monolithic architecture, similar to other startups.
- When UberBLACK was the only client service provided monolithic architecture was simpler for the company's founders. Developers switched to microservices as the startup rapidly grew and they needed to use several languages and frameworks.
- Uber now has 1,300+ microservices that focus on improving the app’s scalability.

Spotify

- With over 75 million active users, Spotify’s founders decided to build a system with independently scalable components to make synchronization easier.
- For Spotify, the main benefit of microservices is the ability to prevent massive failures.
- Even if multiple services fail simultaneously, users won’t be affected.

Complexity

< / >

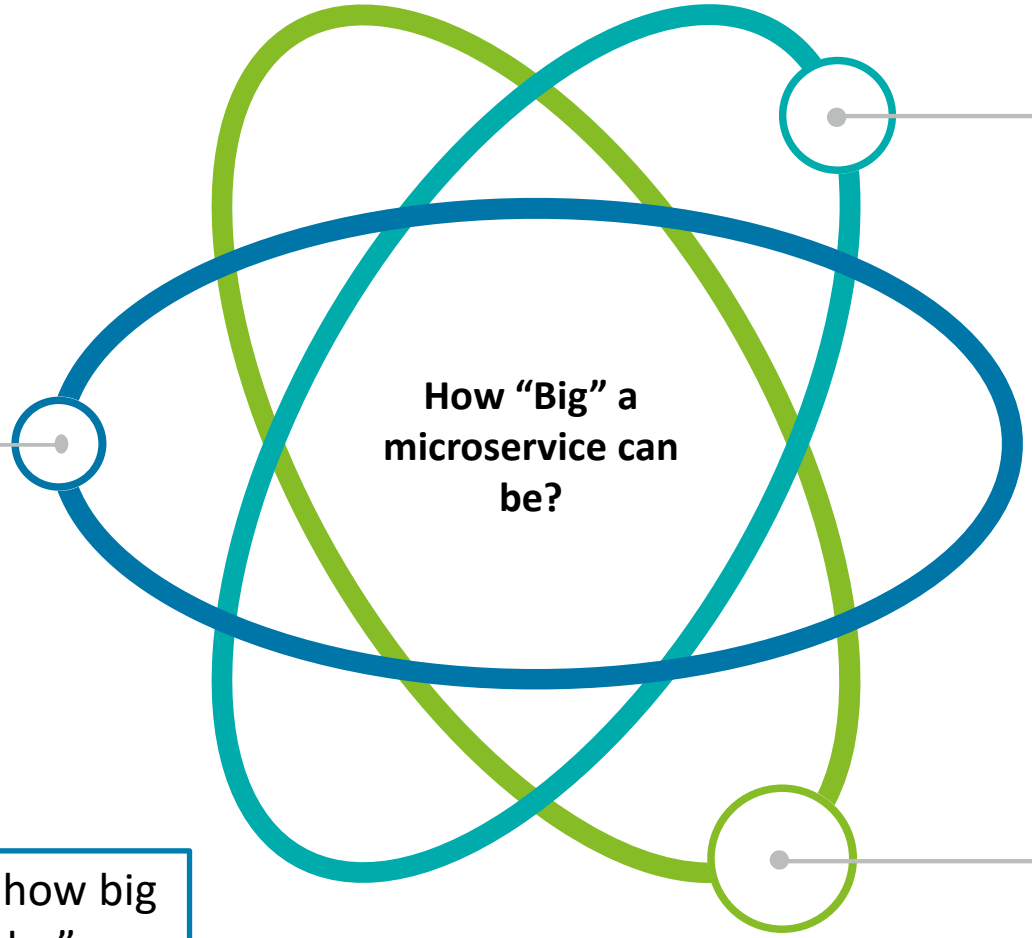
- Services as potential reuse candidates need to be easily discoverable by the project team.
- To make re-using significantly easier than building new these services should provide test consoles, documentation, etc.
- You need to closely monitor interdependencies between services.
- The impact from service outages, Downtime of services, and service upgrades should be proactively analyzed since they can cause a flow of downstream effects.

“With great power comes greater responsibility”

Complexity (Cont.)

< / >

The guideline - Amazon's Two Pizza Team - A microservice should be able to be supported by a team you can feed with two pizzas (around 12 people).



A microservice can be as small as a single API endpoint (Example: 'getOrders'), or it can be several, or even dozens of API endpoints.

Scalability-Higher the scalability (size), the more specialized the service should be.

"So, there is no hard answer on how big a microservice can be or should be"

Challenges with Microservices

Bounded Context

Deciding the boundaries of microservices is an evolutionary process and it comes with experience and business knowledge (following the Domain-Driven-Design).

Configuration Management

Microservices have multiple instances in each environment and there are multiple environments.

Example: 10 microservices with five env's and 50 instances.

Dynamic Scale up and Scale down

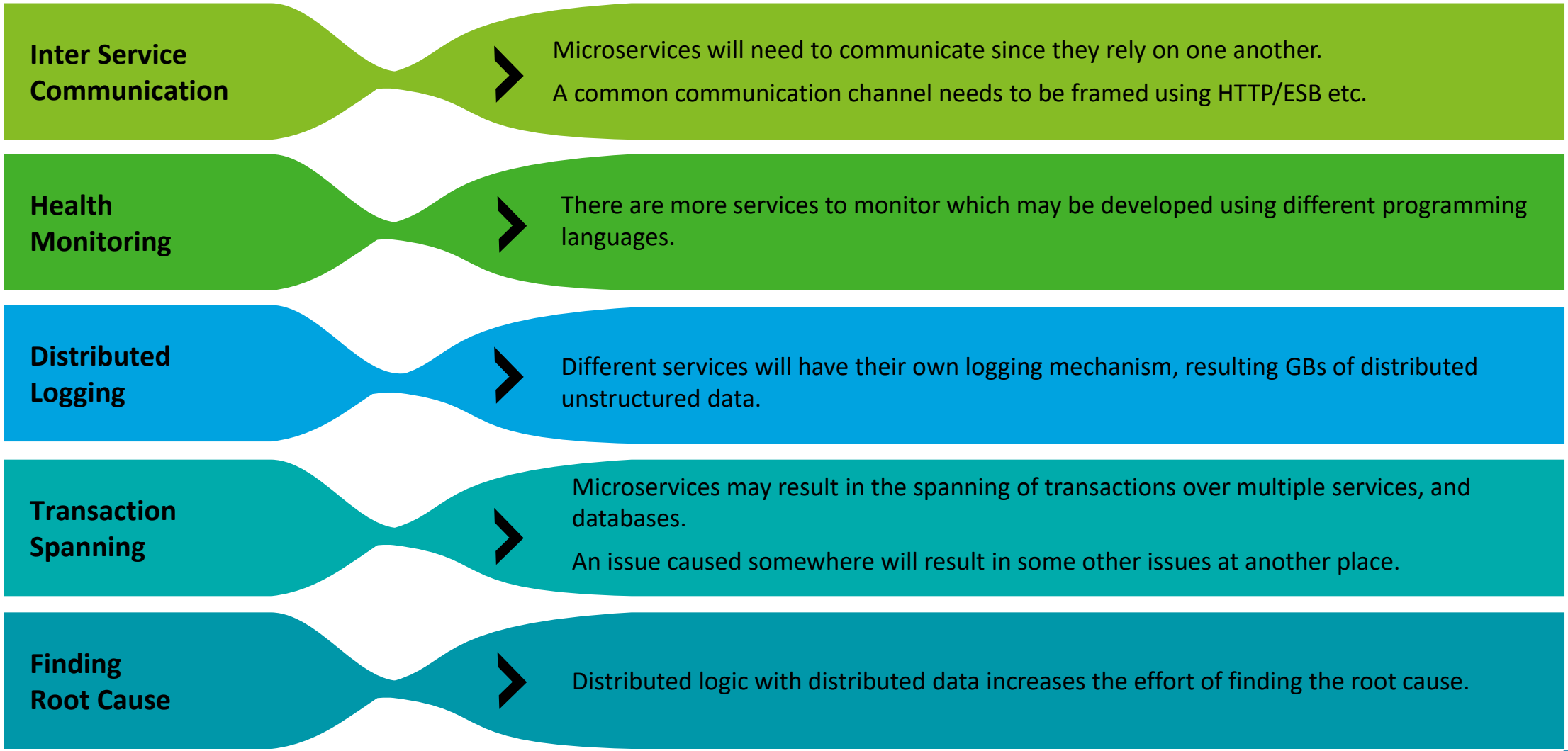
Distributing the load dynamically among the active instances is a challenge as the loads on different microservices may vary at different instances of time.

Visibility and Monitoring

Identifying the bug of a particular microservice in a centralized log and finding the root cause is challenging and monitoring the multiple microservices to make sure they are up and running.

More Challenges

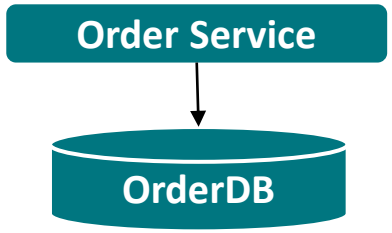
< / >



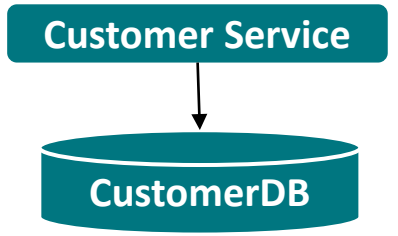
Decomposing

- Decomposing is the process of taking a larger monolithic application and breaking it up into microservices.
- Single Responsibility Principle (SRS) says a Class should have just one reason to change; meaning your classes should be very specific in what they do.
- SRS can be applied to Microservices, Do one thing, and do it very well.

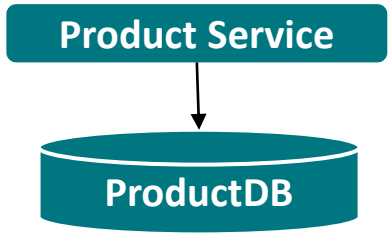
By Business Capability: Order Service



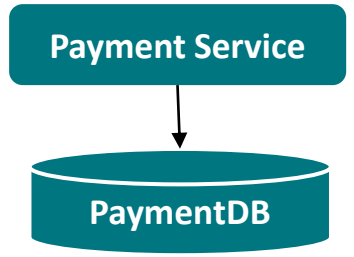
By Nouns: Customer Service



By Domain Objects: Product Service



By Action Verbs: Payment Service



A Real Time Example

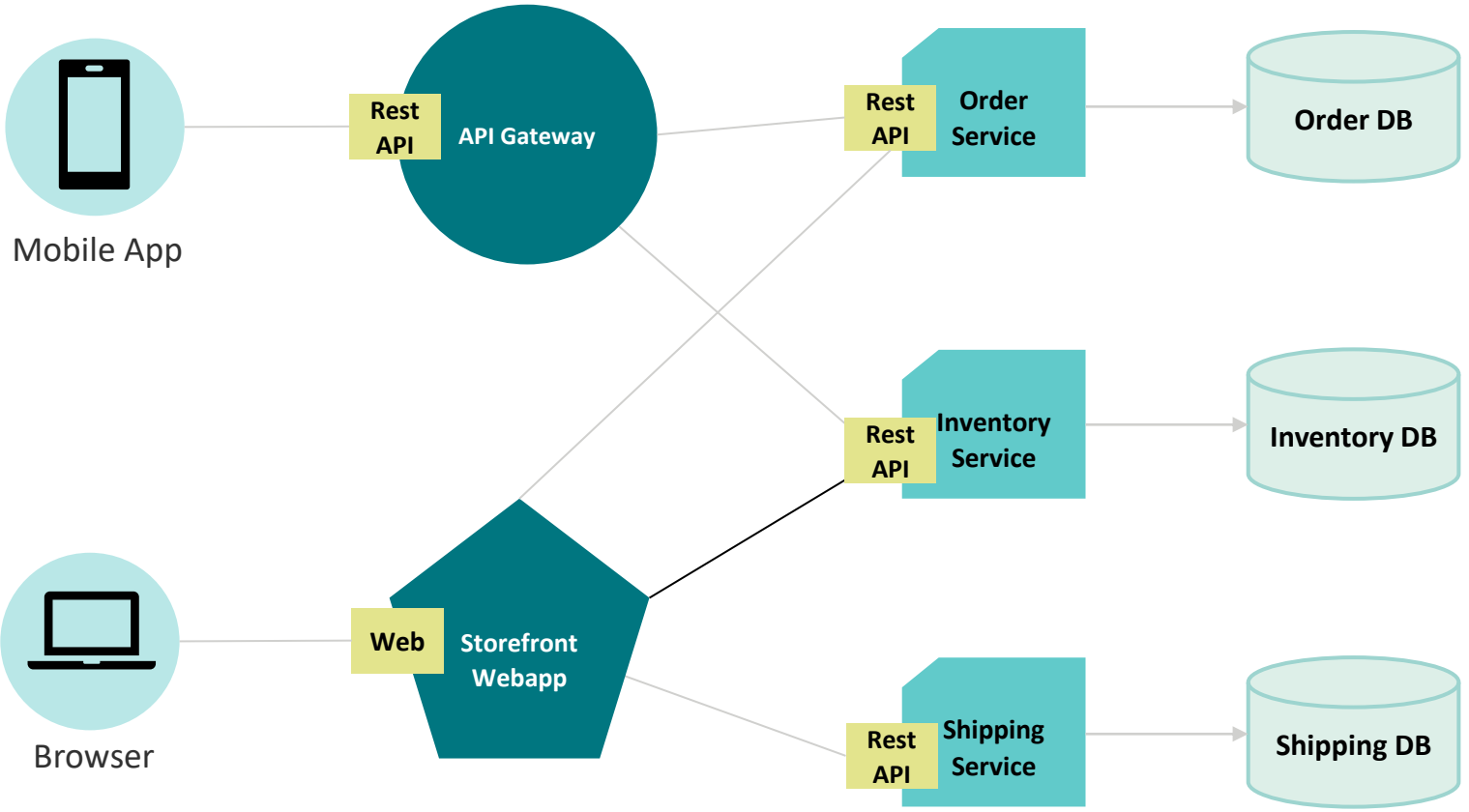
</ />

Fictitious e-commerce Application

- You need to build an imaginary e-commerce application that can verify inventory and available credit, take orders from customers, and ship those orders.
- There are several components included in the application including the Storefront UI. Through Storefront UI users can implement the user interface, can check credit using some backend services, maintain inventory, and ship orders.

A set of microservices make up the application:

- Account Service encapsulates the customer orders business logic.
- Inventory Service encapsulates the logic for maintaining inventory items.
- Shipping Service encapsulates the logic for shipping the customer order once confirmed.



Case Study

</>

Employee Service: (Microservice 1 running on Port 8080)

```
@RestController
@RequestMapping("/employeeservice")
public class EmployeeController {

    Employees employees = new Employees();

    @GetMapping("/employees/{employeeid}")
    public Employee getEmployeeById(@PathVariable("employeeid")
    Integer employeeId) {
        return employees.getEmployees().stream()
                .filter(emp -> emp.getEmployeeId() == employeeId)
                .findFirst().get();
    }

    @GetMapping("/employees")
    public Employees getAllEmployees() {
        return this.employees;
    }

}
```

Department Service: (Microservice 2 running on Port 8082)

```
@RestController
@RequestMapping("/departmentservice")
public class DepartmentController {

    List<Department> departmentList = Arrays.asList(
            new Department(1, "Department 1"),
            new Department(2, "Department 2"),
            new Department(3, "Department 3"));

    @GetMapping("/departments/{departmentid}")
    public Department getDepartmentById(@PathVariable("departmentid")
    Integer departmentId) {
        return departmentList.stream()
                .filter(dept -> dept.getDeptId() == departmentId)
                .findFirst().get();
    }

}
```

Communication Between Microservices

</ />

Employee Data: (Microservice 3 running on Port 8083)

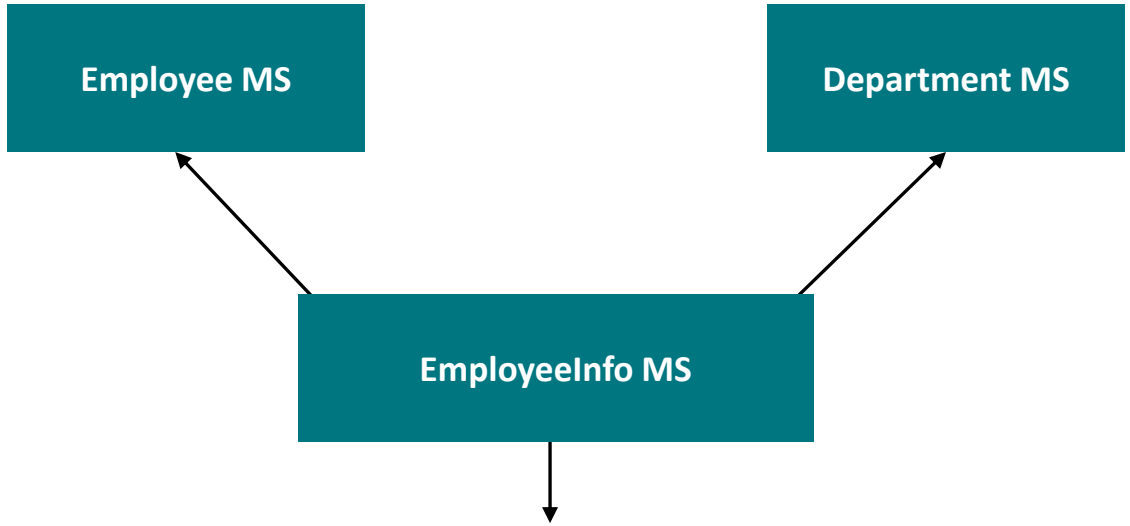
```
@RestController
@RequestMapping("/employeedataservice")
public class EmpDeptController {

    @Autowired
    EmpDeptService empDeptService;

    @GetMapping("/employeeinfo/{empId}")
    public EmployeeInfoResponse getEmployeeInfo(@PathVariable("empId")
    Integer empId) {
        Employee employee = empDeptService.getEmployeeById(empId);

        return new EmployeeInfoResponse(
                employee.getEmployeeName(),
                employee.getSalary(),
                empDeptService.getDeparmentById(employee.getDeptId())
                .getDeptName());
    }

}
```

Employee Data Service: Communication with MS1 and MS2

```
@Service
public class EmpDeptService {

    @Autowired
    RestTemplate restTemplate;


    public Employee getEmployeeById(Integer empId) {
        return restTemplate.getForObject(
                "http://localhost:8080/employeeservice/employees/"+empId,
                Employee.class);
    }

    public Department getDeparmentById(Integer deptId) {
        return restTemplate.getForObject(
                "http://localhost:8082/departmentservice/departments/"+deptId,
                Department.class);
    }

}
```

Communication Between Microservices (Cont.)

Employee Data: (In Action)



REST API: /employeedataservice/employeeinfo/1

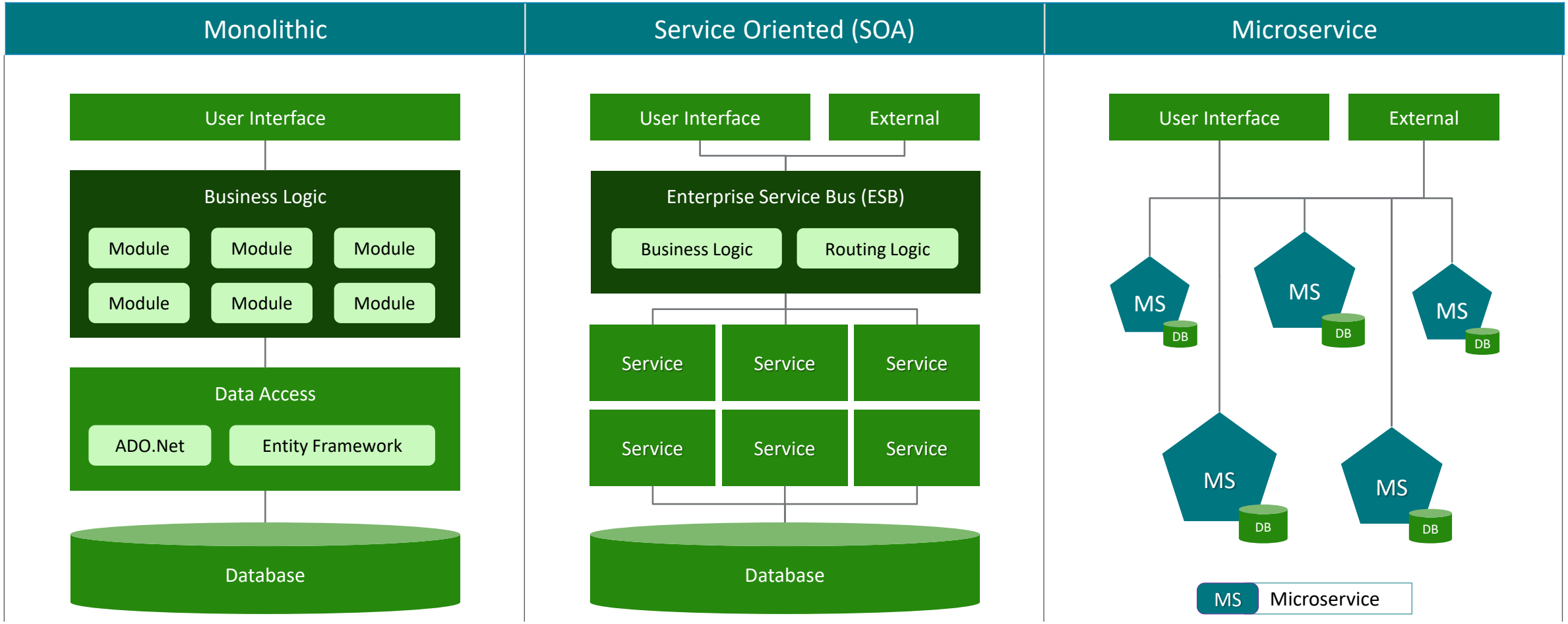
{"name":"Paul ","sal":1500.0,"deptName":"Department 1"}

Microservice—Overview

Microservice is an architecture style—A variant of the Service-Oriented Architecture (SOA), that structures an application as a collection of small, focused, and loosely coupled services.

- In this architectural style, a collection of small autonomous services is modeled around a business domain.
- Each service implements a single business capability and is self-contained to primarily use its own domain model.
- Each such service should own its domain data model, underlying business logic, and data.
- Conceptually, multiple microservices in an application can use different data storage technologies (SQL, NoSQL), and different programming languages.
- Each service communicates with each other using protocols such as HTTP/HTTPS, WebSocket, or Advanced Message Queuing Protocol (AMQP) and runs in its own process.
- Loosely coupled services have autonomy of development, deployment, and scale, for each service.

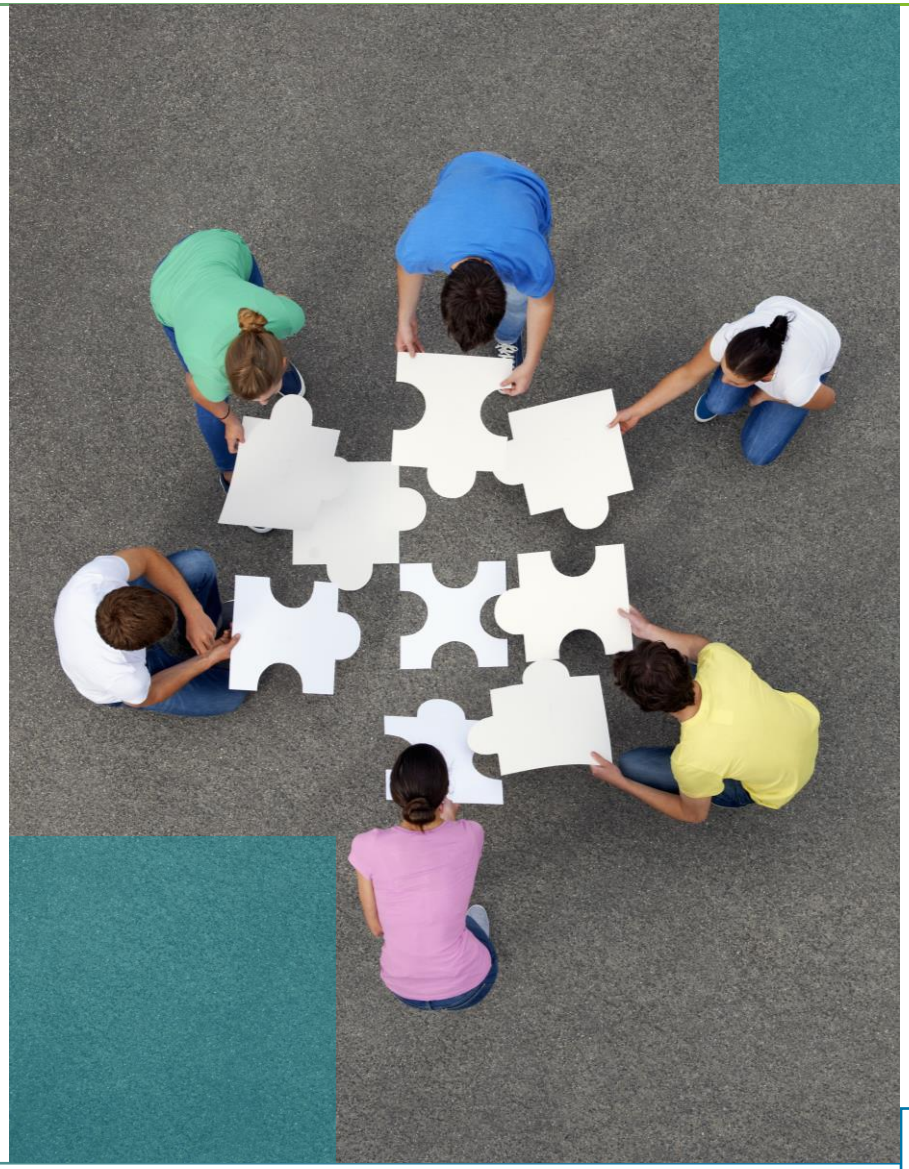
Monolithic vs. Service Oriented (SOA) vs. Microservice



Monolithic Architecture—Challenges

Monolithic application design has a single large database system and all other components (data layer, business layer, etc.,) are interwoven within a single application codebase.

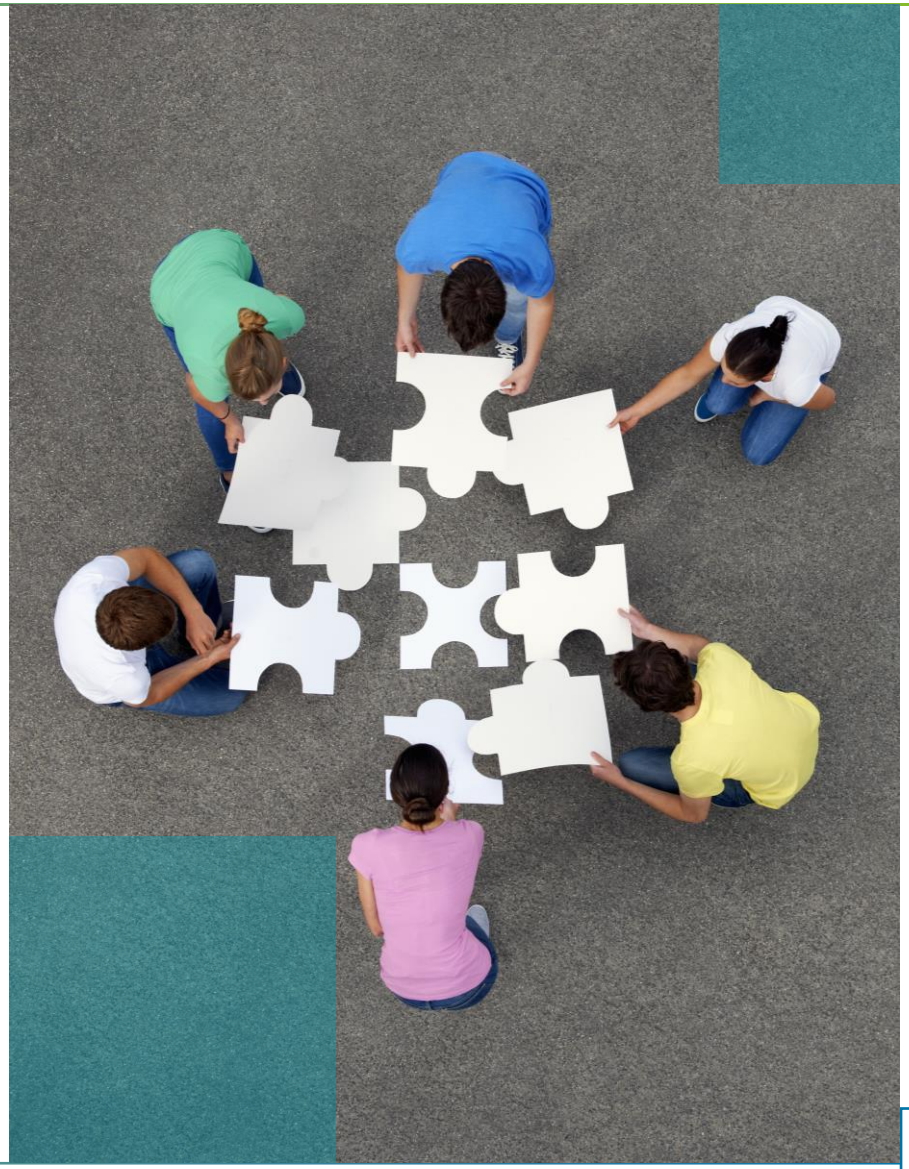
- **Difficult to maintain**—Large and complex monolithic applications are generally difficult to understand, and it slows down the maintenance. A lot of regression testing and validations may even be required for small changes.
- **Higher ramp-up time**—Generally any new developer joining a team requires a lot of time to reasonably understand the system and start making changes to it confidently.
- **Complex build and deployments**—Monolithic applications require the complete deployment of the system including databases and other dependencies–all together. This makes the deployment process quite complex and time-consuming. It's very difficult to implement CI and CD on monolithic applications.
- **Difficult to scale**—Since a monolithic application is one large codebase, it can be scaled entirely as a whole. In its original form, it's not possible to scale up or down only certain modules or features.
- **Error prone**—Small changes/bug fixes might require a lot of impact analysis, and potentially might break the system.
- **Inflexible**—Monolithic applications generally face barriers to the adaptation of new technology.



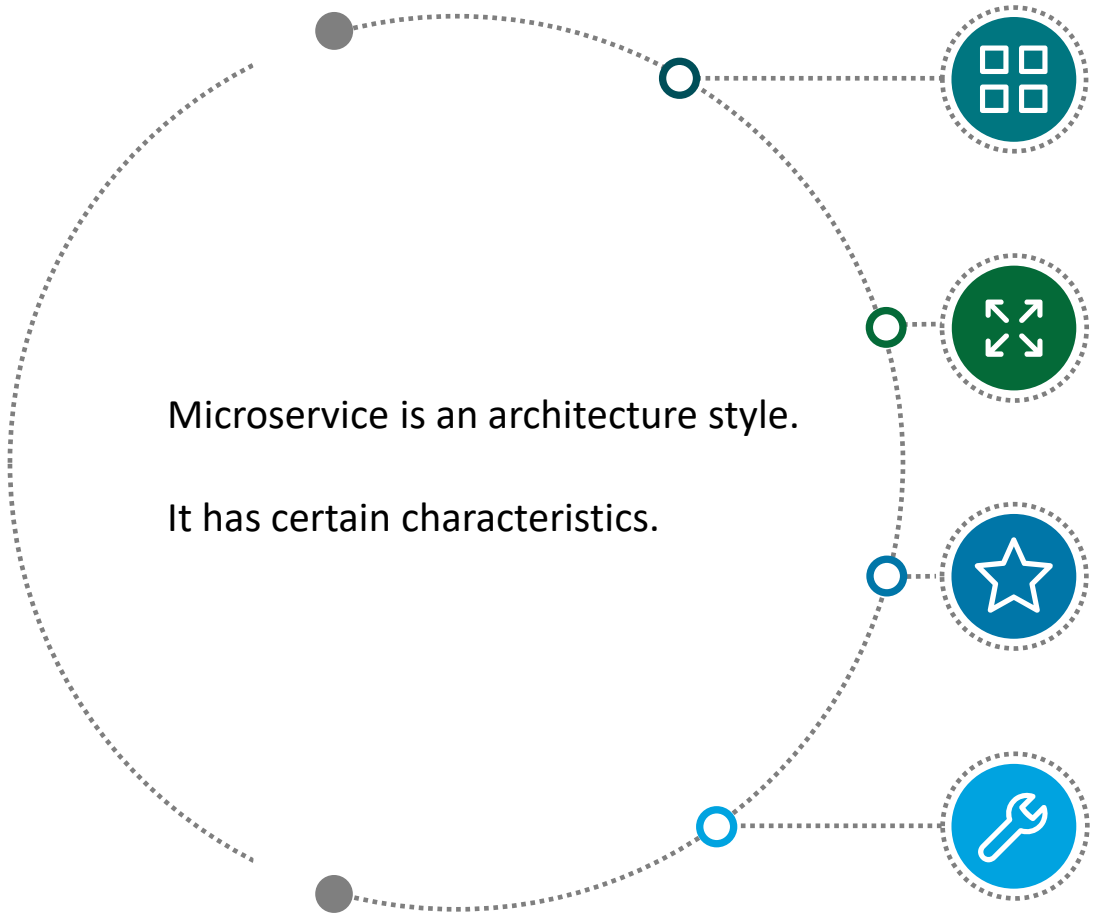
Advantages

< / >

- **Business-focused (Domain-Driven) design**—Small and autonomous teams based on the business model (domain) and not based on technology (e.g., database, interface team, etc.). In this model, the team is focused on the domain functionality.
- **Easy to modify or replace**—It's always easy to maintain or replace a microservice with a newer version. Also, adopting newer technology can be tried one of the microservice independently without affecting other services.
- **Design for failure**—In a microservice architecture, consuming applications or other services are expected to handle any unhandled exception while using dependent services and eventually handle the error gracefully. Also, when dependent service is not available, microservice is expected to stop making further requests to it and allow it to recover.
- **Reusability**—Microservices are available as RESTful APIs. Also, these services cater to a very specific business need/functionality. This makes it project and technology agnostic.
- **Incremental development**—Smaller and most independent functionalities can be developed as an initial microservice. Later more domains can be identified, and services can be added as needed.
- **Scalable**—With a set of smaller services microservice-based applications can be easily scaled in both directions (up/down) as per the request load.
- **Continuous Delivery**—CI and CD can be achieved on large complex projects having microservices architecture, as only modified services are deployed to the server as compared to the entire codebase in the case of monolithic.



Microservice—Characteristics



Componentization

- Around business capabilities (Domain-driven design)
- Via REST APIs (Autonomous services)
- Design for failure (Internal or external dependencies)

Decentralization

- Decentralized governance
- Decentralized data management (separate data store)
- Smart endpoints and dumb pipes

Infrastructure

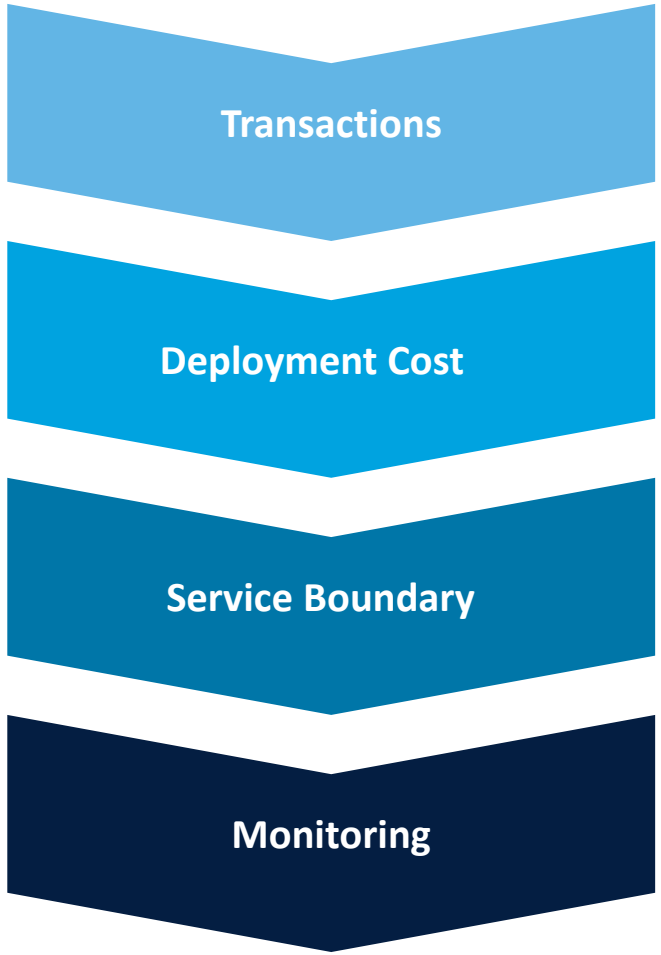
- Cloud infrastructure
- Test automation
- Deployment automation (CI-CD)
- Tracing and auditing

Maintenance

- Products not projects
- Evolutionary design
  (deploy only updated components and replace/upgrade as needed)

Challenges

Microservice might not be the best-suited architecture style for all applications. Generally, it's adopted by large enterprise applications undergoing frequent changes and requiring quick turnaround.



Unlike monolithic, it's not straightforward to implement transactions in microservices. Any functional flow performing CUD operations across multiple microservices would require more code to maintain data sanity. The project should have a structure/framework in place to deal with transaction needs.

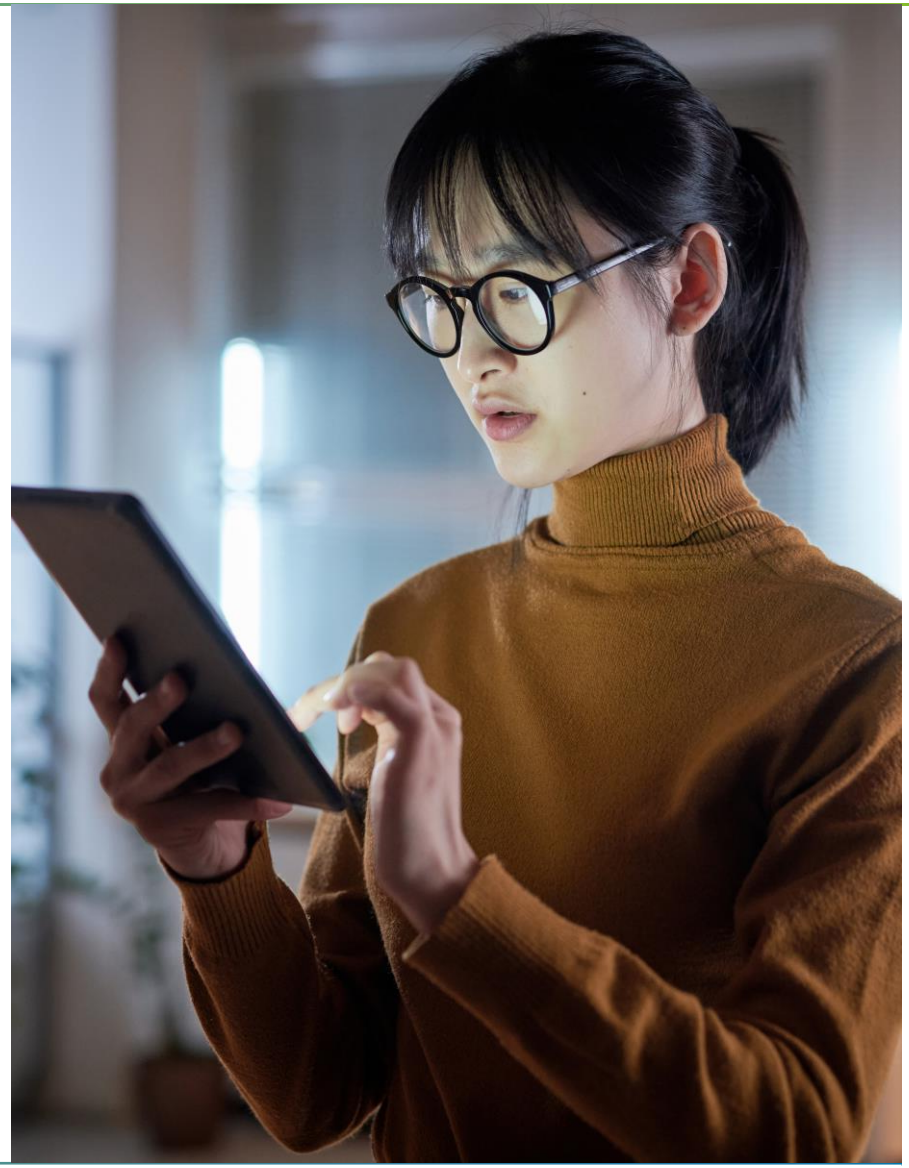
As the application grows larger, there might be a greater number of microservices running in parallel, and that too using different tools, technology, and configuration. This can potentially increase deployment costs.

Microservices are designed well along with their underlying business model. A lot of time business models and related functionality overlap, and it becomes confusing when deciding the boundary of service. It should not happen that one service has code and functionality related to the other.

Monitoring a distributed application can be a lot harder as compared to a monolithic application. Such monitoring would require consolidation and establishing the correlation between telemetry from multiple services.

</ />

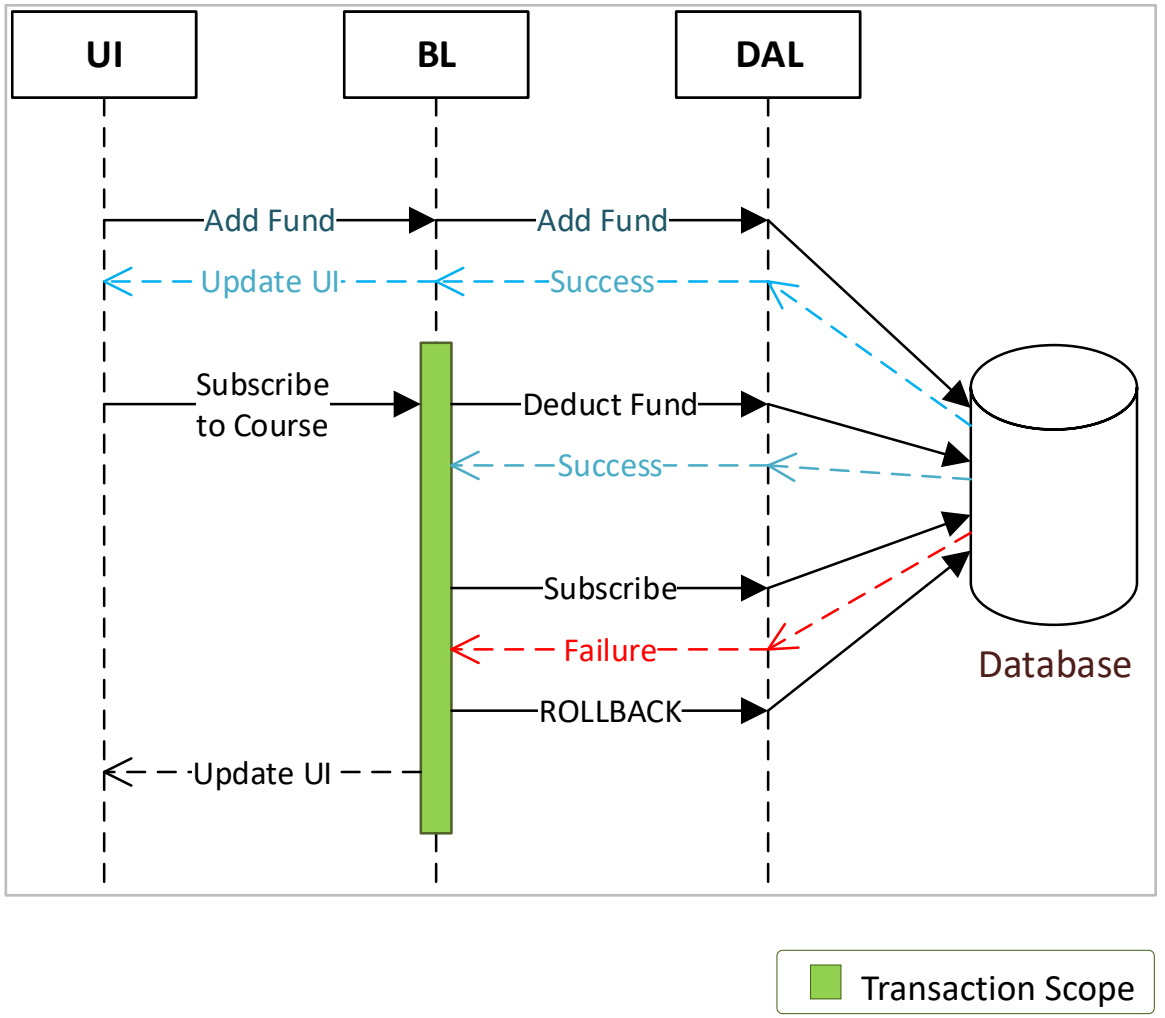
Transaction



Distributed Transactions

Monolithic Application:

- Distributed Transactions generally span across multiple physical systems/databases over a network.
- Generally, an enterprise application developed using monolithic architecture, uses transactions to maintain data integrity and sanity.
- Transactions can be implemented within the database system, where multiple DML statements are executed within a single stored procedure.
- Transaction should be atomic, i.e., either all or no execution is successful.
- Typically, Business Layer plays the role of transaction orchestrator. It manages transactions across different stored procedures or different systems.
- COMMIT is performed once all DML executions within the scope of the transaction have been completed successfully.
- ROLL BACK is performed if any one of the DML execution within the scope of the transaction has failed.



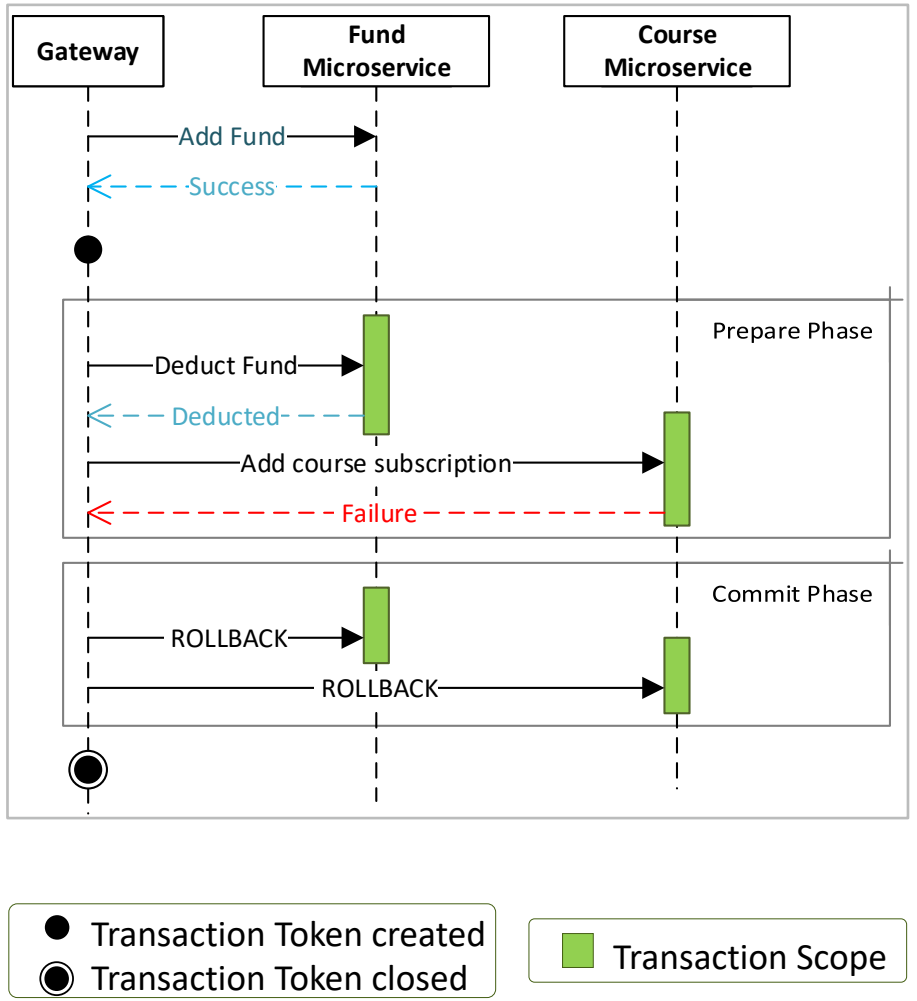
Microservices Architecture

< / >

- Microservice can have heterogeneous underlying systems/databases.
- Also, different microservice in the same application may use different programming/scripting language.
- In microservice architecture keeping transactions atomic is not as straightforward as in monolithic systems.
- An intermediate layer (API Gateway/Transaction orchestrator) manages the transaction scope exposed by microservices. It's more of a coordinator role and is called a Two-phase commit or popularly known as 2PC.
- Alternatively, an event bus is used to maintain the atomicity of transactions between different microservices. A dependent microservice performs an action based on the event trigger. Here, microservice will also expose counter operation events to handle rollbacks. It is more of a choreographer role and is called Eventual Consistency and Compensation or popularly known as Saga.
- There are advantages and disadvantages with both approaches.

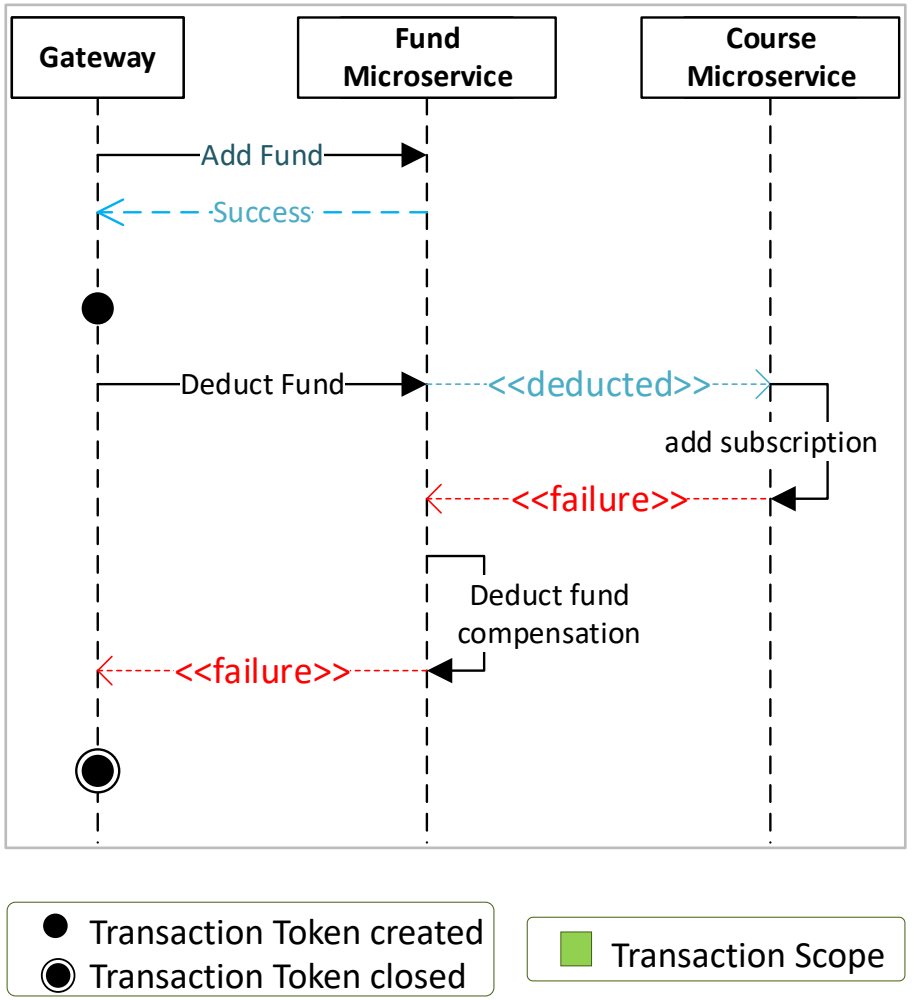
Two-Phase Commit (2PC)

- As the name suggests, this mechanism has two phases to accomplish transactions i.e., ‘Prepare Phase’ and ‘Commit Phase’.
- With this approach, a coordinator (like API Gateway) is required outside of microservices to initiate prepare, and commit phase calls.
- Advantages:
  - Strong consistency and guarantees atomicity of transaction
  - Allows read-write isolations. This means only committed changes are visible to other users.
- Disadvantages:
  - This approach follows the synchronous flow and is hence generally not recommended for microservices architecture.
  - This mechanism will have to lock the resource, and this might create a performance bottleneck.



Eventual Consistency and Compensation/Saga

- Distributed transactions are achieved by multiple asynchronous local transactions on related microservices.
- An event bus is used to establish async communication between two microservices.
- This is widely used and considered the preferred approach.
- Advantages:
  - This approach support long-lived transactions.
  - Each microservice focus on its own local transaction and doesn't block other service and their execution.
  - Transactions run in parallel and don't require a lock.
- Disadvantages:
  - Difficult to debug with a greater number of microservices
  - This mechanism doesn't have read-lock out of the box (however, same can be achieved using custom code i.e., use of flags).



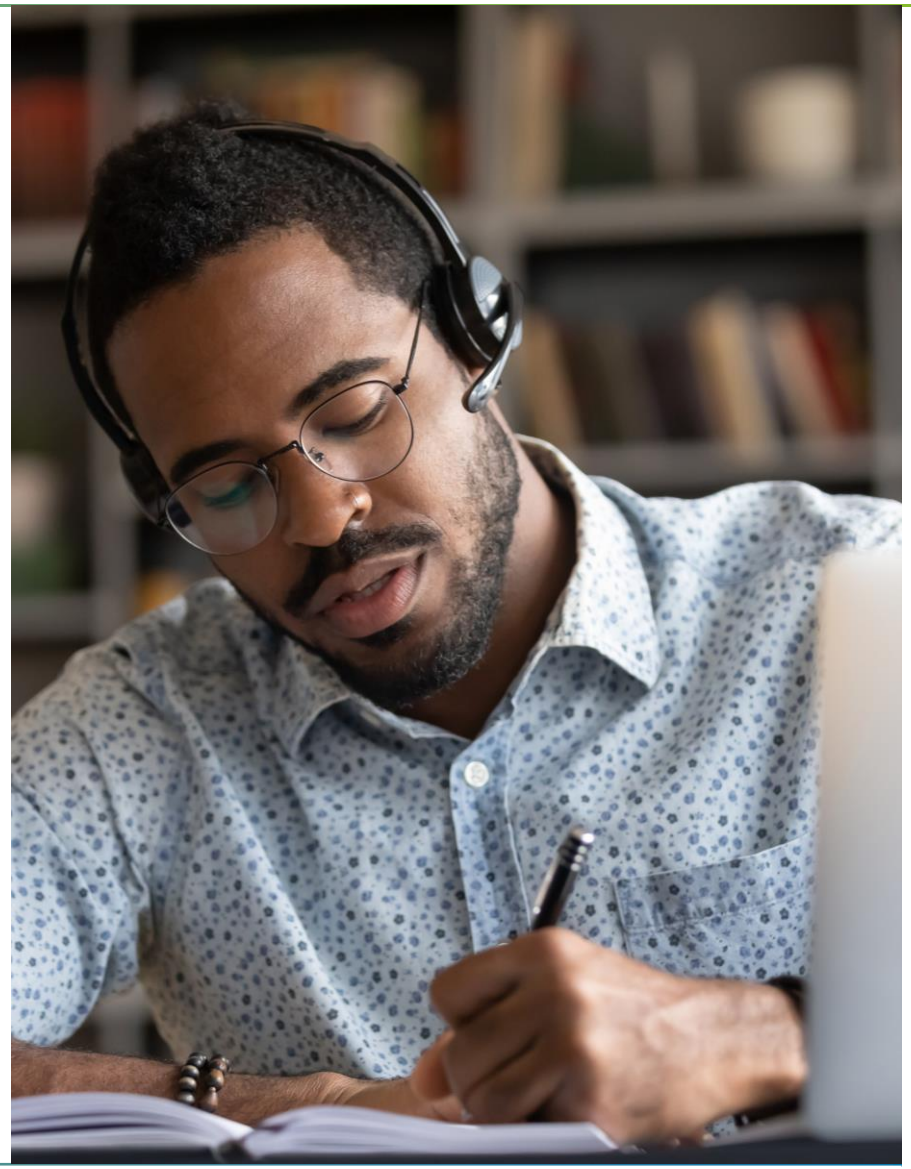
Summary

</>

Here are the key learning points of the module.

- Monolithic application is one big application where all software components are assembled and tightly packed.
- Microservices is an architecture style and an approach for software development to satisfy modern business demands.
- Distributed Transactions generally span across multiple physical systems/databases over a network.

Hands-On Labs



Hands-On Activity 1

</ />

Activity Details	
Problem Statement	Create a simple microservice called shopping-service. Expose REST API to retrieve available products for an online shopping app.

Sample Input	GET API should return all the Products available
	GET - /shopping/products - returns List<Product>
	Product: {
	String productId;
	String productName;
	String brand;
	int price;
	int stock;
	}

Thank You

Deloitte.