# PAD Laboratory Work 1

# Ceban Dan FAF-202

## E-Commerce Car Shop

### Assess Application Suitability (2p.)

1. **Complexity and Scalability:** E-commerce applications often grow in complexity as they offer more features and handle a large number of users. Microservices allow you to break down this complexity into smaller, manageable services that can be independently developed, deployed, and scaled.

2. **Independent Development:** In an e-commerce platform, various components like the product catalog, user accounts, order processing, and payment handling can evolve independently. Microservices enable different teams to work on these components without affecting each other's codebase.

3. **Technological Diversity:** Different parts of your application may require different technologies. Microservices allow you to choose the most suitable technology stack for each service. For instance, you can use JavaScript for the frontend, Python for order processing, and Java for payment handling.

4. **Fault Isolation:** Microservices provide fault isolation, meaning that if one service fails or experiences issues, it doesn't necessarily bring down the entire application. This enhances the overall system's fault tolerance and availability.

5. **Scalability:** Microservices can be scaled independently based on the demand for each service. For example, the product catalog service may need more instances during a sale event, while the order processing service may require more resources during order spikes.

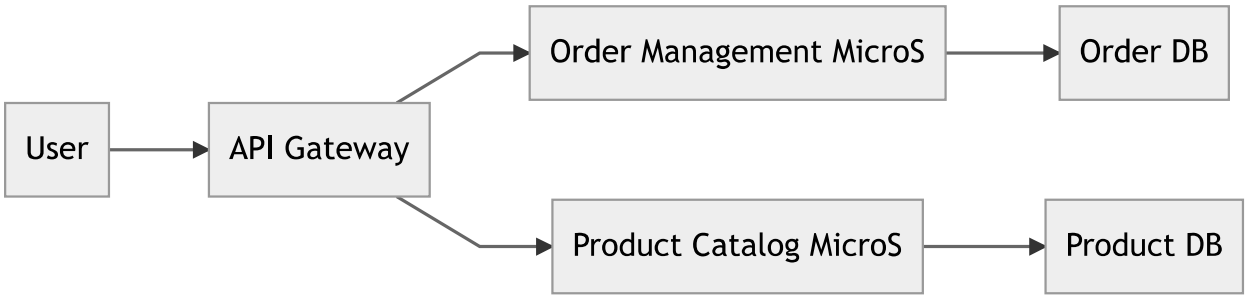
6. **Continuous Delivery:** Microservices facilitate continuous integration and continuous delivery (CI/CD), making it easier to release updates and new features more frequently without affecting the entire application.

7. **Third-party Integration:** E-commerce platforms often integrate with various third-party services like payment gateways, shipping providers, and analytics tools. Microservices can be dedicated to handling these integrations, making it easier to manage and maintain them.

8. **Real-world Examples of Projects Employing Microservices:** Uber's platform for ride-hailing and food delivery is built on microservices. Different microservices handle tasks such as user authentication, real-time location tracking, and payment processing, allowing Uber to provide a seamless and scalable experience to users worldwide. Also, Amazon, one of the world's largest e-commerce platforms, relies on microservices to power its website and services. Each function, from product search to recommendation algorithms, is encapsulated within a microservice, enabling Amazon to scale and evolve its platform continuously.

## Define Service Boundaries (2p.)

Creating clear service boundaries is essential in a microservices architecture to ensure modularity and independence of each microservice. Here's a simple system architecture diagram illustrating the service boundaries for an e-commerce application for cars:




## Choose Technology Stack and Communication Patterns (2p.)

### Microservice 1: Product Catalog Service (JavaScript)

- **Programming Language:** JavaScript (Node.js)
- **Framework:** Express.js for building RESTful APIs
- **Database:** MongoDB for storing car information (NoSQL database)
- **Communication Pattern:** RESTful API for synchronous communication with the frontend and other services.
- **Authentication:** Use JWT (JSON Web Tokens) for authentication and authorization.

**Microservice 2: Order Management Service (Python)**

- **Programming Language:** Python
- **Framework:** Flask or FastAPI for building RESTful APIs
- **Database:** PostgreSQL for storing order and user data (Relational database)
- **Communication Pattern:** RESTful API for synchronous communication with the frontend and other services.
- **Payment Integration:** Use a third-party payment gateway's API for handling payments.
- **Asynchronous Processing:** Implement a message queue system (e.g., RabbitMQ or Apache Kafka) for handling asynchronous tasks like sending order confirmation emails.

## Design Data Management (3p.)

Designing data management and defining endpoints for microservices is a critical aspect of e-commerce application. Below, I'll enumerate the key endpoints for both microservices and describe the data to be transferred in JSON format, along with the expected response for each endpoint.

**Microservice 1: Product Catalog Service (JavaScript)**

1. **Endpoint: GET /cars**
    - **Description:** Retrieve a list of all available cars in the catalog.
    - **Request:** None (GET request)
    - **Response:**

```
{
  "cars": [
```

```
    {
      "id": "1",
      "make": "Toyota",
      "model": "Camry",
      "year": 2023,
      "price": 25000,
      "available": true,
      "image_url": "https://example.com/car1.jpg"
    }
  ]
}
```

2. **Endpoint: GET /cars/{carId}**

- **Description:** Retrieve details of a specific car by its ID.
- **Request:** None (GET request)
- **Response:**

```
{
  "car": {
    "id": "2",
    "make": "Toyota",
    "model": "Prius",
    "year": 2023,
    "price": 52000,
    "available": true,
    "image_url": "https://example.com/car2.jpg"
  }
}
```

3. **Endpoint: GET /cars/search?make=Toyota&model=Avalon&year=2023**

- **Description:** Search for cars based on make, model, and year.
- **Request:** Query parameters for filtering
- **Response:**

```
{
  "cars": [
    {
      "id": "3",
      "make": "Toyota",
```

```
      "model": "Avalon",
      "year": 2023,
      "price": 80000,
      "available": true,
      "image_url": "https://example.com/car3.jpg"
    }
  ]
}
```

**Microservice 2: Order Management Service (Python)**

1. **Endpoint: POST /orders**
    - **Description:** Place a new order for one or more cars.
    - **Request:**

```
{
  "user_id": "123",
  "cars": [
    {
      "car_id": "1",
      "quantity": 2
    }
  ]
}
```

**Response:**

```
{
  "order_id": "456",
  "status": "pending",
  "total_price": 50000
}
```

2. **Endpoint: GET /orders/{orderId}**

- **Description:** Retrieve details of a specific order by its ID.
- **Request:** None (GET request)
- **Response:**

```
{
  "order": {
    "order_id": "456",
    "status": "pending",
    "total_price": 50000,
    "user_id": "123",
    "cars": [
      {
        "car_id": "1",
        "quantity": 2
      }
    ]
  }
}
```

3. **Endpoint: GET /users/{userId}/orders**

- **Description:** Retrieve a list of orders for a specific user.
- **Request:** None (GET request)
- **Response:**

```
{
  "orders": [
    {
      "order_id": "456",
      "status": "pending",
      "total_price": 50000,
      "user_id": "123",
      "cars": [
        {
          "car_id": "1",
          "quantity": 2
        }
      ]
    }
  ]
}
```

4. **Endpoint: POST /payments**

- **Description:** Initiate a payment for an order.
- **Request:**

```
{
  "order_id": "456",
  "payment_method": "credit_card",
  "payment_details": {
    // Payment details specific to the selected method
  }
}
```

**Response:**

```
{
  "payment_id": "789",
  "status": "processing"
}
```

These endpoint definitions provide a clear understanding of the data to be transferred in JSON format for each request and the expected JSON response for each endpoint. Additionally, the endpoints follow RESTful conventions, making them intuitive and easy to work with for both developers and clients of your microservices.

## Set Up Deployment and Scaling (1p.)

For this task, I definitely will choose “Docker” just because I have experience working with it and I feel myself more prepared and confident by implementing the last task by it. Containerization using Docker allows you to package your microservices and their dependencies into lightweight, portable containers. Each microservice should have its own Docker container.