1. **TAs:** Athira and Shaarvari

Problem Statement: Migrating a monolithic e-commerce application to a microservices architecture
Description: The project's primary objective is to transition an e-commerce web application into a microservices architecture. This web application can be any simple application built with the tech stack you are comfortable with. Each discrete functionality of the application will be converted into a standalone microservice.
To facilitate the migration process, students will explore industry-standard methodologies such as the Strangler Pattern. This pattern involves gradually replacing parts of the existing monolithic application with microservices, ensuring minimal disruption to the overall system while modernizing its architecture.

Reference:
https://www.geeksforgeeks.org/strangler-pattern-in-micro-services-system-design/

Sample File Structure (For a Flask application):

A sample file structure, if using Flask for your application could look like this:

```
ecommerce-website/
│
├── frontend/
│   ├── static/
│   │   ├── css/
│   │   └── img/
│   └── templates/
│        ├── index.html
│        ├── login.html
│        ├── register.html
│        ├── …
│   ├── Dockerfile
│   ├── ...
│
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── models.py
│   │   ├── routes.py
│   │   └── utils.py
│   ├── config.py
│   ├── run.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── ...
│
├── database/ (if not using MySQL/MongoDB)
│
```

```
│── frontend-deployment.yaml
│   ├── backend-deployment.yaml
│   ├── secret.yaml
│   ├── services.yaml
│   └── ...
│
└── README.md
```

Deliverables:

- **Week 1:** Creating the app
  - Create a basic e-commerce CRUD application using any tech stack
  - Develop the initial microservices architecture plan and set up the basic project structure. Define the high-level architecture of the application, identifying the different components or microservices that will make up the system. Consider factors like scalability, maintainability, and ease of deployment.
  - Decide on the communication protocols between microservices, such as RESTful APIs or message queues.
  - Plan how data will be shared and stored across microservices, considering databases, caching layers, etc.
  - Create a basic directory structure for the project, organizing it based on microservices or functional modules.

- **Week 2:** Writing docker file for {frontend, backend, database} or {each functionality will be a microservice}
  - For each microservice, create a Dockerfile that describes the environment and dependencies needed to run the service. Include necessary commands to install dependencies, set up configurations, and start the service within the Docker container.
  - Build Docker images for each microservice using the corresponding Dockerfiles.
  - Test the Docker images locally to ensure they can be started and operate as expected within containers.
  - Push the Docker images to a container registry (e.g., Docker Hub) for easy access during deployment.

- **Week 3:** Deploying the application
  - Prepare the production environment for deploying the application, which may involve setting up servers, networking, and any required infrastructure components. Configure any additional services needed, such as load balancers, databases, or caching layers.
  - Decide on a deployment strategy (Eg. Strangler Pattern), considering factors like zero-downtime deployment, rollback mechanisms, and scalability.
  - Choose a deployment tool or platform suitable for managing containerized applications, such as Kubernetes.

    - Deploy the containerized microservices to the production environment using the chosen deployment strategy. Monitor the deployment process closely, ensuring that each microservice is deployed successfully and is accessible.

2. **TAs:** Srushti and Dhruthi

Problem Statement: Building a Task Management Application with Raft Consensus Algorithm and MySQL.
Description:
The project aims to develop a task management application leveraging the Raft consensus algorithm for consistency and fault tolerance across multiple nodes. MySQL is the backend database to store task data, enabling users to perform CRUD operations on tasks through a user-friendly interface.
Objectives:
- Implement a task management system with the Raft consensus algorithm for distributed coordination.
- Utilise MySQL as the backend database to store task data.
- Ensure fault tolerance and consistency across multiple nodes using Raft.
- Enable users to perform CRUD operations on tasks via a user-friendly interface.

Deliverables:
- **Week 1:** Select a suitable Raft library/framework and define the Raft cluster's architecture, specifying the number of nodes and communication protocols.
- **Week 2:** Complete Raft node implementation, ensuring fault tolerance and consistency with leader election and log replication. Integrate MySQL database for task storage, ensuring replication across the Raft cluster.
- **Week 3:** Implement CRUD operations for task management, ensuring coordination and replication of changes across the cluster and deploying the application.

3. **TAs: Surabhi and Ranjitha.**
   Problem statement: Microservices communication using RabbitMQ
   Description:

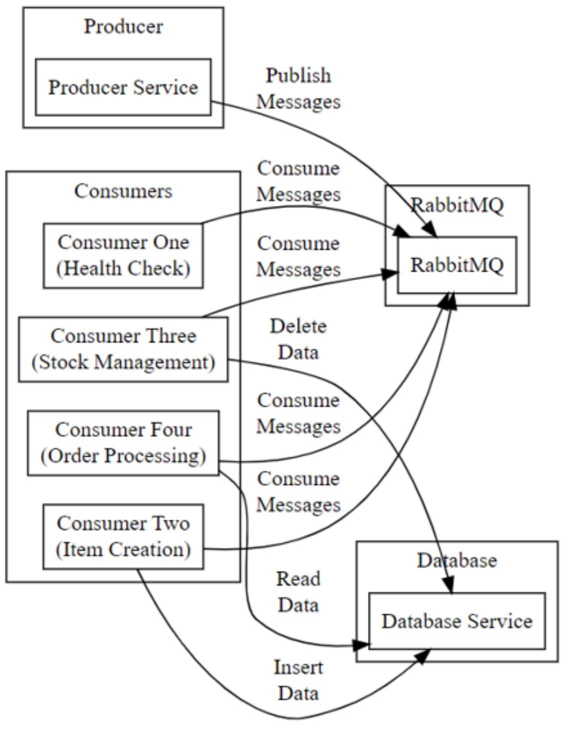
The inventory management system aims to efficiently manage inventory items, track stock levels, and handle orders through a microservices architecture. The system will utilize RabbitMQ for inter-service communication and Docker for containerisation, ensuring scalability, modularity, and ease of deployment.

Objectives:

1. Scalable Architecture: Design a microservices architecture that allows independent development, deployment, and scaling of each component.
2. Robust Communication: Implement communication between microservices using RabbitMQ queues to ensure reliable message passing.

3. Functional Microservices: Develop microservices to handle specific tasks such as health checks, item creation, stock management, and order processing.
4. Database Integration: Integrate a database service to store inventory data and ensure proper CRUD operations.
5. Containerization: Containerize the application using Docker to provide consistency and portability across different environments.
6. Testing and Documentation: Conduct thorough testing to ensure the system functions as expected and provide comprehensive documentation for setup, usage, and maintenance.

Flow diagram:



Deliverables:
Week 1: Setup and Basic Implementation

Setup Docker Environment:
- Create Dockerfiles for each microservice.
- Write a docker-compose.yml file to define the services and network configurations.

- Test Docker setup locally to ensure containers can communicate with each other.

Implement RabbitMQ Communication:

- Set up RabbitMQ container and network.
- Implement basic RabbitMQ communication between producer and consumer services.
- Test message passing between microservices via RabbitMQ queues.

Develop HTTP Servers:

- Choose a suitable web framework (e.g., Flask for Python) for implementing HTTP servers.
- Implement HTTP endpoints for health checks, and CRUD operations related to inventory management.
- Test HTTP servers locally to ensure they are functional.

Week 2: Microservices Development

Producer Service:

- Develop the producer service responsible for constructing queues/exchanges and transferring data to consumers.
- Implement the HTTP server to listen to health_check and CRUD requests.
- Test the producer service to ensure it interacts correctly with RabbitMQ and handles HTTP requests.

Consumer Services:

- Implement consumer services (consumer_one to consumer_four) to handle specific tasks like health checks, item creation, stock management, and order processing.
- Ensure each consumer service can communicate with RabbitMQ and perform its designated actions.
- Test each consumer service individually to verify its functionality.

Week 3: Integration and Testing

Integration Testing:

- Integrate all microservices into the Docker environment.
- Perform end-to-end testing to ensure seamless communication between microservices via RabbitMQ queues.
- Test various scenarios to ensure the proper functioning of the inventory management system.

4. TA's : Aaditya and Sneha

Project: Building a Distributed Key-Value Store with etcd (3 Weeks)

This project guides you through building a simple distributed key-value store using etcd, a popular key-value store for distributed systems. You'll explore basic functionalities and gain experience working with etcd during the 3-week period.

Project Goal: Develop a key-value store application that allows users to set and get key-value pairs using etcd for storage.

Week 1: Setting Up and Understanding etcd

- **Deliverables:**
  - Install and configure a single-node etcd cluster locally.
  - Write a program (Python/Go/Java) to connect to the etcd cluster.
  - Implement functions to list all keys, get the value for a specific key, and put a key-value pair into etcd.
- **Tasks:**
  - Download and install etcd according to your OS instructions (https://github.com/etcd-io/etcd/releases).
  - Follow tutorials or documentation to set up a single-node cluster (https://etcd.io/docs/v3.5/quickstart/).
  - Choose a programming language (Python, Go, Java are good options) and explore client libraries for interacting with etcd (https://etcd.io/docs/v3.4/integrations/).
  - Write functions to:
    - List all keys using etcd client library.
    - Get the value for a specific key provided by the user.
    - Put a key-value pair into etcd, allowing users to specify both key and value.

Week 2: Adding Features and Error Handling

- **Deliverables:**
  - Implement functionality to delete a key-value pair.
  - Incorporate error handling for various operations (e.g., key not found, connection issues).
  - Design a simple user interface (command-line or basic web interface) for interacting with the key-value store.
- **Tasks:**
  - Extend your program to include a delete function that removes a key-value pair based on the provided key.
  - Implement error handling mechanisms to catch potential issues like:
    - Key not found errors when getting or deleting non-existent keys.
    - Connection errors when the etcd cluster is unavailable.

- Design a user interface (text-based command-line or a simple web interface using a framework like Flask/Django) to allow users to:
  - See a list of available options (put, get, delete, list).
  - Provide key and value inputs for put operation.
  - Enter a key for get and delete operations.
  - Display appropriate messages based on the operation's success or failure.

Week 3: Scaling and Testing

- **Deliverables:**
  - Explore running a multi-node etcd cluster (optional).
  - Write unit tests for your program's functionalities.
  - Document your project with explanations and comments in the code.
- **Tasks:**
  - (Optional) Experiment with setting up a multi-node etcd cluster to understand how data is distributed across nodes. You can find instructions in the etcd documentation (https://etcd.io/docs/).
  - Write unit tests for the core functionalities of your program (put, get, delete, list) to ensure they behave as expected under different scenarios.
  - Document your project with comments in the code explaining each function's purpose and overall program logic. You can also create a separate README file outlining the project setup, functionalities, and instructions to run the program.

5. TAs: Phanindra and Nytik

Problem Statement: Back Up service using docker and Kubernetes

Description: Creating a backup service that periodically backs up the contents of a folder to Google Drive using Docker and Kubernetes involves several steps.

In this project, you will work with Docker and Kubernetes to create a Backup service.

Deliverables:

- **Week-1: Containerized Google Drive client**

Here's a high-level technical breakdown:

1. **Set up Google Drive API**:
    - Obtain credentials for the Google Drive API.
    - Use the `google-api-python-client` library to interact with Google Drive.
2. **Create a Docker Container**:
    - Write a `Dockerfile` that includes all necessary dependencies and your backup script.
    - Build the Docker image.
3. **Write the Backup Script**:
    - Develop a script in Python that uses the Google Drive API to upload files.
    - Ensure the script can be triggered at regular intervals.

- **Week-2: Kubernetes Deployment & Orchestration**
    - **Kubernetes CronJob**:
        i. Define a `CronJob` resource in Kubernetes to schedule the backup operation.
        ii. The `CronJob` will run the Docker container at specified intervals.
    - **Persistent Volume Claims (PVC)**:
        i. Use PVCs in Kubernetes to ensure the data you want to back up is accessible to the container running the backup script.
    - **Monitoring and Logging**:
        i. Implement logging to track the backup process.
        ii. Optionally, set up monitoring to alert you in case of failures.
    - **Security Considerations**:
        i. Securely manage API credentials and sensitive data.
        ii. Use Kubernetes secrets to store sensitive information.
    - **Testing and Validation**:
        i. Test the backup process thoroughly to ensure data integrity.
        ii. Validate the recovery process from the backups.

6. TAs: Kumari Shivangi, Chetana Patil

Problem Statement: Building an E-commerce Microservices Application on Cloud using Docker, Kubernetes, Jenkins, and Git

Description: The aim of this project is to develop an ecommerce microservices application that can be deployed on the cloud using Docker, Kubernetes, Jenkins, and Git. The application will consist of several microservices that will be deployed as Docker containers on a Kubernetes cluster. Jenkins will be used for continuous integration and deployment, while Git will be used for version control.

Objectives:

- Design and implement the microservices architecture for the application.
- Create Docker containers for each microservice.
- Use Kubernetes to orchestrate the containers locally.
- Implement a Jenkins pipeline to automate the deployment process.
- Integrate Git with Jenkins to trigger the pipeline on code changes.

Deliverables:

Week 1:
1)Design the Microservices Architecture

- Define the different microservices that will be part of the application.
- Determine the communication protocols between the microservices.
- Plan the data model and schema for the microservices.

2)Develop Microservices

- Develop the different microservices using appropriate programming languages and frameworks.
- Implement REST APIs to allow communication between the microservices.
- The app should contain different modules connected to a database to store data
- For instance, a user page, product page and order page
- User Management: This module handles the registration, authentication, and authorization of users. It allows users to create accounts, login, and manage their profiles.
- Product Management: This module handles the management of products. It allows admins to add, edit, and delete products. It also allows users to view and search for products.
- Order Management: This module handles the management of orders. It allows users to view their order history, track their orders, and manage their orders.
- Review Management (optional): This module handles the management of product reviews. It allows users to view and add reviews for products.

Deliverables: Microservices architecture document and code for microservices.

Week 2:
Containerize and Orchestrate Microservices using Docker and Kubernetes respectively.

1). Containerize Microservices using Docker

- Write Dockerfiles for each microservice.
- Build and test Docker images for each microservice.

Deliverable: Docker images for each microservice.

2).Orchestrate Microservices using Kubernetes

- Create Kubernetes deployment manifests for each microservice.
- Create Kubernetes services for each microservice.
- Test and validate the Kubernetes deployment.

Deliverable: Kubernetes deployment manifests and services.

Week 3:
1) Implement Continuous Integration and Deployment using Jenkins

- Set up Jenkins on a server.
- Create Jenkins jobs and corresponding Jenkinsfile for building, testing, and deploying the microservices.
- Configure Jenkins to monitor the Git repository for changes and trigger builds and deployments automatically.

2)Version Control using Git

- Create a Git repository for the microservices code.
- Commit and push code changes to the Git repository.
- Use Git to manage different versions and branches of the code.

Deliverables: Jenkins jobs and configuration files.