Problem Statement 1:
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 must have the following features -

1) User Login/Registration
2) Product Browsing Page (Display all available products together)
3) Product Landing Page (Description/Price for that particular product)
4) Cart after adding products
5) Checkout/Payment page

Feel free to add more features and customize the layout/appearance etc.

The recommended tech stack is -

1) Flask with MongoDB/Mongo Express
2) React with MongoDB

However, you are free to use your own tech stack for the application.

Each discrete functionality of the application will be converted into a standalone microservice.

To facilitate the migration process, you can explore 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/
|
├── user-management/
|   ├── app.py
|   ├── models.py
|   ├── routes.py
|   ├── Dockerfile
|   └── ...
|
├── product-management/
|   ├── app.py
|   ├── models.py
|   ├── routes.py
|    ├── Dockerfile
|    └── ...
|
├── cart/
|    ├── app.py
|    ├── models.py
|    ├── routes.py
|    ├── Dockerfile
|    └── …
|
├── payment-gateway/
|    ├── app.py
|    ├── models.py
|    ├── routes.py
|    ├── Dockerfile
|    └── ...
|── deployments.yaml
|── secret.yaml
|── services.yaml
|── ...
└── README.md
```

Deliverables:

- **Week 1: Creating the app:**
  - Create a basic e-commerce CRUD application using any tech stack (Flask/React etc.)
  - 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. (You can follow the same pattern as we have given above, or come up with your own microservices. Ensure there are a minimum of three microservices).
  - Decide on the communication protocols between microservices, such as RESTful APIs.
  - Plan how data will be shared and stored across microservices, considering databases, etc.
  - Create a basic directory structure for the project, organizing it based on microservices or functional modules (as shown in the example above).

- **Week 2: Writing docker file for each of your chosen functionalities**
  - 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.
- If using MongoDB and Mongo Express, you can containerise them as separate services as well

- **Week 3: Deploying the application**
  - Use Kubernetes to deploy the application and configure the yaml files to bring the application together.
  - You can use the secret.yaml file to store passwords and other information
  - YAML definitions for multiple components/resources can be bundled together into one YAML file. '---' is used to separate the definitions of the components/resources
  - In services.yaml, specify a nodePort as well. A nodePort exposes the pod to applications outside the cluster. This will allow you to access the frontend admin interface from your host system

General Guidelines for the project:

1. Create a GitHub Repository.
2. Weekly progress according to the problem statement assigned must be pushed to the repo and this will be considered while evaluating.
3. Each team's weekly progress update in the repository must be shown to the teachers in class.