List of Experiments
Create a Kanban Board to Visualize the Tasks. • Create Columns for To Do, In-Progress and Done. • Add Atleast 5 Sample Tasks • Move the Tasks across the Columns to Simulate the Workflow.
Sketch a Simple Prototype of a Bus Ticket Booking System using Figma Tool
Create a Scrum Project in Jira. • Add a backlog with at least 5 items (e.g., "Create user registration page", "Develop API for login"). • Prioritize the backlog and create a 1-week sprint. • Move backlog items into the sprint and start the sprint. • Finally show the Screenshot of the sprint board at the start and end of the sprint.
Use the following requirements for a Library Management System: • Add a feature to search books by title and author. • Implement an online book reservation system. • Generate monthly reports on borrowed books for administrators. • Enable email notifications for overdue books. • Add support for QR code scanning for borrowing and returning books. • Create a user-friendly dashboard for librarians. • Allow users to review and rate books. • Integrate a chatbot for user assistance. • Develop a mobile app version of the system. • Provide multi-language support. Categorize each requirement using MOSCOW Method (Must-Have, Should-Have, Could-Have, or Won't-Have) based on the following criteria: • Impact on the users and stakeholders. • Feasibility considering time, budget, and resource constraints. Finally Submit the completed Google Sheet or Excel file with all requirements categorized and justified.
Link Jira tasks with Confluence to streamline task tracking and progress monitoring for the Library Management System development. • Create a new page in Confluence titled "Library Management System Project Overview." • Embed at least 5 Jira issues related to the development of the Library Management System (e.g., tasks from the sprint like "Develop book search functionality," "Create user login page," etc.). • Use the Jira macro to display issues with status (e.g., "To Do," "In Progress," "Done"). • Add a progress bar in the Confluence page to visually track the completion of each embedded Jira task (e.g., percentage of tasks completed in the sprint). • Submit a screenshot of the Confluence page showing the embedded Jira tasks and the progress bar.

List of Experiments

Create a simple API using Swagger/OpenAPI and test it with Swagger UI.
Tasks:
1. Create an OpenAPI Specification:
   - Use Swagger Editor to define a simple API.
   - Create an API with the following two endpoints:
     - GET /greeting – Returns a simple greeting message (e.g., "Hello, World!").
     - POST /greeting – Accepts a name (e.g., "John") and returns a personalized greeting (e.g., "Hello, John!").

   Specification Details:
   - For GET /greeting:
     - Response: A JSON object with a message (e.g., { "message": "Hello, World!" }).
   - For POST /greeting:
     - Request: A JSON object with a name field (e.g., { "name": "John" }).
     - Response: A personalized greeting in JSON format (e.g., { "message": "Hello, John!" }).
2. Render and Test the API in Swagger UI:
   - Use Swagger UI to visualize and interact with the API.
   - Import the OpenAPI specification you just created into Swagger UI.
   - Test the GET /greeting endpoint to make sure it returns the correct greeting.
   - Test the POST /greeting endpoint by sending a name (e.g., "John") in the request body and check if the API responds with a personalized greeting.
3. Submit Deliverables:
   - Provide a screenshot of the Swagger UI showing both the GET and POST endpoints, including the test results.
   - Submit the OpenAPI specification (as .yaml or .json format).

List of Experiments

Define, document, create and test a simple Book Store API using Swagger/OpenAPI.
Tasks:

1. Create an OpenAPI Specification for the Book Store API:
   - Use Swagger Editor to define an OpenAPI specification for a basic Book Store API.
   - The API should have the following endpoints:
     - GET /books – Retrieves a list of all books.
     - POST /books – Adds a new book to the store.
     - GET /books/{id} – Retrieves a specific book by ID.
     - DELETE /books/{id} – Deletes a specific book by ID.

   Specification Details:
   - GET /books: Should return a list of books in JSON format, with each book having an id, title, author, and price.
   - POST /books: Accepts a JSON object with title, author, and price to add a new book to the store.
   - GET /books/{id}: Accepts a book ID and retrieves the details of the specific book.
   - DELETE /books/{id}: Deletes a book by ID.
2. Render and Test the API in Swagger UI:
   - Use Swagger UI to visualize and interact with the Book Store API based on the OpenAPI specification.
   - Test each endpoint by performing the following actions:
     - Use GET /books to retrieve the list of books.
     - Use POST /books to add a new book (e.g., "The Great Gatsby").
     - Use GET /books/{id} to retrieve details of a specific book.
     - Use DELETE /books/{id} to delete a book.
3. Submit Deliverables:
   - Provide a screenshot or link to the Swagger UI showing the test results for all API endpoints.
   - Submit the OpenAPI specification (as a .yaml or .json file).

Integrate React with a simple REST API to build a To-Do List application.
Tasks:

1. Create a Simple Backend API:
   - Set up a basic Node.js/Express server that provides the following endpoints:
     - GET /todos – Returns a list of to-do items.
     - POST /todos – Creates a new to-do item.
     - DELETE /todos/{id} – Deletes a specific to-do item by ID.
2. Set up React Front-End:
   - Create a new React project.
   - Build components:
     - A TodoList component that fetches and displays to-do items from the API.
     - A NewTodo component with a form to create new to-do items and send the data to the API.
     - A DeleteTodo component to delete to-do items.
3. Connect Front-End and Back-End:
   - Use Axios or Fetch API to make HTTP requests from React to the backend API.
   - Display the to-do items dynamically on the React page, and implement functionality to add and remove items.
4. Test and Document:
   - Test the full flow of the application: Adding a to-do, listing to-dos, and deleting a to-do.
   - Submit a screenshot of the React application in action and provide the code for both the front-end and back-end.

List of Experiments

Use Angular to create a CRUD (Create, Read, Update, Delete) application and connect it to a RESTful API.
Tasks:

1. Create the Backend API:
   - o Set up a Node.js/Express or Python Flask/Django backend that offers the following API endpoints:
     - GET /items – Returns a list of items.
     - POST /items – Adds a new item.
     - PUT /items/{id} – Updates an existing item.
     - DELETE /items/{id} – Deletes an item by ID.
2. Set up Angular Front-End:
   - o Create a new Angular project.
   - o Build components and services:
     - ItemListComponent: Fetches and displays the list of items from the backend.
     - ItemFormComponent: A form to add new items.
     - ItemEditComponent: A form to edit existing items.
     - ItemDeleteComponent: A button or option to delete items.
3. Use Angular Services to Connect with the Backend:
   - o Create an ItemService in Angular to manage HTTP requests to the backend API using HttpClient.
   - o Handle CRUD operations (GET, POST, PUT, DELETE) from the front-end, and display the updated data.
4. Test and Document:
   - o Test the full functionality of the application: Add, view, edit, and delete items.
   - o Submit a screenshot of the Angular application in action and the source code for both the front-end and back-end.

Demonstrate how to work collaboratively in Git/GitHub on a project using the fork-and-pull request workflow.
Tasks:

1. Fork an existing public GitHub repository (e.g., a sample JavaScript or Python project).
2. Clone the forked repository to your local machine using Git.
3. Create a new branch for the feature or change you want to work on.
4. Make modifications or add new features (e.g., add a function, fix a bug, or update the README).
5. Commit your changes and push the branch to GitHub.
6. Go to the GitHub repository and create a pull request to merge your feature branch into the main branch.
7. Review the pull request and provide feedback on the changes.
8. Respond to feedback by making additional commits to the feature branch if necessary.
9. Once the pull request is approved, merge it into the main branch.
10. Finally submit the link to the pull request along with a summary of the changes you made and how you collaborated.

List of Experiments
Demonstrate how to work with Git branches and resolve merge conflicts when collaborating with others. Tasks: 1. Clone a shared repository to your local machine. 2. Create a new branch and switch to it. 3. Make changes to a file (e.g., update a README or modify code in a specific function). 4. Commit your changes. 5. Push the changes to the remote repository. 6. Before merging, pull the latest changes from the main branch. 7. Switch back to your feature branch. 8. Merge the main branch into your feature branch. 9. If there are conflicts, resolve them manually by editing the conflicting files. After resolving conflicts, mark the conflicts as resolved. 10. Commit the resolved merge. 11. Push your feature branch with the merged changes to GitHub and create a pull request. 12. Submit a summary of the steps you performed, the conflicts you encountered, and how you resolved them.
Create a Static Website and Containerize, Build & Serve it using Docker. Tasks: 1. Create a Simple Static Website (index.html file) with basic HTML content. 2. Write/create a Dockerfile to serve the website using Nginx. 3. Build the Docker Image 4. Run the container: 5. Access the Website using a Browser
Create a Simple Python Flask API, Containerize the Application, Build & Push the Image using Docker and Deploy the Application using Kubernetes. Tasks: 1. Create a Simple Flask API by writing a Python file (app.py) with basic endpoints. 2. Containerize the Flask App using Dockerfile. 3. Build the Image using Docker. 4. Push the Image to Docker Hub. 5. Create Kubernetes manifests (Deployment YAML & Service YAML) to deploy the application. 6. Apply the Manifests and Access the API via NodePort.
Implement TDD to ensure high-quality, testable code. Tasks: 1.Select a feature or functionality to develop (e.g., a simple calculator). 2.Write the first failing unit test using a testing framework (e.g., JUnit for Java, PyTest for Python). 3.Implement the minimum code to pass the test. 4.Refactor the code, ensuring tests continue to pass. 5.Add additional tests and repeat the cycle.
Set up a CI/CD pipeline to automate the building, testing, and deployment of a containerized application. Tasks: 1.Set up Jenkins on a local machine or server. 2.Create a Dockerfile to containerize a sample application. 3.Write a Jenkinsfile to automate the process of building the Docker container, running tests, and deploying to a cloud platform (e.g., AWS or GCP). 4.Configure Jenkins to trigger builds upon code commits or pull requests.

List of Experiments
Implement Continuous Deployment using GitHub Actions to deploy a Dockerized application. Tasks: 1.Set up a GitHub repository and push a simple Dockerized application. 2.Create a GitHub Actions workflow to automatically build and push the Docker image to Docker Hub or GitHub Container Registry. 3.Automate deployment to a cloud platform (e.g., AWS ECS, Azure Kubernetes Service, or Google Kubernetes Engine). 4.Test the CI/CD pipeline by pushing new code changes and verifying that the deployment occurs automatically.
A software company needs to gather requirements for a new web application from multiple departments (marketing, sales, customer service). The focus group method is chosen to collect insights.
The stakeholders have conflicting views on the user interface design for a new mobile app. A prototype is created, and the stakeholders will test and give feedback.
You are managing a product backlog for an e-commerce application. The product team must prioritize features to develop based on limited resources using MoSCoW prioritization.
A mobile app development team wants to analyze user feedback to understand which features will drive customer satisfaction and which are must-haves for the app’s success.
Two key stakeholders want to implement conflicting requirements in a new system. You are tasked with resolving these conflicts while maintaining the integrity of the project scope.
You are tasked with migrating a monolithic e-commerce application to a microservices-based architecture to improve scalability and maintainability. Each service should be developed, deployed, and scaled independently.

List of Experiments
A team is tasked with automating the deployment, scaling, and management of a containerized application across multiple clusters. They need to use Kubernetes for orchestration and ensure high availability.
You are tasked with developing an API for a mobile application with a focus on scalability and documentation. You need to create and manage API endpoints before development begins, ensuring clear contracts with the front-end team.
A team needs to deploy multiple microservices that will run in different environments (development, staging, and production). The microservices need to be containerized for easier deployment and scalability.
The organization's microservices need to handle traffic spikes during peak business hours. The team needs to configure Kubernetes to scale the services horizontally to meet demand.