# CI/CD Pipeline with GitHub Actions & Docker- Project Report

**Introduction**

Software development has evolved to focus on speed, reliability, and automation. Traditional manual deployment processes are slow and error-prone. Continuous Integration and Continuous Deployment (CI/CD) address this challenge by automating testing, building, and deployment. In this project, a CI/CD pipeline was implemented using **GitHub Actions, Docker, and a local Virtual Machine (VM)**. The objective was to create a workflow that ensures the code is tested, containerized, and deployed automatically without depending on external cloud platforms.

---

**Abstract**

This project presents a simple yet complete CI/CD pipeline for a Flask-based Python web application. The process begins when the developer pushes code to GitHub. GitHub Actions then automatically runs **unit tests with pytest** to confirm code quality. If the tests succeed, the pipeline builds a Docker image and pushes it to **Docker Hub**. The final stage involves pulling the image onto a local VM and running the application in a Docker container.

The workflow demonstrates how DevOps practices can be applied even in a local environment, showcasing the power of automation, containerization, and repeatable deployments. It also provides a foundational understanding of CI/CD concepts for beginners.

---

**Tools Used**

- **GitHub Actions** – Automates CI/CD workflows including testing, building, and pushing images.
- **Docker** – Provides containerization for consistent application runtime.
- **Docker Hub (Free)** – Cloud registry for storing and sharing Docker images.
- **Flask** – Lightweight Python framework for the sample web app.
- **Pytest** – Unit testing framework used to validate functionality.
- **Ubuntu VM (VirtualBox)** – Local environment for deploying and testing the containerized app.

---

**Steps Involved in Building the Project**

1. **Application Development**
    - Built a basic Flask app (app.py) returning a test message.
    - Created requirements.txt to list dependencies for reproducible installations.
2. **Testing Setup**

    - Added a tests/ folder containing test_app.py.
    - Tests verified whether the root endpoint (/) responded correctly.

3. **Containerization**
    - Designed a Dockerfile to package the app into an image.
    - Used docker build and docker run to test the container locally.
    - Added optional docker-compose.yml for simplified local setup.
4. **CI/CD Workflow with GitHub Actions**
    - Configured .github/workflows/ci-cd.yml.
    - Automated jobs included:
        - Installing dependencies.
        - Running pytest for validation.
        - Building Docker image if tests passed.
        - Pushing the image to Docker Hub.
5. **Local Deployment on VM**
    - Pulled the pushed image from Docker Hub.
    - Deployed the app on Ubuntu VM using Docker.
    - Verified successful deployment at http://localhost:5000.
6. **Documentation and Verification**
    - Collected pipeline run results, logs, and application output screenshots.
    - Stored them in the screenshots/ folder for reference.

---

**Conclusion**

The project demonstrates the **complete lifecycle of CI/CD**: code → test → build → push → deploy. By integrating GitHub Actions with Docker, the pipeline ensures reliability and consistency in every build. Running tests before deployment prevents faulty code from being shipped. Docker guarantees that the application runs identically in any environment, while Docker Hub simplifies distribution.

Even without external cloud platforms, this project proves that **a professional DevOps pipeline can be achieved entirely in a local environment**. It provides a strong foundation for understanding modern DevOps practices and can be extended further by adding databases, monitoring tools, or cloud deployments in the future.