DevOps Project

DevOps Project On: CI/CD PROJECT WITH GitHub ACTIONS

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)
Dundigal, Hyderabad – 500 043, Telangana

Department of CSE
Bachelor of Technology
in
CSE

-BY

YARLAGADDA SAI KETHAN
23951A058C

DONIKALA SAI TEJA
23951A058N

KAMMARI SAI CHARAN
23951A0586

DECLARATION

I certify that

a. The work contained in this report is original and has been done by me under the guidance of my supervisor (s).
b. The work has not been submitted to any other Institute for any degree or diploma.
c. I have followed the guidelines provided by the Institute for preparing the report.
d. I have conformed to the norms and guidelines given in the Code of Conduct of the Institute.
e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the report and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Place: Hyderabad **Signature of the Student**

Date: **Roll No:**

ABSTRACT

In today's fast-paced software development landscape, automation and rapid delivery have become essential to maintaining competitiveness and ensuring high-quality software products. This project explores the implementation of a **CI/CD (Continuous Integration/Continuous Deployment) pipeline using GitHub Actions**, a modern DevOps tool integrated within the GitHub platform.

The primary objective of this project is to automate the process of building, testing, and deploying software applications with minimal manual intervention. By leveraging GitHub Actions, developers can define workflows using YAML configurations that are triggered by events such as code pushes or pull requests. These workflows automate critical development stages—ensuring that each code change is properly validated and seamlessly deployed to production environments.

The methodology adopted in this project reflects key DevOps principles: automation, collaboration, continuous feedback, and incremental delivery. From setting up version control with Git, to configuring CI/CD pipelines, and deploying applications using containerization tools like Docker (optional), the project provides a comprehensive hands-on implementation of end-to-end DevOps practices.

The outcome of this project demonstrates how GitHub Actions can significantly streamline the software development lifecycle. It enhances team productivity, reduces integration issues, accelerates deployment cycles, and promotes a culture of continuous improvement. This project not only serves as a practical guide for implementing CI/CD pipelines but also highlights the broader impact of DevOps in achieving agile, scalable, and reliable software delivery.

CONTENTS

CHAPTER 1
INTRODUCTION

1.1 About CI/CD Pipelining

In modern software development, the demand for faster delivery and improved software quality has made automation a necessity. Continuous Integration and Continuous Deployment (CI/CD) are software engineering practices designed to achieve these goals.

Continuous Integration (CI) involves the frequent integration of code changes into a shared repository, where automated builds and tests are run. The primary objective is to identify integration issues early, making them easier and faster to resolve. Developers commit code frequently—often several times a day—and each commit triggers the CI process to validate the codebase.

Continuous Deployment (CD) is the next phase after integration. It automates the delivery of applications to selected environments, such as staging or production, without manual intervention. This means that once code passes the CI phase, it is automatically released into production, leading to faster innovation and customer feedback.
Together, CI/CD bridges the gap between development and operations, streamlining software delivery. GitHub Actions enhances this process by offering native support for automation directly within the GitHub platform. With GitHub Actions, developers can write simple YAML configuration files to automate tasks such as running tests, building applications, deploying code, and sending notifications. These workflows are event-driven, allowing for flexible and efficient automation pipelines.
This project focuses on creating a CI/CD pipeline using GitHub Actions, enabling developers to build, test, and deploy their applications with minimal manual effort.

1.2 Requirements

In order to successfully implement a CI/CD pipeline using GitHub Actions, certain tools, environments, and configurations must be in place. These requirements can be categorized into **software dependencies**, **codebase prerequisites**, and **platform configurations**. Below is a detailed breakdown of what is needed:

1.2.1 GitHub Repository

A GitHub repository is the central component of this project. It serves as the version-controlled storage space where the application code resides. This is also where GitHub Actions workflows are stored and executed. A repository can be either public or private, depending on the scope and visibility preferences of the developer or team.

The repository should contain:

- The source code of the application (e.g., a Python, Node.js, or Java app)
- A .github/workflows directory where YAML workflow files will reside
- Any configuration files necessary for testing and deployment (e.g., Dockerfile, requirements.txt, .env files, etc.)

1.2.2 GitHub Actions

GitHub Actions is a built-in feature in GitHub that enables automation. It allows developers to write custom workflows that are triggered by events such as code commits, pull requests, or the creation of a release. GitHub Actions runs on runners (virtual machines) that can execute commands in Linux, Windows, or macOS environments.

Key GitHub Actions features needed for the pipeline:

- Workflow configuration files written in YAML
- Predefined or custom GitHub Action steps (e.g., actions/checkout, actions/setup-node)

1.2.3 Application Codebase

The codebase serves as the content being tested, built, and deployed. It is crucial to ensure the code is structured and includes all necessary files for automation. If the application includes a backend and frontend, both parts should be well integrated.
Typical requirements for the codebase:

- Valid and working source code
- Automated test scripts (e.g., unit tests using PyTest, Jest, etc.)
- Build configuration (like package.json, Makefile, or setup.py)

1.2.4 Docker (Optional but Recommended)

If the application is to be containerized, Docker is required. Docker allows applications to be packaged into containers along with all their dependencies, making them portable and consistent across different environments.
Requirements related to Docker:

- A valid Dockerfile to build the container image
- Docker CLI installed locally for manual testing
- Docker Hub or another container registry for storing and distributing images

1.3 Prerequisites

Before embarking on the implementation of a CI/CD pipeline using GitHub Actions, it is essential to ensure that certain foundational knowledge, tools, and configurations are in place. These prerequisites help in understanding the workflow and allow for smoother execution of tasks related to continuous integration and continuous deployment.
The prerequisites for this project fall into three major categories: **technical knowledge**, **tooling setup**, and **application readiness**.

1.3.1 Technical Knowledge

To effectively design and manage a CI/CD pipeline, developers should have a foundational understanding of the following concepts:

- **Version Control with Git**: A good grasp of Git commands (clone, commit, push, pull, branch, merge) and the ability to navigate GitHub repositories are crucial. Since GitHub Actions is tied directly to GitHub repositories, familiarity with Git operations is a must.
- **Basic YAML Syntax**: GitHub Actions workflows are written in YAML. Understanding YAML structure, indentation, key-value pairs, and basic logic (if, jobs, steps) is necessary to avoid configuration errors.
- **Continuous Integration Concepts**: Knowledge of how automated testing and building help ensure software quality during frequent code changes is important.

1.3.2 Tooling Setup

To build and test the CI/CD pipeline locally and on GitHub, the following tools should be installed and configured:

- **Git**: Installed on the local development machine to interact with GitHub repositories. (Download Git)
- **Text Editor or IDE**: Tools like Visual Studio Code, PyCharm, or IntelliJ IDEA are useful for editing code, writing tests, and editing workflow files.
- **GitHub Account**: A personal GitHub account is required to host the repository and access GitHub Actions. Basic familiarity with GitHub features like issues, pull requests,

and repositories is beneficial.

1.3.3 Application Readiness

The application that is to be integrated into the pipeline must be prepared and structured correctly. This includes:

- **Proper Project Structure**: The application should have a clear folder structure, with separate directories for source code, tests, and configuration files.
- **Automated Tests**: At least a basic set of automated unit or integration tests should be written. These tests ensure that changes do not break existing functionality and are executed automatically in the CI stage.

CHAPTER 2

METHODOLGY

DevOps is a cultural and technical movement that aims to unify software development (Dev) and IT operations (Ops). The primary goal of DevOps is to shorten the software development life cycle while delivering features, fixes, and updates frequently, reliably, and in close alignment with business objectives.

The methodology used in this project is rooted in DevOps practices, especially those related to automation, continuous feedback, and collaborative development. The CI/CD pipeline built with GitHub Actions is a real-world implementation of DevOps ideology—focusing on **continuous integration**, **continuous delivery**, and **continuous deployment**.

This project follows several core DevOps principles in its methodology:

1. **Automation**: Manual tasks such as testing, building, and deploying are automated through GitHub Actions workflows.

2. **Collaboration**: All code, including automation scripts and configurations, is version-controlled and accessible to team members via GitHub.

3. **Continuous Integration (CI)**: Every code commit triggers an automated process that builds the application and runs tests.

4. **Continuous Deployment (CD)**: When CI is successful, the new version is automatically deployed to a server or cloud environment.

5. **Monitoring and Feedback**: Logs and status checks from each pipeline step provide immediate feedback to developers.

Phases of the Methodology

The CI/CD pipeline implementation in this project can be broken down into several interconnected phases, all aligned with DevOps workflow:

1. Code Development and Version Control

The process begins with developers writing application code using a modular and testable architecture. Code is regularly committed to a GitHub repository using Git. Branching strategies (e.g., feature, develop, and main) may be used to maintain clean integration workflows.

Tools involved:

- Git (for version control)
- GitHub (for hosting the repository and managing collaboration)

2. Workflow Configuration with GitHub Actions

The .github/workflows/ci.yml file is created in the repository to define the CI/CD pipeline using **YAML syntax**. This file contains instructions for:

- The triggering event (e.g., push, pull request)
- The jobs to run (e.g., test, build, deploy)
- The sequence of steps in each job

3. Continuous Integration

Whenever a developer pushes code to the repository, GitHub Actions automatically triggers the **CI workflow**. This workflow typically includes:

- **Dependency Installation**: Installing project dependencies (e.g., via pip, npm, or maven)
- **Linting and Static Code Analysis**: Ensuring code quality and formatting
- **Unit Testing**: Running automated tests to validate new changes

4. Continuous Delivery and Deployment

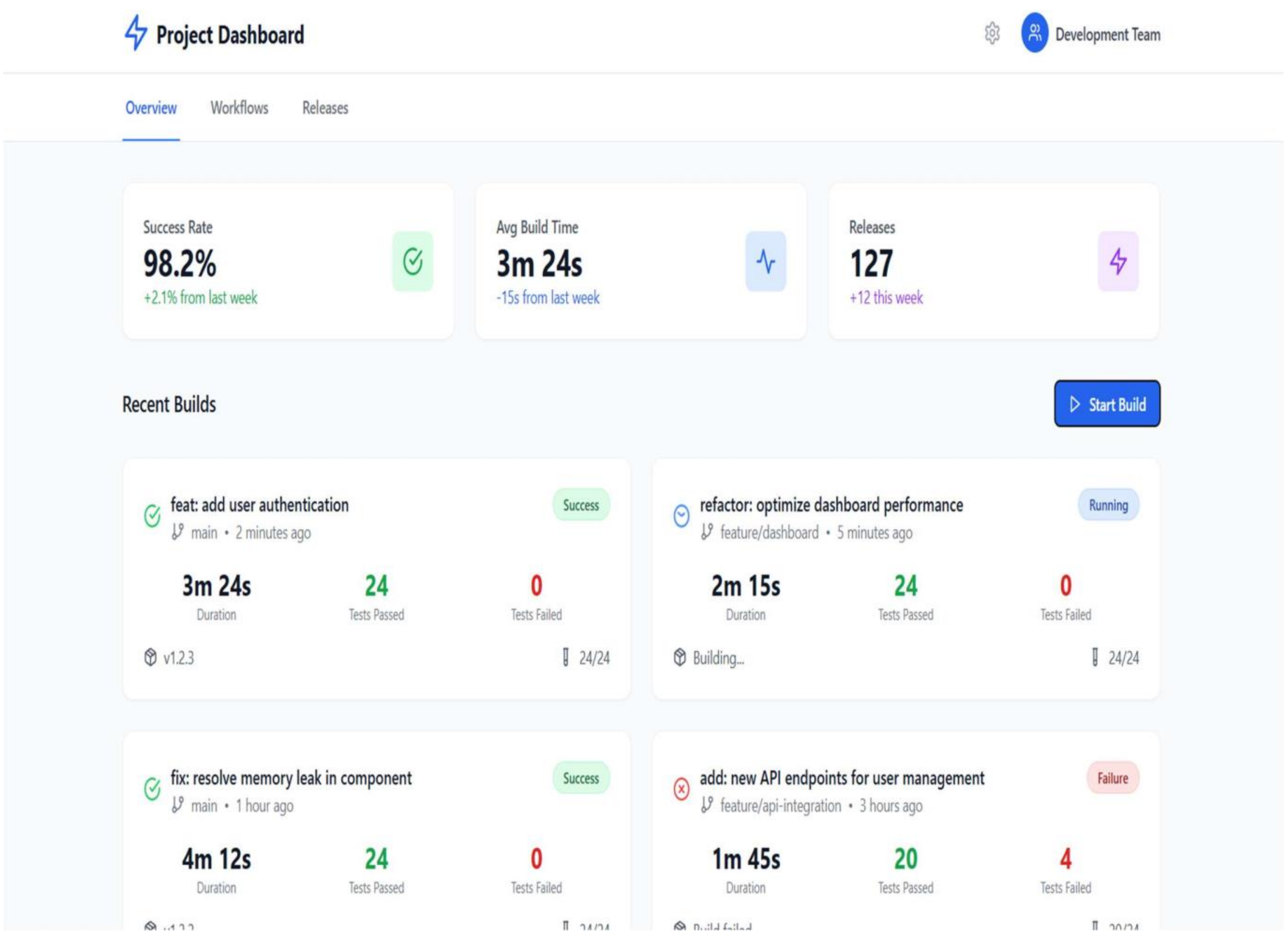
Once the CI phase is complete and successful, the pipeline moves to **CD**, where the application is built (e.g., compiled or containerized) and deployed to a designated environment.

Depending on the project scope, the deployment target can be:

- A static hosting service (e.g., GitHub Pages)
- A cloud-based PaaS (e.g., Heroku, AWS Elastic Beanstalk)
- A container registry and orchestration platform (e.g., Docker Hub + Kubernetes)

CHAPTER 3
RESULTS AND DISCUSSION

Execution:



Project Dashboard

Development Team

Overview Workflows Releases

Success Rate
98.2%
+2.1% from last week

Avg Build Time
3m 24s
-15s from last week

Releases
127
+12 this week

Recent Builds

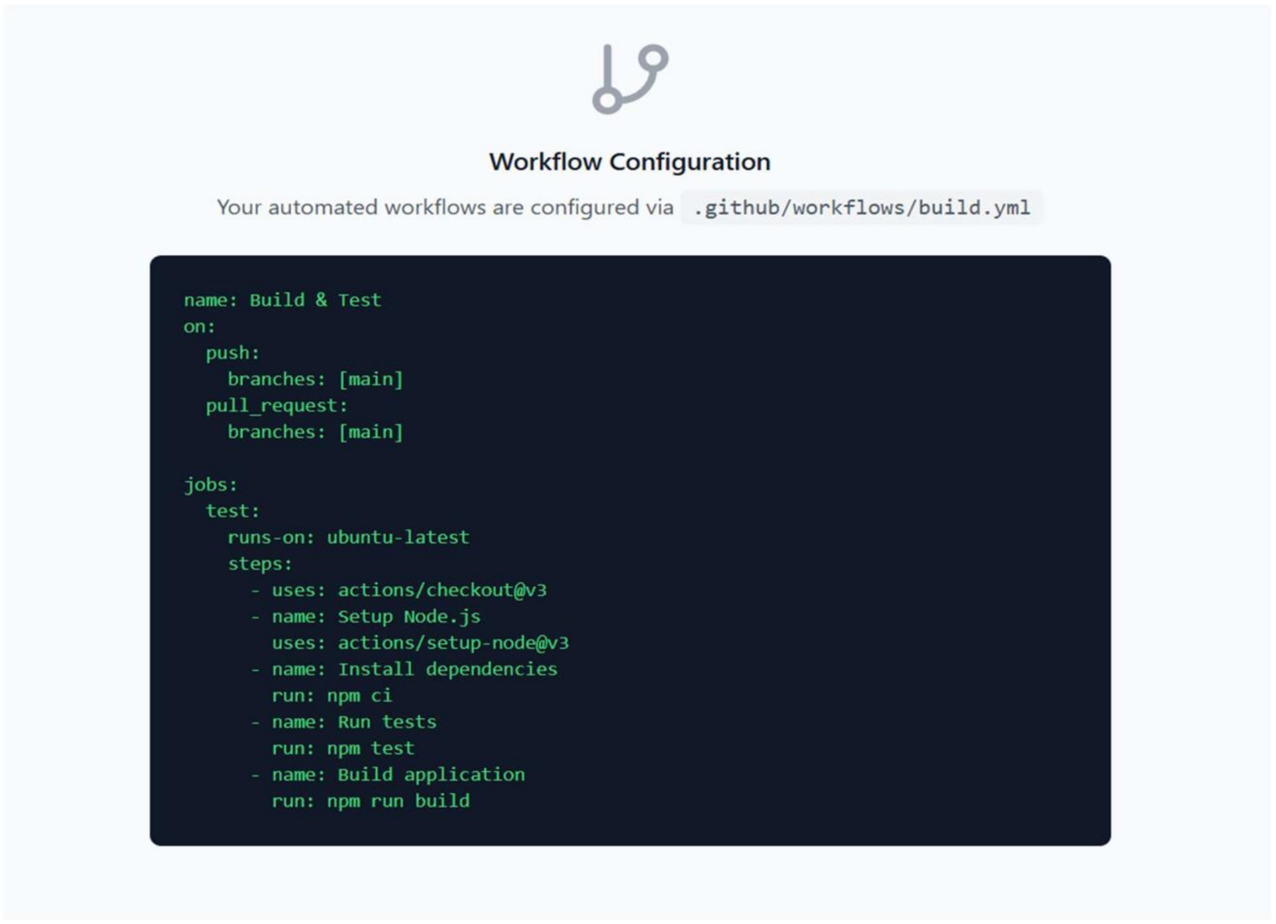
Start Build

feat: add user authentication — Success
main • 2 minutes ago
3m 24s Duration | 24 Tests Passed | 0 Tests Failed
v1.2.3 | 24/24

refactor: optimize dashboard performance — Running
feature/dashboard • 5 minutes ago
2m 15s Duration | 24 Tests Passed | 0 Tests Failed
Building... | 24/24

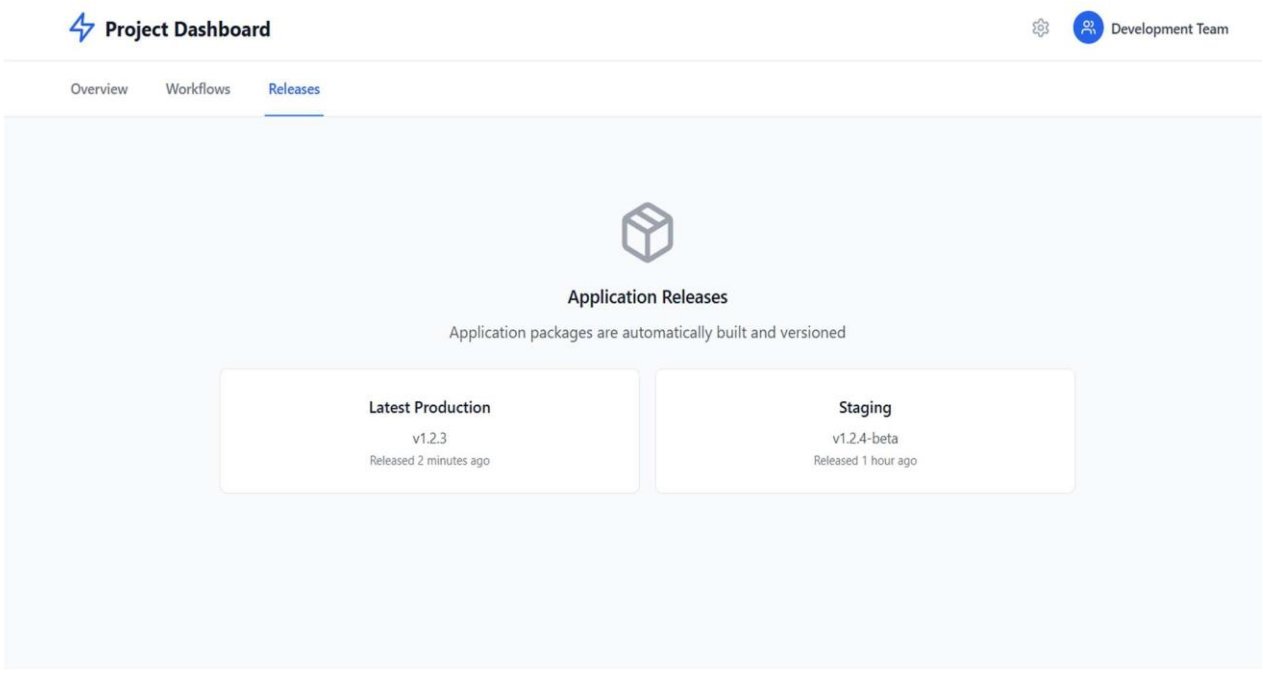
fix: resolve memory leak in component — Success
main • 1 hour ago
4m 12s Duration | 24 Tests Passed | 0 Tests Failed

add: new API endpoints for user management — Failure
feature/api-integration • 3 hours ago
1m 45s Duration | 20 Tests Passed | 4 Tests Failed



Your automated workflows are configured via `.github/workflows/build.yml`

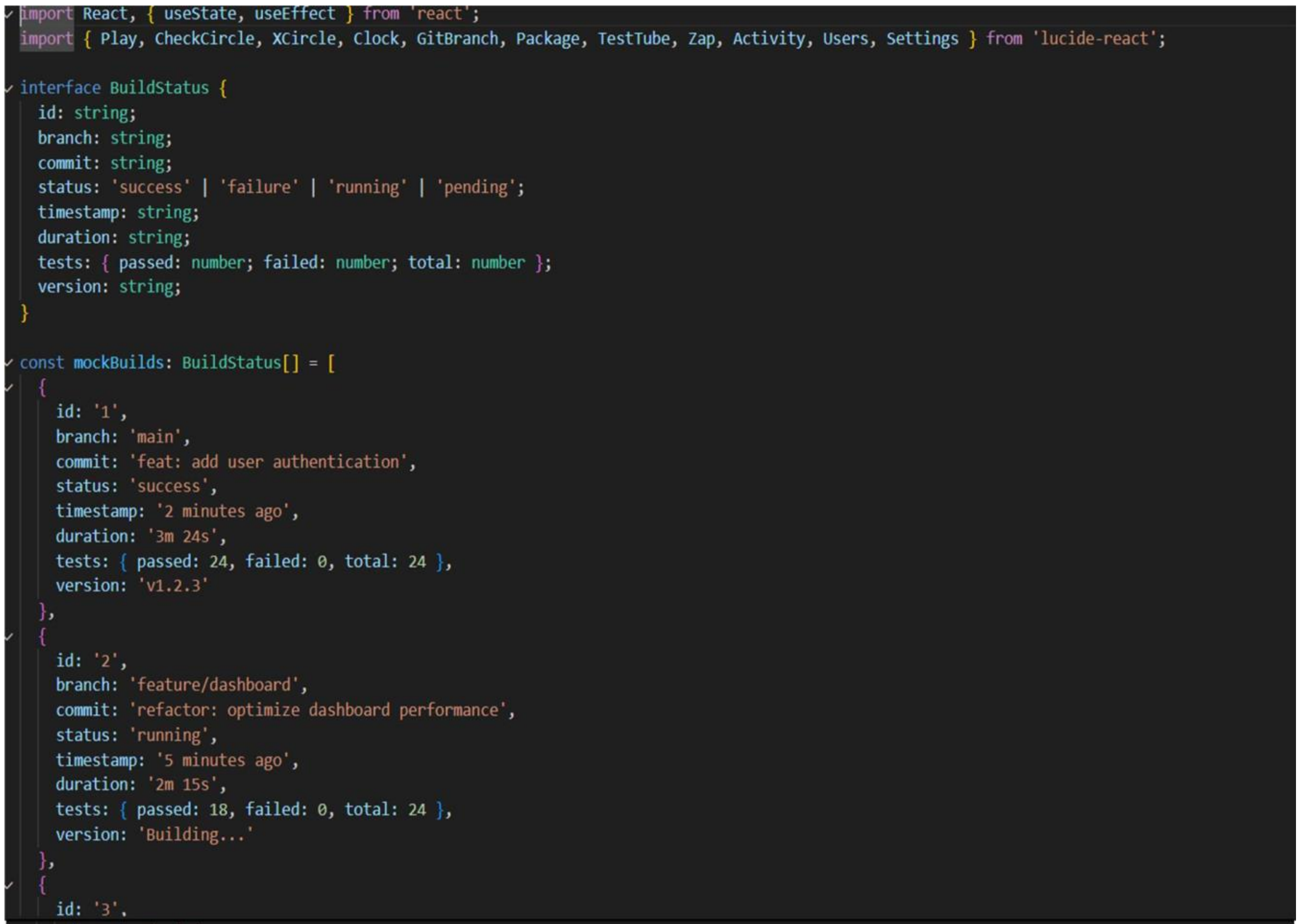
```
name: Build & Test
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
      - name: Build application
        run: npm run build
```



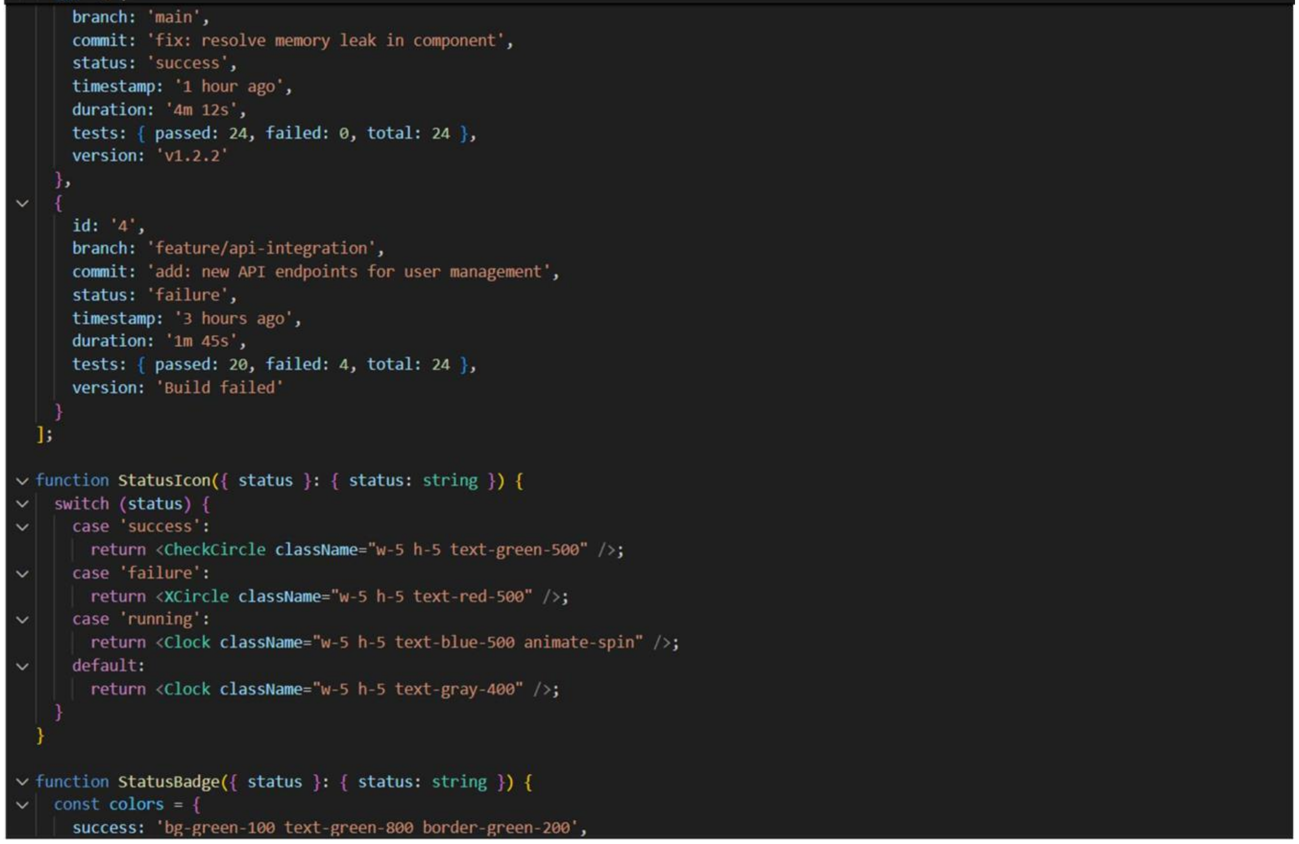
Application packages are automatically built and versioned

Latest Production	Staging
v1.2.3	v1.2.4-beta
Released 2 minutes ago	Released 1 hour ago

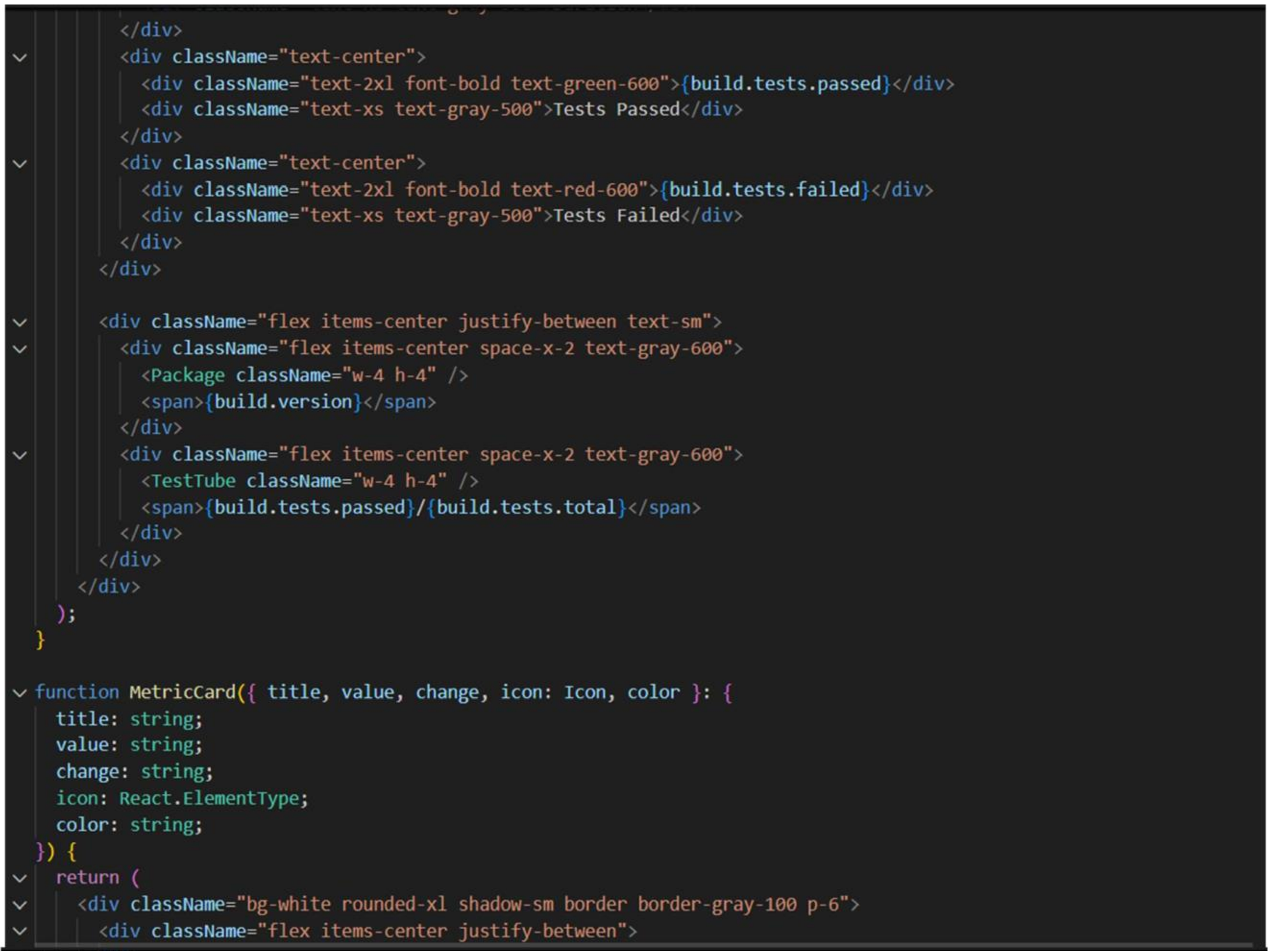
Source Code:



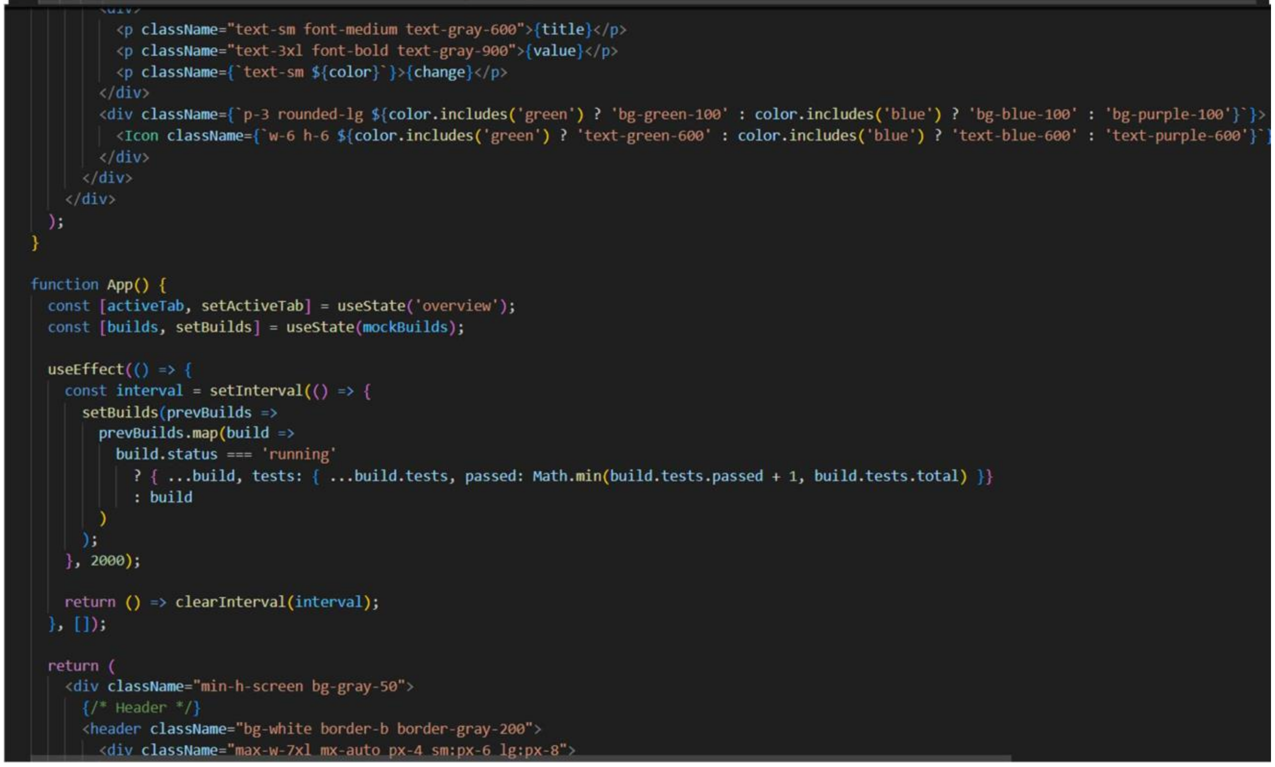
```
import React, { useState, useEffect } from 'react';
import { Play, CheckCircle, XCircle, Clock, GitBranch, Package, TestTube, Zap, Activity, Users, Settings } from 'lucide-react';

interface BuildStatus {
  id: string;
  branch: string;
  commit: string;
  status: 'success' | 'failure' | 'running' | 'pending';
  timestamp: string;
  duration: string;
  tests: { passed: number; failed: number; total: number };
  version: string;
}

const mockBuilds: BuildStatus[] = [
  {
    id: '1',
    branch: 'main',
    commit: 'feat: add user authentication',
    status: 'success',
    timestamp: '2 minutes ago',
    duration: '3m 24s',
    tests: { passed: 24, failed: 0, total: 24 },
    version: 'v1.2.3'
  },
  {
    id: '2',
    branch: 'feature/dashboard',
    commit: 'refactor: optimize dashboard performance',
    status: 'running',
    timestamp: '5 minutes ago',
    duration: '2m 15s',
    tests: { passed: 18, failed: 0, total: 24 },
    version: 'Building...'
  },
  {
    id: '3',
```



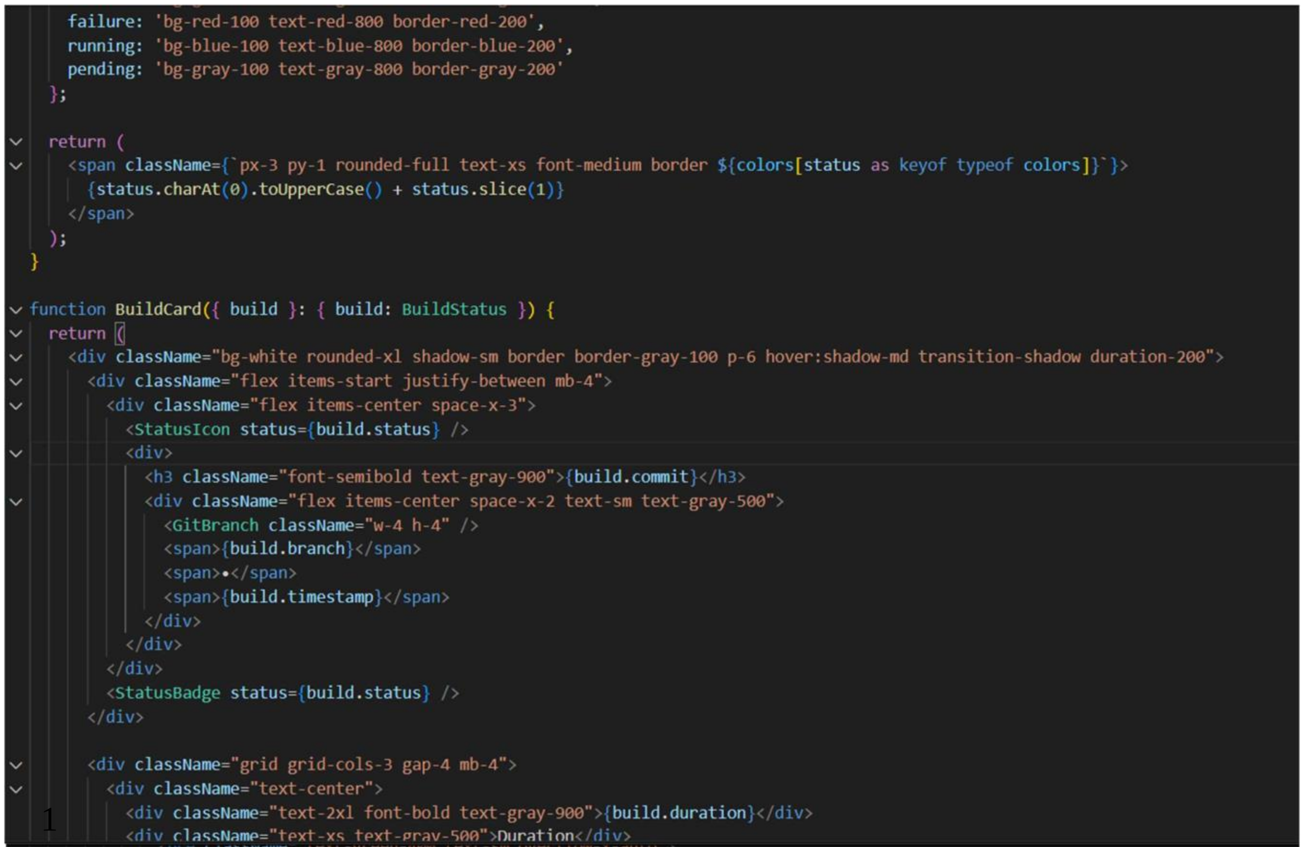
```
    branch: 'main',
    commit: 'fix: resolve memory leak in component',
    status: 'success',
    timestamp: '1 hour ago',
    duration: '4m 12s',
    tests: { passed: 24, failed: 0, total: 24 },
    version: 'v1.2.2'
  },
  {
    id: '4',
    branch: 'feature/api-integration',
    commit: 'add: new API endpoints for user management',
    status: 'failure',
    timestamp: '3 hours ago',
    duration: '1m 45s',
    tests: { passed: 20, failed: 4, total: 24 },
    version: 'Build failed'
  }
];

function StatusIcon({ status }: { status: string }) {
  switch (status) {
    case 'success':
      return <CheckCircle className="w-5 h-5 text-green-500" />;
    case 'failure':
      return <XCircle className="w-5 h-5 text-red-500" />;
    case 'running':
      return <Clock className="w-5 h-5 text-blue-500 animate-spin" />;
    default:
      return <Clock className="w-5 h-5 text-gray-400" />;
  }
}

function StatusBadge({ status }: { status: string }) {
  const colors = {
    success: 'bg-green-100 text-green-800 border-green-200',
```



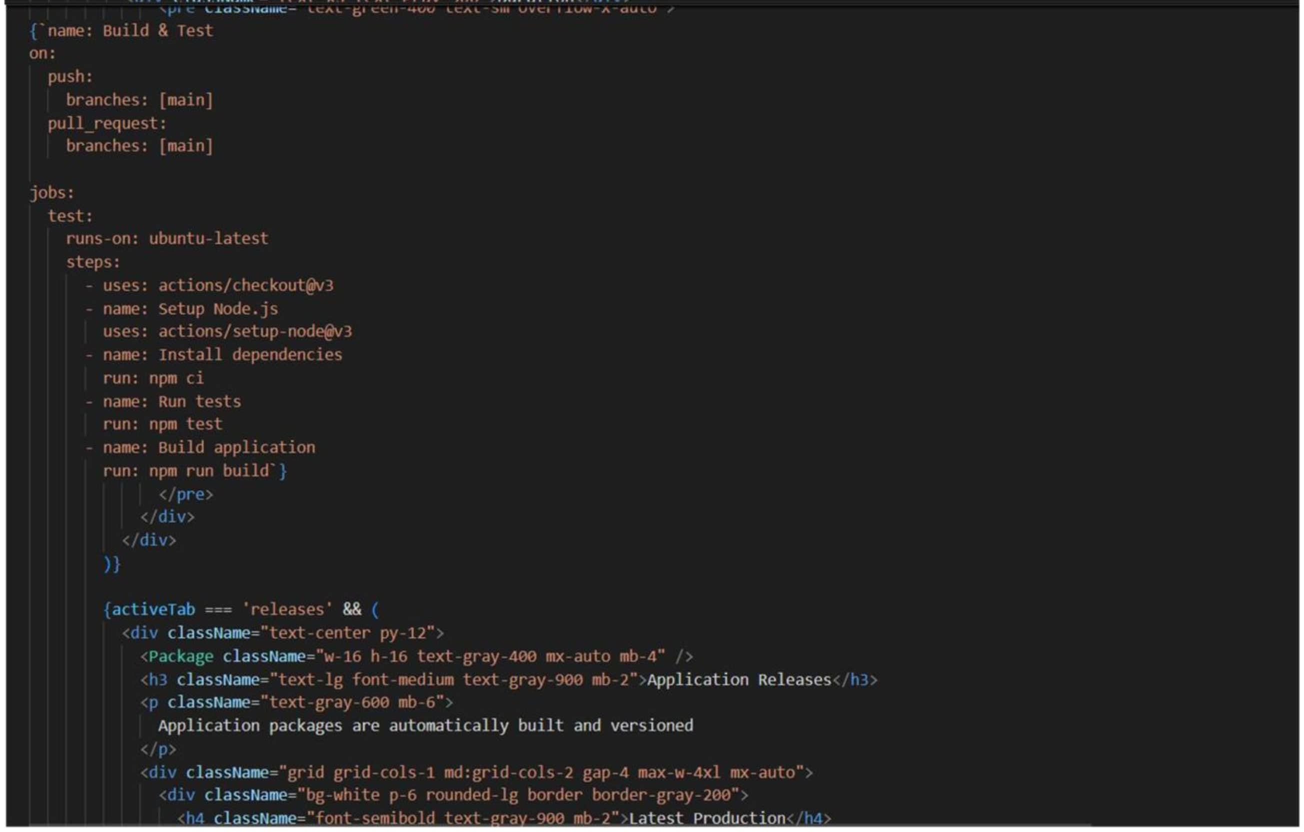
```
          </div>
          <div className="text-center">
            <div className="text-2xl font-bold text-green-600">{build.tests.passed}</div>
            <div className="text-xs text-gray-500">Tests Passed</div>
          </div>
          <div className="text-center">
            <div className="text-2xl font-bold text-red-600">{build.tests.failed}</div>
            <div className="text-xs text-gray-500">Tests Failed</div>
          </div>
        </div>

        <div className="flex items-center justify-between text-sm">
          <div className="flex items-center space-x-2 text-gray-600">
            <Package className="w-4 h-4" />
            <span>{build.version}</span>
          </div>
          <div className="flex items-center space-x-2 text-gray-600">
            <TestTube className="w-4 h-4" />
            <span>{build.tests.passed}/{build.tests.total}</span>
          </div>
        </div>
      </div>
    );
}

function MetricCard({ title, value, change, icon: Icon, color }: {
  title: string;
  value: string;
  change: string;
  icon: React.ElementType;
  color: string;
}) {
  return (
    <div className="bg-white rounded-xl shadow-sm border border-gray-100 p-6">
      <div className="flex items-center justify-between">
```



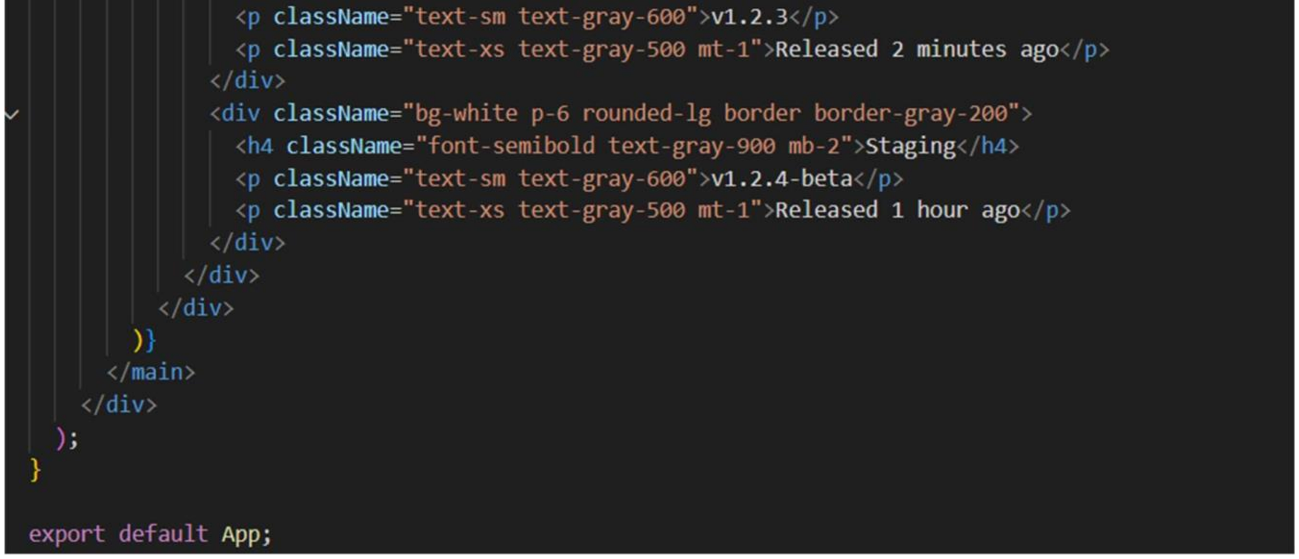
```
          <p className="text-sm font-medium text-gray-600">{title}</p>
          <p className="text-3xl font-bold text-gray-900">{value}</p>
          <p className={`text-sm ${color}`}>{change}</p>
        </div>
        <div className={`p-3 rounded-lg ${color.includes('green') ? 'bg-green-100' : color.includes('blue') ? 'bg-blue-100' : 'bg-purple-100'}`}>
          <Icon className={`w-6 h-6 ${color.includes('green') ? 'text-green-600' : color.includes('blue') ? 'text-blue-600' : 'text-purple-600'}`
        </div>
      </div>
    </div>
  );
}

function App() {
  const [activeTab, setActiveTab] = useState('overview');
  const [builds, setBuilds] = useState(mockBuilds);

  useEffect(() => {
    const interval = setInterval(() => {
      setBuilds(prevBuilds =>
        prevBuilds.map(build =>
          build.status === 'running'
            ? { ...build, tests: { ...build.tests, passed: Math.min(build.tests.passed + 1, build.tests.total) }}
            : build
        )
      );
    }, 2000);

    return () => clearInterval(interval);
  }, []);

  return (
    <div className="min-h-screen bg-gray-50">
      {/* Header */}
      <header className="bg-white border-b border-gray-200">
        <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
```



```
    failure: 'bg-red-100 text-red-800 border-red-200',
    running: 'bg-blue-100 text-blue-800 border-blue-200',
    pending: 'bg-gray-100 text-gray-800 border-gray-200'
  };

  return (
    <span className={`px-3 py-1 rounded-full text-xs font-medium border ${colors[status as keyof typeof colors]}`}>
      {status.charAt(0).toUpperCase() + status.slice(1)}
    </span>
  );
}

function BuildCard({ build }: { build: BuildStatus }) {
  return (
    <div className="bg-white rounded-xl shadow-sm border border-gray-100 p-6 hover:shadow-md transition-shadow duration-200">
      <div className="flex items-start justify-between mb-4">
        <div className="flex items-center space-x-3">
          <StatusIcon status={build.status} />
          <div>
            <h3 className="font-semibold text-gray-900">{build.commit}</h3>
            <div className="flex items-center space-x-2 text-sm text-gray-500">
              <GitBranch className="w-4 h-4" />
              <span>{build.branch}</span>
              <span>•</span>
              <span>{build.timestamp}</span>
            </div>
          </div>
        </div>
        <StatusBadge status={build.status} />
      </div>

      <div className="grid grid-cols-3 gap-4 mb-4">
        <div className="text-center">
          <div className="text-2xl font-bold text-gray-900">{build.duration}</div>
          <div className="text-xs text-gray-500">Duration</div>
```



```
{`name: Build & Test
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
      - name: Build application
        run: npm run build`}
                </pre>
              </div>
            </div>
          )}

          {activeTab === 'releases' && (
            <div className="text-center py-12">
              <Package className="w-16 h-16 text-gray-400 mx-auto mb-4" />
              <h3 className="text-lg font-medium text-gray-900 mb-2">Application Releases</h3>
              <p className="text-gray-600 mb-6">
                Application packages are automatically built and versioned
              </p>
              <div className="grid grid-cols-1 md:grid-cols-2 gap-4 max-w-4xl mx-auto">
                <div className="bg-white p-6 rounded-lg border border-gray-200">
                  <h4 className="font-semibold text-gray-900 mb-2">Latest Production</h4>
```



```
            <p className="text-sm text-gray-600">v1.2.3</p>
            <p className="text-xs text-gray-500 mt-1">Released 2 minutes ago</p>
          </div>
          <div className="bg-white p-6 rounded-lg border border-gray-200">
            <h4 className="font-semibold text-gray-900 mb-2">Staging</h4>
            <p className="text-sm text-gray-600">v1.2.4-beta</p>
            <p className="text-xs text-gray-500 mt-1">Released 1 hour ago</p>
          </div>
        </div>
      </div>
    )}
    </main>
  </div>
 );
}

export default App;
```

CHAPTER 4
CONCLUSION

The integration of CI/CD pipelines using GitHub Actions represents a significant advancement in the adoption of DevOps practices within modern software development. Throughout this project, the core principles of DevOps—automation, collaboration, continuous integration, and continuous deployment—have been practically implemented to improve development efficiency, consistency, and reliability.

By automating the stages of testing, building, and deploying an application directly within a GitHub repository, this project demonstrates how powerful and accessible DevOps workflows have become, even for small teams or individual developers. GitHub Actions has proven to be an intuitive and flexible tool that enables developers to define complex workflows using simple YAML configurations, eliminating the need for external CI/CD tools in many cases.

In addition, the project has highlighted the importance of maintaining code quality, reducing manual errors, and accelerating the feedback loop. These outcomes align perfectly with the goals of DevOps, which seeks to bridge the gap between development and operations teams while ensuring rapid and stable software delivery.

Overall, this project serves not only as a technical implementation of CI/CD but also as a practical example of how DevOps culture and tooling can transform the software lifecycle. The lessons learned here form a strong foundation for more advanced DevOps practices in areas such as container orchestration, infrastructure as code, and scalable cloud deployments.

As the demand for faster and more reliable software continues to grow, CI/CD pipelines like the one built in this project will remain a critical component of modern development workflows—empowering teams to innovate quickly, respond to changes confidently, and deliver value to users more effectively.