WEB-BASED WEATHER APPLICATION DEVOPS PIPELINE

SWE40006

Chowdhury Mohammad Jarif Ferdous #103792481

Hossain Muntasir #103798748

Hridoy Ahmed #103798793

Sazed Ur Rahman #103815346

TABLE OF CONTENTS

Executive Summary

This report details our development of a web-based weather application designed to incorporate DevOps principles, enabling automated deployment, testing, and monitoring to streamline application delivery and maintenance. Our application, built on Python's Flask framework, integrates with the OpenWeather API to retrieve and display real-time weather data for various cities, providing users with a fun experience.

The foundation of this project is a DevOps pipeline built with GitHub Actions that allows for continuous integration and continuous deployment (CI/CD). This pipeline automates testing, deployment, and monitoring, guaranteeing that the application is always up to date with each new commit, reducing manual intervention while increasing reliability. We chose AWS and Azure as our deployment platforms because of their comprehensive scaling and monitoring capabilities, which enable us to efficiently control resource use.

This report details our development process, including consultation, design concepts, pipeline implementation, and deployment configuration. Each stage of the project is thoroughly examined, including explanations of critical decisions, code functionality, and tools used to accomplish automation and scalability. In addition to meeting our objectives, this project provided us with insights into streamlining DevOps procedures to develop a dependable, responsive, and user-friendly application.

Consultation Phase

Our first discussions had an important role in determining the project's path. Our team met to describe project goals, assign tasks, and determine the tools required for a successful DevOps pipeline. We evaluated several version control, testing, and deployment methods, as well as best practices for setting up a DevOps-driven infrastructure that reduces human burden and improves consistency.

After prolonged discussions and study, we determined on a structure and toolkit that would best meet the objectives. GitHub was chosen for version control because it is familiar and easy to use in team communication, and GitHub Actions was chosen to automate CI/CD operations. AWS and Azure were chosen for deployment due to their substantial scaling, monitoring, and DevOps integration capabilities. We also considered potential problems, such as network issues, resource restrictions, and error handling requirements. By discussing potential roadblocks, we developed proactive tactics to overcome them, producing a systematic plan for successful project execution.

In-Scope

- Weather App: Create a web-based weather app that uses the OpenWeather API to access real-time data.
- Automated CI/CD Pipeline: Automating testing and deployment.
- Error Handling: Implementing error-handling techniques for API and connection difficulties.
- Monitoring and Scaling: Configuring monitoring and autoscaling to ensure application reliability.

These in-scope areas were chosen to suit the project's basic criteria, which include functionality, automation, and scalability.

Out of Scope

To stay focused, we omitted some components from the project scope:

- User Authentication: Login feature is not required for weather data retrieval.
- Advanced Weather Features: Leaving out features such as historical data and severe weather notifications.
- Enhanced UI/UX: Restricting design alterations to a functional, user-friendly interface.
- Advanced Security Compliance: Concentrating on fundamental security procedures rather than broad compliance methods.

These initial steps ensured that each team member understood their position and the broader goals of the project. The consultation phase established the foundation for our following design and development activities, encouraging a collaborative and goal-oriented approach that would lead us throughout the project.

Project Description

The primary goal of our project was to create a weather application that provides real-time weather data via an interactive online interface. Based on user input, the program uses the OpenWeather API to retrieve important meteorological information such as temperature, wind speed, humidity, and dawn and sunset times.

Our DevOps-focused strategy sought to automate all stages of the software development lifecycle—build, testing, and deployment to ensure a seamless and quick update process. By automating these processes, we limit the possibility of human error, accelerate the integration of code changes, and keep the application's scalability and accessibility. With this configuration, we successfully aligned the application with DevOps concepts, ensuring continuous delivery and monitoring. The automated method also enabled us to create an application that is well-maintained and easily adaptable to future changes.

The automation solution has also made the program easier to maintain, as changes are sent on a constant basis without the need for user involvement. This strategy also provides for adaptability, making it easy to add future upgrades or respond to customer input. Through this configuration, we produced an application optimized for durability, performance, and adaptability, ensuring stability while adhering to modern DevOps norms for continuous development and deployment.

Objectives

- Create a user-friendly web application for getting and showing real-time weather data using the OpenWeather API.
- Create an automated CI/CD pipeline to streamline the build, testing, and deployment processes.
- Use cloud-based monitoring and scaling to ensure excellent performance and reliability.
- Implement comprehensive error handling and logging to gracefully handle API request difficulties and network outages.

- Create a responsive design that can be adjusted to different screen sizes, ensuring usability on both desktop and mobile devices.
- Document the CI/CD pipeline configuration, application structure, and user manual for future developers to ensure easy maintenance and scalability.

Problem Statement

Traditional deployment methods frequently rely on manual operations, which can lead to errors, delays, and unpredictable application performance. We attempted to address these difficulties using a DevOps strategy, developing a solid, automated pipeline that reduces manual input, maintains consistency, and is scalable. This pipeline facilitates a simplified updating cycle and provides a stable, responsive user experience. We achieved a seamless and reliable deployment process by automating the entire software lifecycle, hence improving the application's stability and usability.

Preliminary Design Concept(s)

Our design approach was guided by the requirements for automation, uniformity, and user-centered functionality. The first step of the project required choosing frameworks and technologies that were aligned with DevOps and CI/CD principles, such as Flask for backend development and HTML/CSS for frontend display. These tools enabled us to construct a lightweight yet robust web application interface for displaying real-time weather data.

Key design options included:

- Frameworks used include Python Flask for backend development and HTML/CSS for frontend display, resulting in a responsive and visually appealing interface.
- Pipeline Automation for GitHub Actions to manage each stage from development and testing to deployment, allowing us to efficiently maintain a CI/CD pipeline.
- Cloud Infrastructure: For deployment, use AWS Elastic Beanstalk or Azure App Service, both of which include monitoring and autoscaling capabilities to support high performance under variable usage demands.

The application architecture enables user input (city name), which is then used to obtain real-time weather data from the OpenWeather API. The DevOps pipeline handles all code updates, with automated testing and deployment ensuring scalability and continuous improvement. This design idea not only improves the user experience, but it also optimizes the system's performance through effective resource management.

Initial Design

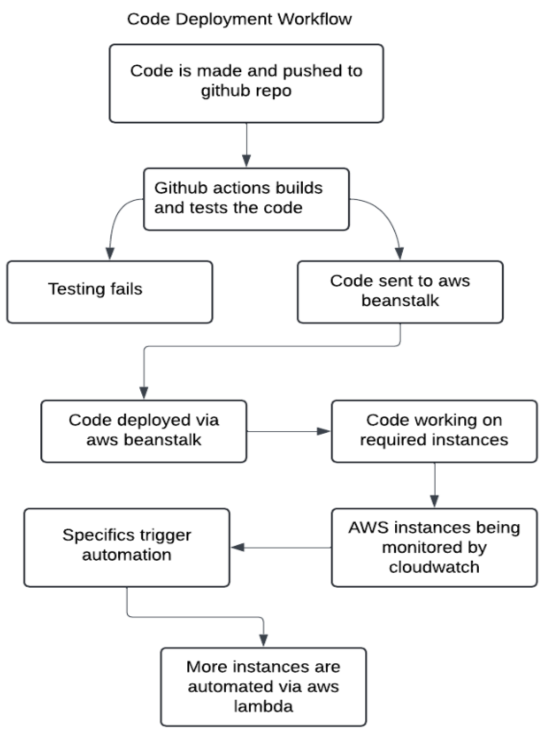

Code Deployment Workflow
Code is made and pushed to github repo
Github actions builds and tests the code
Testing fails
Code sent to aws beanstalk
Code deployed via aws beanstalk
Code working on required instances
AWS instances being monitored by cloudwatch
Specifics trigger automation
More instances are automated via aws lambda

Fig-1 Initial Design of the CI/CD pipeline

This workflow demonstrates the automated CI/CD and scaling procedure.

- Code gets pushed to the GitHub repository, starting the CI/CD pipeline.
- Build and Test: GitHub Actions automatically generates and tests the code.
  - If the tests fail, the process terminates.
  - If the tests pass, the code is deployed to AWS Elastic Beanstalk.
- Deployment and Monitoring: The deployed application runs on AWS instances that are tracked by CloudWatch for performance metrics.
- Scaling: If demand increases, CloudWatch instructs AWS Lambda to automatically add instances as needed.

This configuration allows a smooth deployment, excellent monitoring, and automatic scalability in response to demand.

Design and Development

Our development approach was divided into stages, which included implementing the weather application, setting up the CI/CD pipeline, and configuring testing and monitoring systems.

- Weather Application Development: Using the Flask framework, we built a backend environment capable of processing HTTP requests and returning reliable weather data via the OpenWeather API. Flask's simplicity and scalability make it the best choice for this application. Temperature, wind speed, humidity, and dawn and sunset times are among the key statistics offered, giving visitors a thorough insight of the current weather conditions in a certain city.

- CI/CD Pipeline Setup: The automated CI/CD pipeline was configured in GitHub Actions to run on each code commit. The pipeline reduces human interaction while ensuring consistent code quality by running automated tests and deploying only after they pass. This automated procedure enables speedy, dependable deployments while preserving the application's integrity.
- Testing and monitoring: We used automated unit tests in the pipeline to ensure core functionality, such as API connectivity and data accuracy. AWS CloudWatch and Azure's monitoring tools track metrics such as response time, CPU utilization, and memory consumption, providing information about the application's performance. This arrangement enables real-time monitoring, allowing us to immediately identify and rectify any issues that may arise, ensuring consistent application stability.

Pipeline

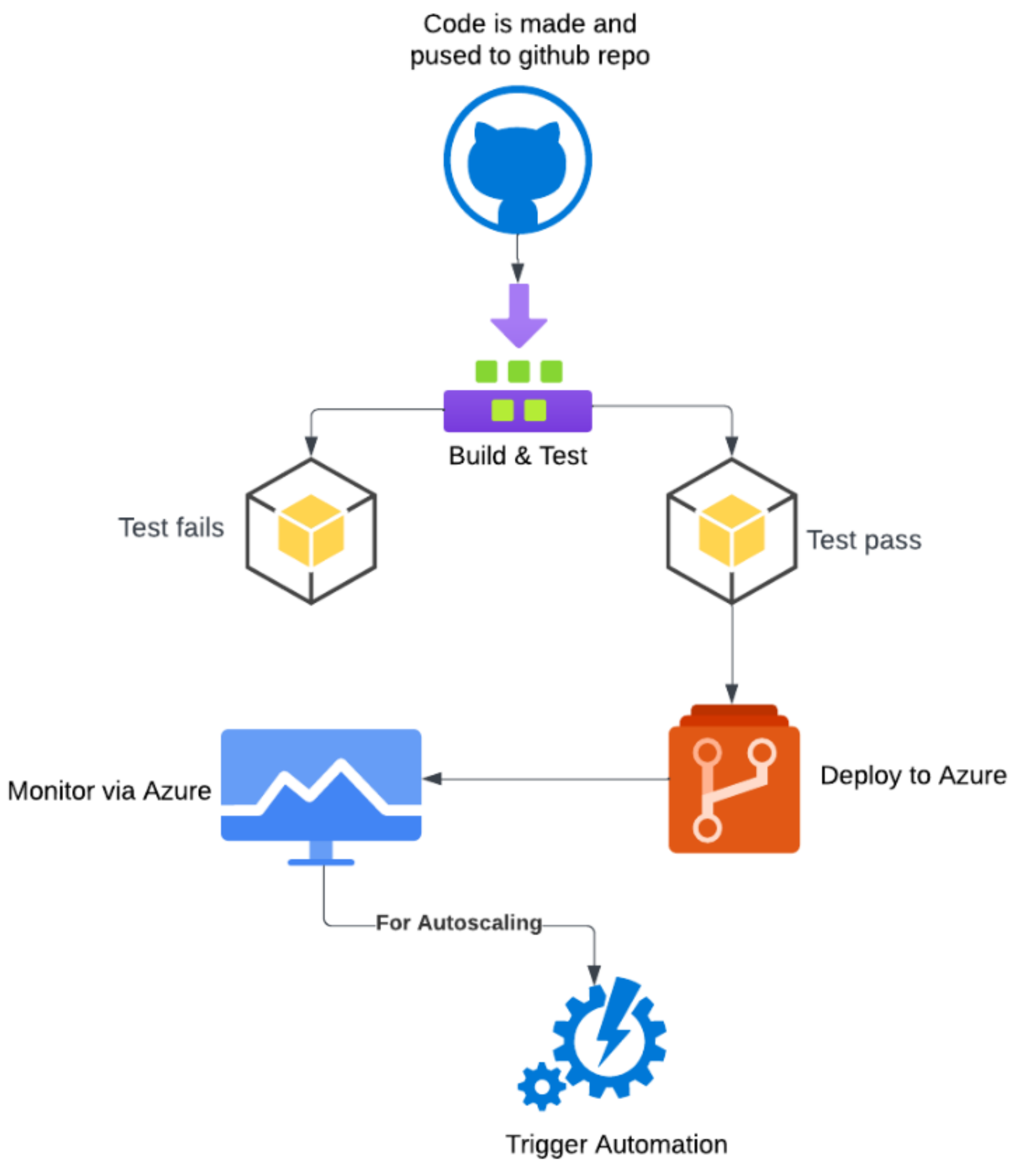


Fig-2 Final Pipeline

Differences from the first design.

The new deployment procedure differs from the original design in that it uses Azure for deployment, monitoring, and autoscaling services rather than AWS.

Initially, we planned to deploy using AWS Elastic Beanstalk, monitor with AWS CloudWatch, and autoscale with AWS Lambda. However, in the current design, Azure has been chosen as the unified platform, providing a more streamlined approach with Azure Monitor for performance monitoring and Azure's built-in autoscaling capabilities.

This migration to Azure consolidates all critical functions into a single platform, improving component integration and simplifying overall management.

Compatibility of Design

The design was created to meet all compatibility specifications. Our CI/CD pipeline related to the cloud deployment platforms Azure, which support automated scaling and monitoring. During testing, we identified small difficulties, such as receiving erroneous API replies, which we resolved by improving our error management algorithms.

Overall, the architecture adheres to DevOps standards, using CI/CD for continuous integration, continuous deployment, and powerful cloud deployment capabilities. The integration of automated testing and monitoring enables application scalability, allowing us to reach our automation and reliability objectives. The design's compliance with cloud infrastructure enables us to provide a smooth deployment, effective monitoring, and responsive scaling in a variety of load scenarios.

Justification

To establish a simplified, automated pipeline, we chose technologies and platforms that were compatible, efficient, and in accordance with DevOps best practices. Here's the reasoning behind each selection:

Why Choose GitHub over Alternatives

We chose GitHub over GitLab and Bitbucket because to its extensive adoption, ease of collaboration, and integration features. GitHub's user-friendly interface enables efficient code management, branching, and pull requests. Its interaction with GitHub Actions enables us to centralize version control and CI/CD, which reduces setup complexity. Our team's familiarity with GitHub also resulted in a more efficient workflow and reduced learning time.

Why use GitHub actions?

GitHub Actions was chosen for CI/CD because of its easy connection with GitHub repositories, enabling automated workflows triggered by code updates. Compared to standalone solutions such as Jenkins, GitHub Actions simplifies setup by functioning within GitHub, increasing efficiency and minimizing maintenance. Its flexibility allows us to create custom workflows for developing, testing, and deployment, making it excellent for our DevOps requirements.

AWS vs Azure

- AWS: AWS Elastic Beanstalk has strong integration with AWS services like as Lambda and CloudWatch for autoscaling and monitoring. Its worldwide reach and scalable architecture make it an excellent fit for DevOps operations.

- Azure: We ultimately decided on Azure for deployment because of its compatibility with Microsoft tools and unified experience. Azure App Service makes deployment easier, while Azure Monitor enables easy monitoring and autoscaling within the platform, making it suited for dynamic scaling requirements.

Our options guarantee a dependable, scalable pipeline: GitHub and GitHub Actions enable centralized version control and CI/CD, while Azure streamlines deployment and monitoring, thereby supporting our aims of automation, dependability, and efficient administration.

Specifications

Our application includes the following technological specs and components:

- Application Framework: The backend was built with Flask, while the frontend was built with HTML and CSS, resulting in a responsive and user-friendly interface.
- Libraries:
  - Flask: A core web framework that handles HTTP requests and routing.
  - urllib: Used to conduct API queries and encode city names that contain special characters.
  - json: Parses JSON answers from the API, making data extraction and display easier.
- API Integration: The OpenWeather API is used to access real-time weather data, giving consumers up-to-date information.
- GitHub's CI/CD Pipeline Actions automates the building, testing, and deployment of applications to ensure consistent performance.
- Deployment Infrastructure: The application is hosted on Azure App Service, which provide autoscaling and monitoring features to accommodate changing demand.

Issues faced

We encountered several issues during the deployment process, mainly with AWS. These challenges were mostly caused by configuration constraints, role management limits, and profile restrictions that come with free trial accounts. Below is an outline of the specific challenges we encountered:

AWS Issues

Deploying on AWS caused major hurdles, which hampered our workflow and slowed progress. AWS's extensive setup settings necessitated meticulous administration, which was exacerbated by limitations on free trial accounts.

- Role Management: Configuring AWS roles was complicated and frequently resulted in permission difficulties. Ensuring that each service had the appropriate rights

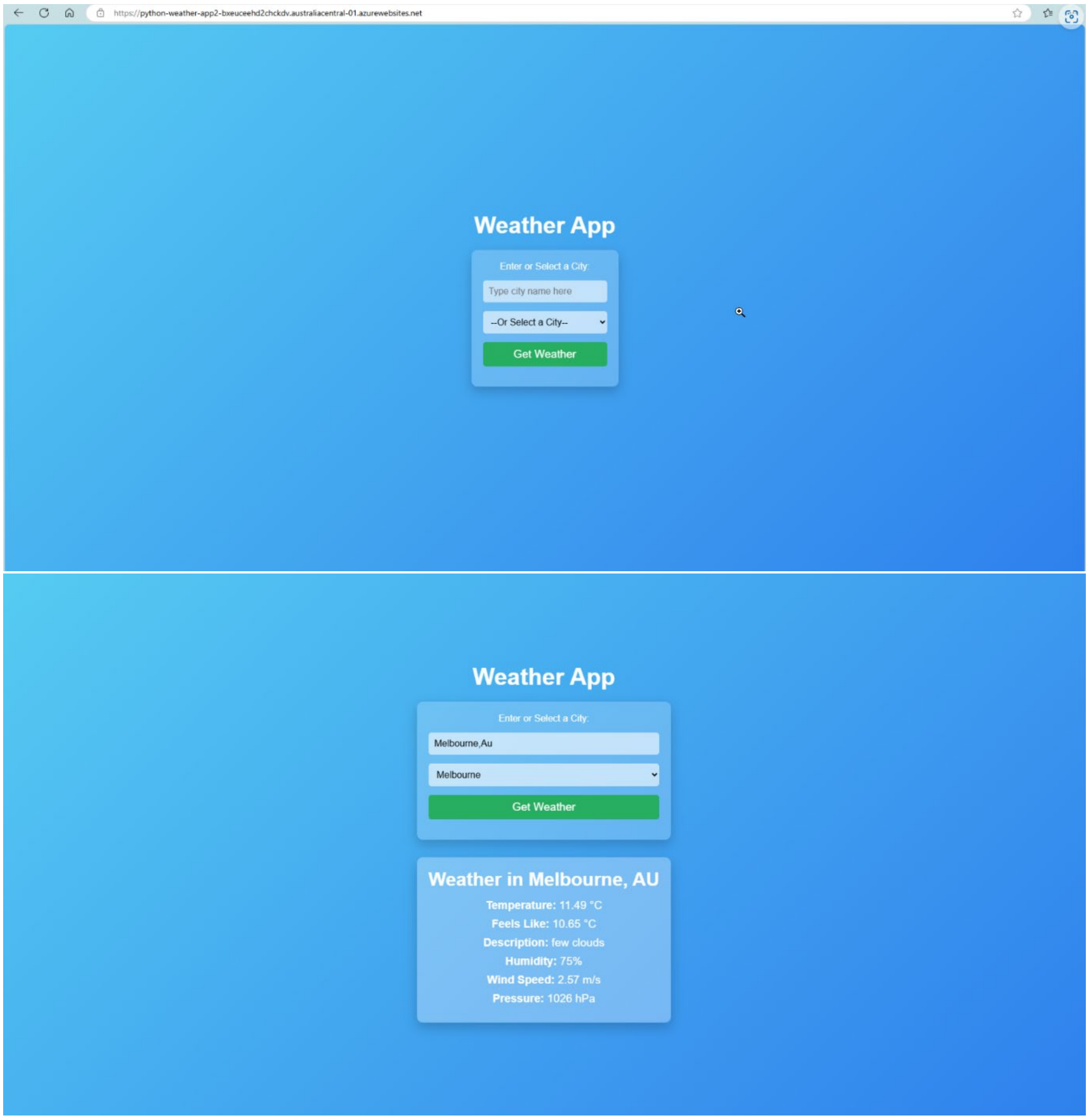
necessitated lengthy debugging, particularly given AWS's complex role management structure.
- Profile Restrictions with Free Trials: The free tier restricted us access to some capabilities, limiting our ability to test the application in production-like situations and execute advanced setups.

Challenges with AWS Deployment

- Elastic Beanstalk Configuration issues: While setting up Elastic Beanstalk, we encountered numerous configuration issues, including improper service roles and missing instance profiles. These difficulties impeded seamless integration and necessitated more time to overcome.
- Launch setup flaws caused the auto-scaling group creation to fail many times. Resolving these difficulties necessitated extensive debugging of AWS settings, which slowed rollout.
- Authentication Issues: Basic authentication posed difficulties, particularly when setting up the publish profile. These authentication concerns limited access to critical resources, complicating the deployment process.

These difficulties underlined the complexity of maintaining AWS configurations and underscored the importance of a complete understanding of AWS's role-based permissions. Despite these challenges, the experience increased our expertise with AWS' architecture, prompting us to examine Azure as an alternate deployment platform.

Data Display



Weather in Melbourne, AU

Temperature: 11.49 °C
Feels Like: 10.65 °C
Description: few clouds
Humidity: 75%
Wind Speed: 2.57 m/s
Pressure: 1026 hPa

Fig-3,4 Display of the Weather Website

The application's UI allows users to enter a city name to receive detailed weather data, as shown below:

- Temperature and Feels Like: Displayed in Celsius, providing an accurate reading of the current conditions.
- Weather Description: A brief description (e.g., "clear sky" or "light rain") gives context to present weather conditions.
- Humidity and Wind Speed: Displayed as a percentage and meters per second, allowing for a more complete understanding of the environment.
- Sunrise and Sunset: Sunrise and sunset are presented in a human-readable manner, modified from UNIX timestamps to make them easier to interpret.

This straightforward UI design presents data in an orderly, easy-to-read format, allowing users to easily evaluate weather conditions. Screenshots of the interface and data display are included in the appendix.

Monitoring

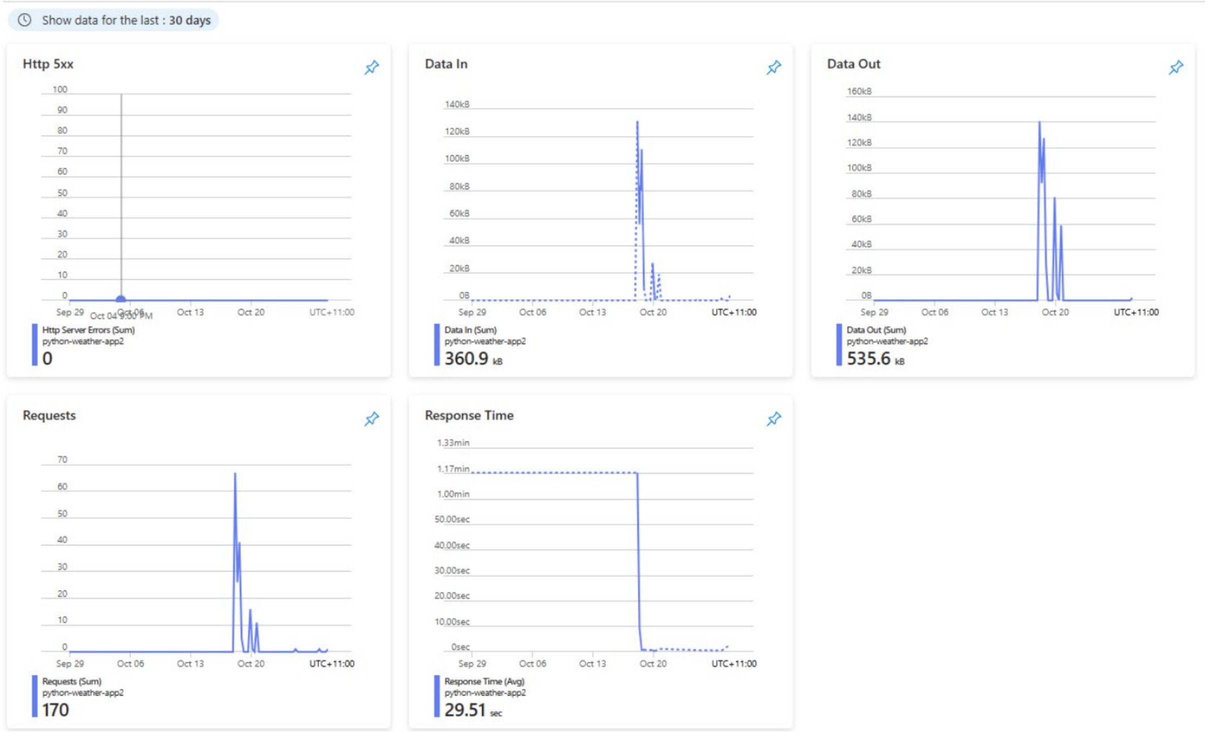


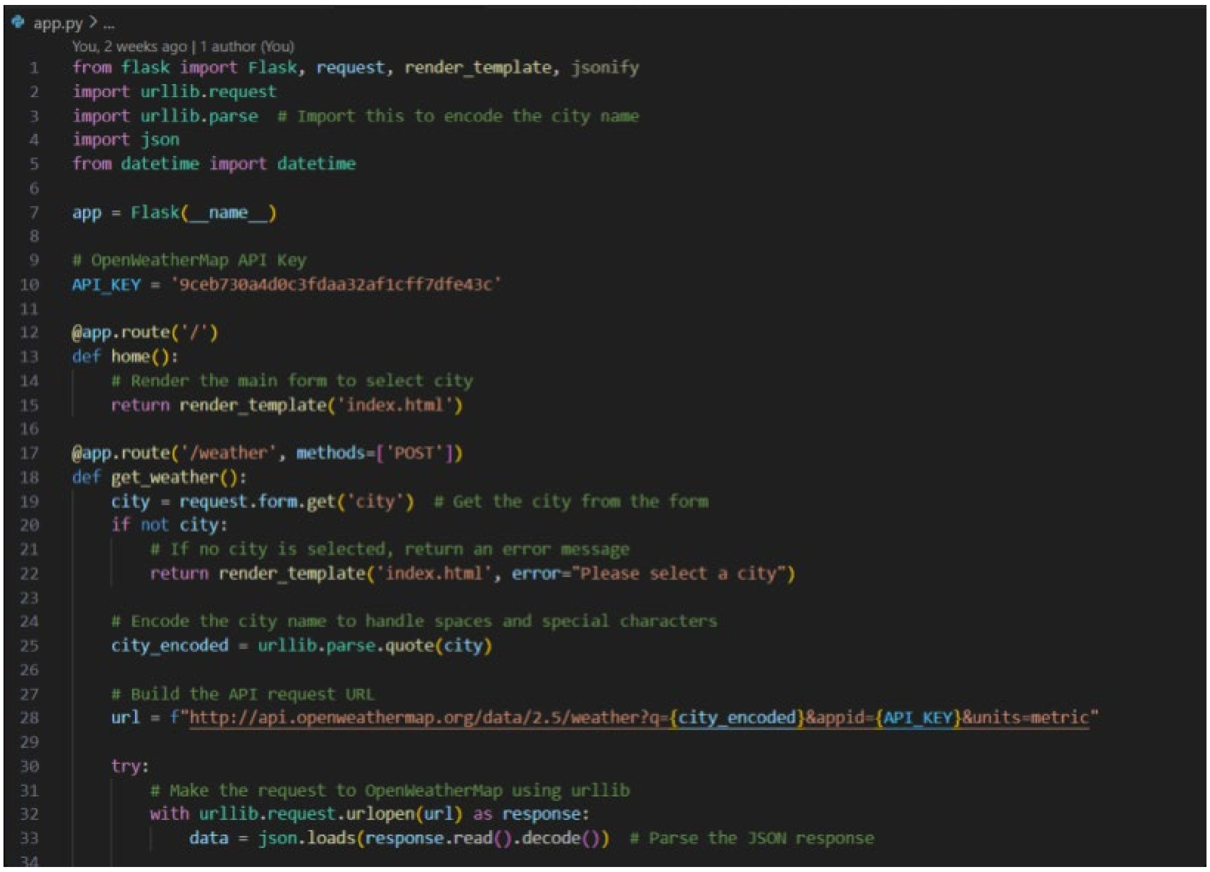
Fig-5 Monitoring Snapshot

The monitoring dashboard displays critical performance information for the weather application from the previous 30 days. Each panel shows data to help you assess the application's health and responsiveness.

- HTTP 5xx Errors: This panel monitors server errors; a count of 0 indicates no significant server failures or critical mistakes in the last month.
- Data In: Displays all data received by the application. A surge is seen around mid-October, reaching roughly 360.9 KB, indicating a period of increasing use or testing.
- Data Out: Displays the application's outgoing data, which totals 535.6 KB. Peaks in data output correspond to spikes in data input, which represent user requests being processed and responses being provided.
- Response Time: Shows the application's average response time, with a maximum response time of 1.33 minutes. The response time soon stabilizes after first peaks, demonstrating that the application adapts to demand effectively.
- These metrics help measure the application's performance, efficiency, and scalability by providing information about data utilization, request handling, and server response behavior.

Code Explanation

App.py



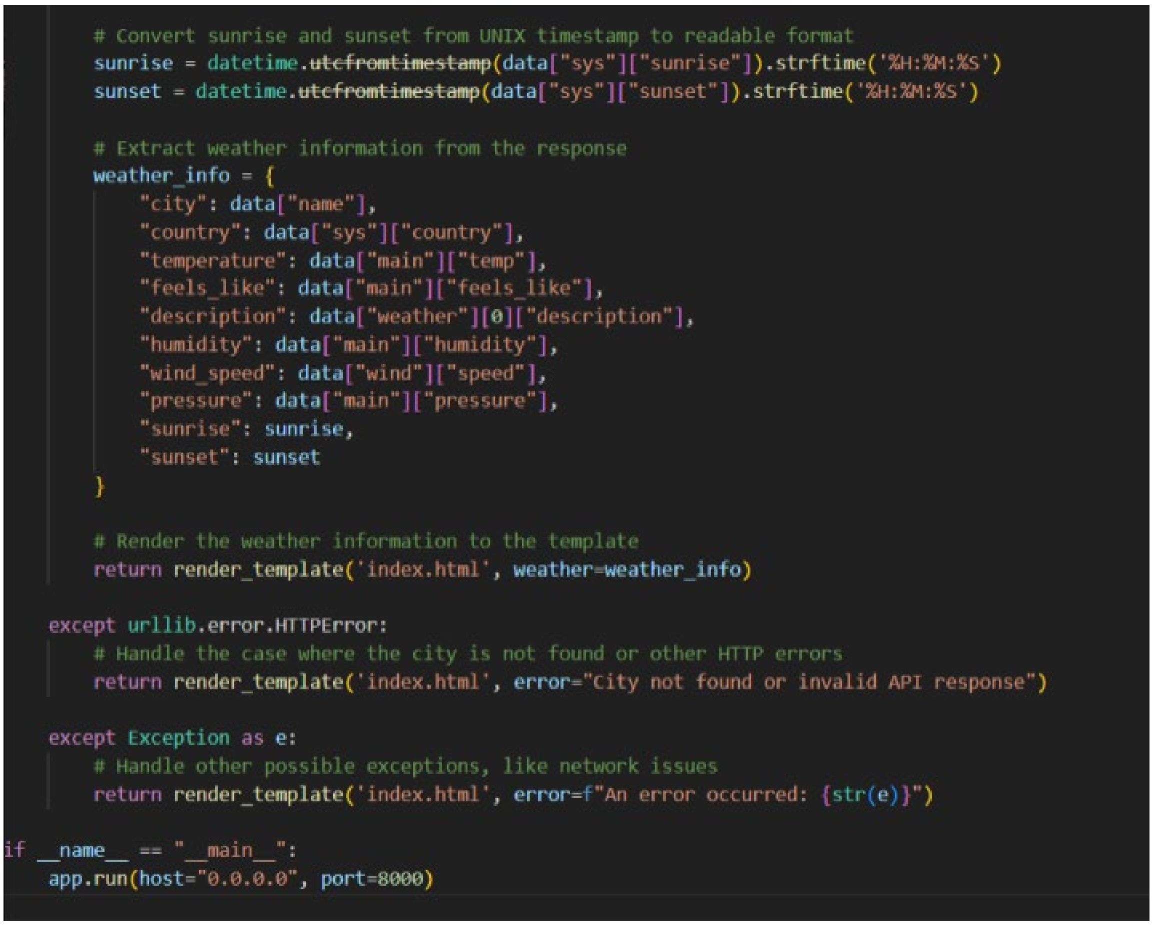
```python
from flask import Flask, request, render_template, jsonify
import urllib.request
import urllib.parse  # Import this to encode the city name
import json
from datetime import datetime

app = Flask(__name__)

# OpenWeatherMap API Key
API_KEY = '9ceb730a4d0c3fdaa32af1cff7dfe43c'

@app.route('/')
def home():
    # Render the main form to select city
    return render_template('index.html')

@app.route('/weather', methods=['POST'])
def get_weather():
    city = request.form.get('city')  # Get the city from the form
    if not city:
        # If no city is selected, return an error message
        return render_template('index.html', error="Please select a city")

    # Encode the city name to handle spaces and special characters
    city_encoded = urllib.parse.quote(city)

    # Build the API request URL
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city_encoded}&appid={API_KEY}&units=metric"

    try:
        # Make the request to OpenWeatherMap using urllib
        with urllib.request.urlopen(url) as response:
            data = json.loads(response.read().decode())  # Parse the JSON response
```

Fig – 6 Snapshot of Code (1)

This Python code snippet serves as the weather app's backend, and it uses Flask to retrieve real-time weather data from the OpenWeather API.

- Imports: The following libraries are required: Flask for routing, urllib for API queries, json for response parsing, and datetime for managing time data.
- API Key: The OpenWeather API key is configured to authenticate queries.
- Routes:
  - @app.route('/'): Displays index.html, allowing users to provide a city name.
  - @app.route('/weather', methods=['POST']): Accepts POST requests to retrieve weather data based on user input.
- Data retrieval and API requests:
  - Retrieves and encodes the city's name.
  - Creates the API request URL, submits it, then parses the JSON answer to display weather information.

This code handles the app's fundamental operations, including input, API calls, and data parsing.



```python
        # Convert sunrise and sunset from UNIX timestamp to readable format
        sunrise = datetime.~~utcfromtimestamp~~(data["sys"]["sunrise"]).strftime('%H:%M:%S')
        sunset = datetime.~~utcfromtimestamp~~(data["sys"]["sunset"]).strftime('%H:%M:%S')

        # Extract weather information from the response
        weather_info = {
            "city": data["name"],
            "country": data["sys"]["country"],
            "temperature": data["main"]["temp"],
            "feels_like": data["main"]["feels_like"],
            "description": data["weather"][0]["description"],
            "humidity": data["main"]["humidity"],
            "wind_speed": data["wind"]["speed"],
            "pressure": data["main"]["pressure"],
            "sunrise": sunrise,
            "sunset": sunset
        }

        # Render the weather information to the template
        return render_template('index.html', weather=weather_info)

    except urllib.error.HTTPError:
        # Handle the case where the city is not found or other HTTP errors
        return render_template('index.html', error="City not found or invalid API response")

    except Exception as e:
        # Handle other possible exceptions, like network issues
        return render_template('index.html', error=f"An error occurred: {str(e)}")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

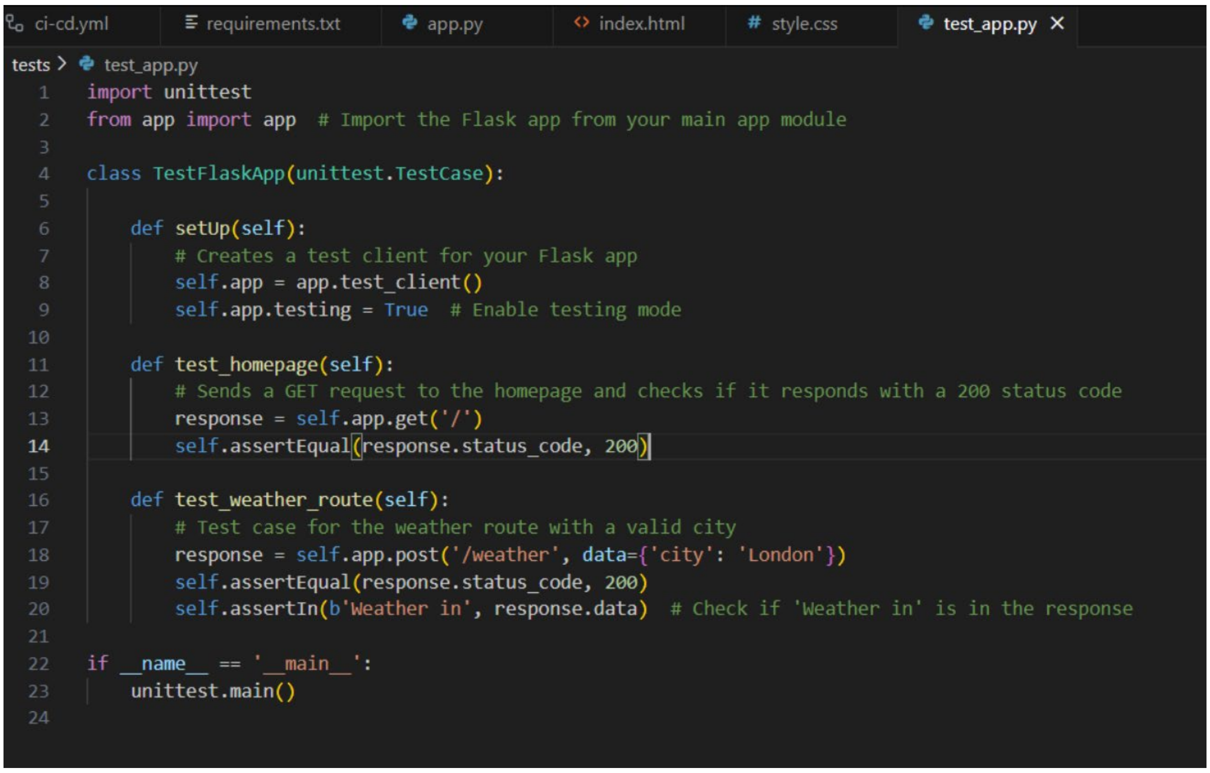
Fig – 7 Snapshot of Code (2)

This portion of code handles the API response and prepares weather data for presentation on the front end.

- Sunrise and Sunset Conversion: The datetime.utcfromtimestamp() function converts UNIX timestamps for sunrise and sunset into a readable format.
- Weather Data Extraction: extracts crucial information from the API response, such as:
  - city, country, temperature, feel_like, description, humidity, wind_speed, and pressure.
  - Adds formatted sunrise and sunset times to the weather_info dictionary.
- Template Rendering: Passes the weather_info dictionary to index.html to display the data.
- Error Handling:
  - HTTPError: Renders an error message if the city is not found.
  - General Exception: Catches other issues, displaying a generic error message.
- Application Run: Runs the Flask app on host 0.0.0.0 and port 8000, making it accessible on the local network.

This code handles data processing, error management, and data presentation for the weather app.

Testing



ci-cd.yml · requirements.txt · app.py · index.html · style.css · test_app.py

tests > test_app.py

```python
import unittest
from app import app  # Import the Flask app from your main app module

class TestFlaskApp(unittest.TestCase):

    def setUp(self):
        # Creates a test client for your Flask app
        self.app = app.test_client()
        self.app.testing = True  # Enable testing mode

    def test_homepage(self):
        # Sends a GET request to the homepage and checks if it responds with a 200 status code
        response = self.app.get('/')
        self.assertEqual(response.status_code, 200)

    def test_weather_route(self):
        # Test case for the weather route with a valid city
        response = self.app.post('/weather', data={'city': 'London'})
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'Weather in', response.data)  # Check if 'Weather in' is in the response

if __name__ == '__main__':
    unittest.main()
```

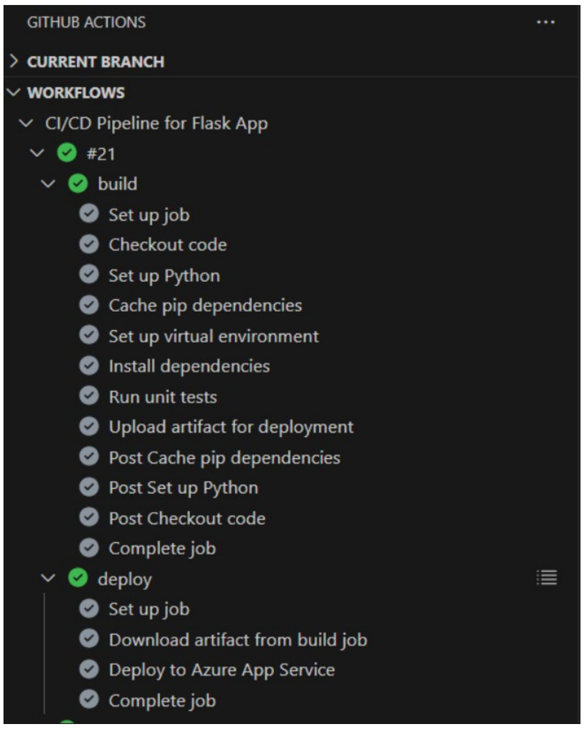
Fig – 8 Snapshot of Unittests

This code snippet is a test file for the weather application, which uses Python's unittest framework to evaluate its functioning.

- Imports: Unittest and the Flask app module are used to test the application routes.
- Test Class Setup: This section defines TestFlaskApp as a test class that derives from unittest.TestCase.
    - setUp(self): Creates a test client for the Flask app and activates testing mode, which allows isolated test requests.
- Homepage Test:
    - test_homepage(self): Sends a GET request to the homepage ('/') and checks if the response status code is 200, indicating that the page has loaded correctly.
- Weather Route Test:
    - test_weather_route(self): Makes a POST request to the /weather route using a sample city ('London') as the data. It checks for a response status code of 200, indicating that the route is available.
    - If the word 'Weather' appears in the response data, it indicates that the route provides weather-related information.
- Test Execution: If the script is ran directly, the tests are run using unittest.main.

This testing code guarantees that the homepage and weather route function properly, by evaluating successful response codes and suitable content in responses.

Workflow


GITHUB ACTIONS

CURRENT BRANCH

WORKFLOWS

- CI/CD Pipeline for Flask App
  - #21
    - build
      - Set up job
      - Checkout code
      - Set up Python
      - Cache pip dependencies
      - Set up virtual environment
      - Install dependencies
      - Run unit tests
      - Upload artifact for deployment
      - Post Cache pip dependencies
      - Post Set up Python
      - Post Checkout code
      - Complete job
    - deploy
      - Set up job
      - Download artifact from build job
      - Deploy to Azure App Service
      - Complete job

Fig- 9 Snapshot of Codeflow in GithubActions

This GitHub Actions process describes the Flask application's CI/CD pipeline, covering the build and deployment stages.

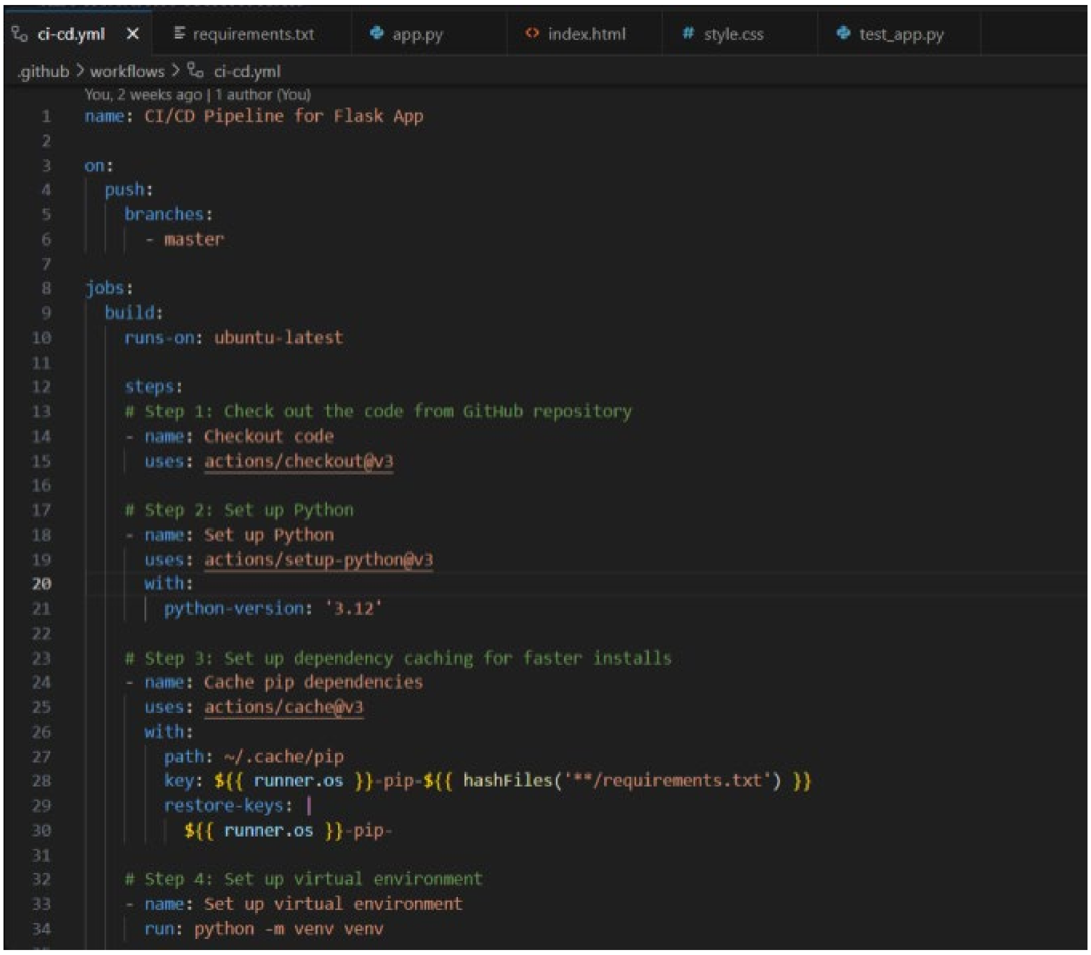
Build Stage:

- Setup job: Initializes the build environment.
- Checkout Code: Retrieves code from the GitHub repository.
- Setup Python: Configures the Python environment.
- Cache pip dependencies: Stores dependencies to speed up future builds.
- Set up a virtual environment. Installing dependencies: Creates a virtual environment and installs the required components.
- Run the unit tests: Executes unit tests to validate that the code works as intended.
- Upload an artifact for deployment. Prepares and submits build artifacts to the deployment stage.
- Post-job steps: Clears the cached resources and completes the build.

Deployment Stage:

- Setup job: Initializes the deployment environment.
- Download artifact from build job. Retrieves the artifact generated during the build stage.
- Deploy to Azure App Services: The application is then deployed to Azure App Service for hosting.
- Complete the job: Finalizes the deployment procedure.
- This CI/CD pipeline automates code testing, building, and deployment, reducing the development cycle and ensuring dependability.

YML


ci-cd.yml
requirements.txt
app.py
index.html
style.css
test_app.py
.github > workflows > ci-cd.yml
You, 2 weeks ago | 1 author (You)

```yaml
name: CI/CD Pipeline for Flask App

on:
  push:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    # Step 1: Check out the code from GitHub repository
    - name: Checkout code
      uses: actions/checkout@v3

    # Step 2: Set up Python
    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.12'

    # Step 3: Set up dependency caching for faster installs
    - name: Cache pip dependencies
      uses: actions/cache@v3
      with:
        path: ~/.cache/pip
        key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}
        restore-keys: |
          ${{ runner.os }}-pip-

    # Step 4: Set up virtual environment
    - name: Set up virtual environment
      run: python -m venv venv
```

Fig – 10 Snapshot of YML (1)

This YAML file sets up the GitHub Actions CI/CD pipeline for the Flask application.

- Trigger: Executes on pushes to the master branch to automate testing and deployment.

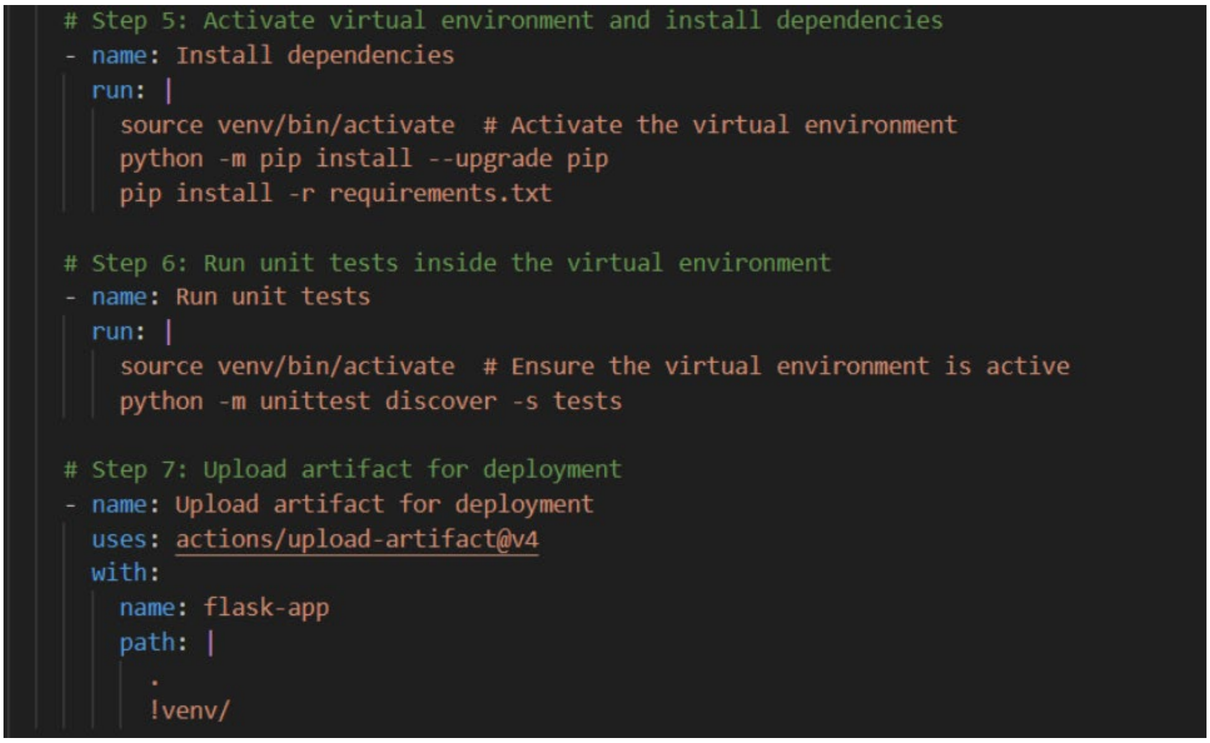
Build Job:

- The environment is based on Ubuntu-latest.

Steps:

- Checkout Code is pulled from the repository using actions/checkout@v3.
- Setup Python: The actions/setup-python@v3 script installs Python 3.12.
- Cache Dependencies: Caches pip dependencies to increase build speed.
- Set up Virtual Environment: Generates a virtual environment.

This configuration automates code retrieval, environment setup, and dependency caching, resulting in consistent and efficient builds.



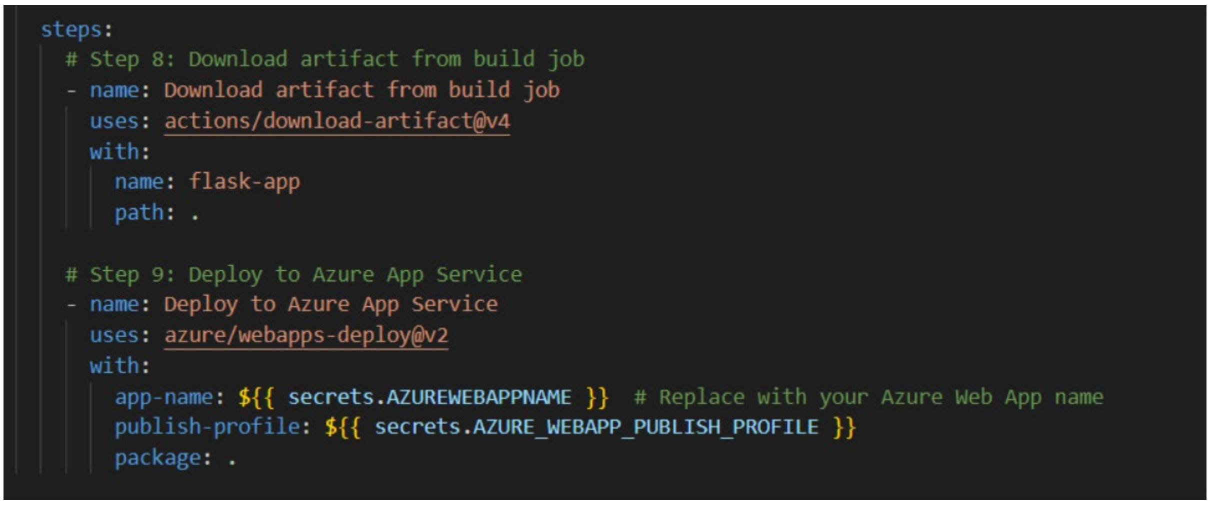
```yaml
# Step 5: Activate virtual environment and install dependencies
- name: Install dependencies
  run: |
    source venv/bin/activate  # Activate the virtual environment
    python -m pip install --upgrade pip
    pip install -r requirements.txt

# Step 6: Run unit tests inside the virtual environment
- name: Run unit tests
  run: |
    source venv/bin/activate  # Ensure the virtual environment is active
    python -m unittest discover -s tests

# Step 7: Upload artifact for deployment
- name: Upload artifact for deployment
  uses: actions/upload-artifact@v4
  with:
    name: flask-app
    path: |
      .
      !venv/
```

Fig – 11 Snapshot of YML (2)

This section of the YAML file describes the stages in the GitHub Actions pipeline for installing dependencies, performing tests, and preparing artifacts for deployment.

- Install dependencies:
  - Activates the virtual environment.
  - Installs the dependencies indicated in requirements.txt after upgrading pip.
- Run unit tests.
  - Activates the virtual environment.
  - Use unittest to run unit tests in the tests directory.
- Upload an artifact for deployment.
  - Actions/upload-artifact@v4 is used to prepare and upload build artifacts for later deployment, ignoring the virtual environment directory (!venv/).

These processes ensure that the application's dependencies are installed, tested, and packaged for deployment, hence facilitating a smooth CI/CD process.



```yaml
steps:
  # Step 8: Download artifact from build job
  - name: Download artifact from build job
    uses: actions/download-artifact@v4
    with:
      name: flask-app
      path: .

  # Step 9: Deploy to Azure App Service
  - name: Deploy to Azure App Service
    uses: azure/webapps-deploy@v2
    with:
      app-name: ${{ secrets.AZUREWEBAPPNAME }}  # Replace with your Azure Web App name
      publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
      package: .
```

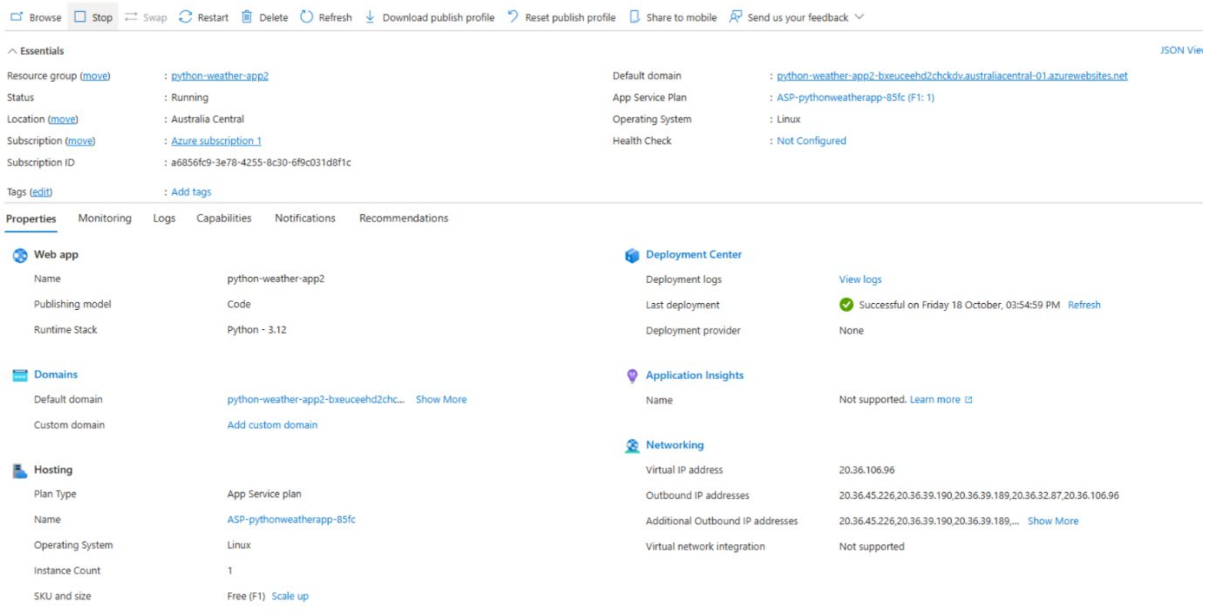
Fig – 12 Snapshot of YML (3)

This section of the GitHub Actions pipeline's YAML file defines the Azure App Service deployment process.

- Download the artifact from the build job.
  - Uses actions/download-artifact@v4 to retrieve the artifact (flask-app) created during the build stage and prepare it for deployment.
- Deploy to Azure App Services:
  - To deploy the application to Azure, run azure/webapps-deploy@v2.
  - To gain safe access to the Azure Web App Service, specify the app-name and publish-profile from GitHub Secrets.

These steps complete the CI/CD pipeline by downloading and delivering the build artifact to Azure, resulting in a smooth and safe deployment procedure.

Deployment



Essentials

Resource group (move)	: python-weather-app2
Status	: Running
Location (move)	: Australia Central
Subscription (move)	: Azure subscription 1
Subscription ID	: a6856fc9-3e78-4255-8c30-6f9c031d8f1c
Tags (edit)	: Add tags
Default domain	: python-weather-app2-bxeuceehd2chckdv.australiacentral-01.azurewebsites.net
App Service Plan	: ASP-pythonweatherapp-85fc (F1: 1)
Operating System	: Linux
Health Check	: Not Configured

Web app

Name	python-weather-app2
Publishing model	Code
Runtime Stack	Python - 3.12

Domains

Default domain	python-weather-app2-bxeuceehd2chc... Show More
Custom domain	Add custom domain

Hosting

Plan Type	App Service plan
Name	ASP-pythonweatherapp-85fc
Operating System	Linux
Instance Count	1
SKU and size	Free (F1) Scale up

Deployment Center

Deployment logs	View logs
Last deployment	✓ Successful on Friday 18 October, 03:54:59 PM Refresh
Deployment provider	None

Application Insights

Name	Not supported. Learn more

Networking

Virtual IP address	20.36.106.96
Outbound IP addresses	20.36.45.226,20.36.39.190,20.36.39.189,20.36.32.87,20.36.106.96
Additional Outbound IP addresses	20.36.45.226,20.36.39.190,20.36.39.189,... Show More
Virtual network integration	Not supported

Fig – 13 Snapshot of Deployment in Azure

This snapshot shows the deployment status and configuration details for the Flask weather app on Azure.

- Essentials:
  - The Python-weather-app2 resource group.
  - The current status is running.
  - Location: Australia Central.
  - Subscription ID: A unique Azure identification for resource tracking.
- Web Application Properties:
  - Name: python-weather-app2.
  - The publishing model is code.
  - The runtime stack is Python 3.12.
- Domains:
  - Default Domain: The Azure-provided URL for app access.
  - Custom Domain: This option is available for custom URLs.
- Deployment Center:
  - Displays deployment logs, with the last deployment marked as successful.
- Networking:
  - Lists the app's virtual IP and outbound IP addresses for connectivity.

This summary verifies that the app was successfully deployed on Azure, with configuration details indicating that it is fully operational.

Improvements

One significant improvement highlighted for the project is the separation of environments for building, testing, and deployment. We currently employ only two environments: one for building and testing, and another for deployment. The combined build-and-test environment may cause issues, as modifications or dependencies during the build process may impair test accuracy.

- Implementing three unique contexts for each phase would improve clarity and reliability.
- Build Environment: A separate area for compiling and preparing code without affecting tests.
- Test Environment: Isolated to guarantee that tests execute independently of build modifications, resulting in more accurate feedback.
- Deployment Environment: Used just for deploying stable, production-ready code.

While we established a virtual testing environment to offset some of these risks, properly separating each phase into independent settings would increase pipeline resilience and lower the likelihood of environmental difficulties.

Project outcomes

The project met all its objectives:

- CI/CD Pipeline: GitHub Actions reliably automates deployment of code updates, ensuring that the application is consistently updated.
- Weather Application: Users can obtain real-time weather data through a dynamic interface, which improves functionality.
- Scalability and Monitoring: Azure's autoscaling and monitoring tools accurately track performance, enabling for rapid scaling.
- Error Handling: Improved error handling gives consumers with helpful feedback when errors emerge, resulting in a more seamless experience.
- Appendices include screenshots of the deployed application, CI/CD logs, and the monitoring dashboard, which highlight the project's results.

Learnings

This project offered significant insights into DevOps, cloud settings, and CI/CD processes.

- Cloud Platform Configuration: Understanding cloud rights, responsibilities, and settings is critical for a successful implementation and scalability.
- CI/CD Pipelines: Automating deployments with CI/CD increased efficiency, reduced manual errors, and enabled consistent updates.
- Environment Segmentation: Separate environments for build, test, and deployment improve clarity and testing accuracy, emphasizing the value of environment management.
- Error Handling and Monitoring: Monitoring tools enabled us to spot problems rapidly, allowing for proactive performance optimization.
- Version Control Integration: Using GitHub for both version control and CI/CD streamlined cooperation and increased workflow efficiency.

These lessons have improved our understanding of DevOps and automation, allowing us to apply best practices to future initiatives.

References

Kim, D., & Kim, J. (2020). *DevOps for Dummies*. Wiley.

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

Microsoft Azure Documentation. (2023). *Deploy a Python web app using Azure App Service*.

Burns, B., & Oppenheimer, D. (2017). *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media.

Amazon Web Services (AWS). (2023). *Best Practices for CI/CD in the Cloud*.