PROJECT REPORT SOURCE CODE MANAGEMENT

Expense Tracker

Submitted By

Name of the student:	Roll No.
Srishti Jain	102316080
Jyotika Mittal	102303722
Delphi Gupta	102315027
Ishaan Sharma	102303795
Saksham Tiwari	102303695

Submitted To: Dr.Stuti Chug

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY, PATIALA-147001, PUNJAB JAN-MAY 2025

TABLE OF CONTENTS

1. Abstract:

This project focuses on developing and deploying a cloud-based Expense Tracker application using modern DevOps and web technologies. The backend of the application is built using Python Flask, which provides lightweight routing and API handling for managing expense data. To ensure consistency, portability, and ease of deployment, the application is containerized using Docker. The Docker image is then pushed to Docker Hub, enabling seamless access from remote environments.

For hosting, an AWS EC2 instance is configured as the cloud server. Docker is installed on the instance, and the application container is executed, making the service available publicly over the internet. Proper security group settings are applied to allow inbound traffic on required ports. Git and GitHub are used throughout development for version control and project management.

The project demonstrates the complete workflow of creating a web application, containerizing it, and deploying it on a cloud platform. It highlights practical skills in cloud computing, Docker containerization, backend development, debugging, and deployment automation. The final outcome is a fully functional, publicly accessible Expense Tracker web application running on AWS.

2. Introduction

2.1 Background

With the rapid growth of digital platforms, modern applications must be efficient, user-friendly, and scalable. Software development today is not limited to writing code; it requires the integration of multiple domains such as frontend design, backend logic, database management, authentication, DevOps tools, containerization, cloud deployment, and version controlThis project integrates these real-world development practices into a single working system.It includes creating a functional web application supported by a backend server, a connected database, and secure login/signup mechanisms. The project also incorporates technologies such as Git, GitHub, Docker, CI/CD pipelines, and cloud deployment services like Vercel or GitHub Pages, reflecting modern software engineering standards.

2.2 Problem Statement

In real development workflows, teams face multiple challenges: Managing separate frontend and backend codebases Ensuring secure and reliable user authentication Maintaining consistent versions of the code across multiple developers. Deploying applications smoothly without manual errors. Running applications consistently on all systems without dependency issues. Delivering updates automatically through CI/CD pipelines. Many student-level or beginner projects focus only on frontend or backend, ignoring real-world practices like version control, containerization, and cloud deployment. There is a need for a complete full-stack solution that integrates all these aspects into one project. This project addresses these challenges by creating a fully functional full-stack web application with authentication, DevOps integration, containerization, and cloud deployment.

2.3 Objectives

The key objectives of this project are:

2.3.1 Development Objectives

- To design a responsive and user-friendly frontend interface.
- To develop a backend server capable of handling requests and data processing. To implement a database system for storing user information securely.
- To create login and signup authentication with proper validation and security.

2.3.2 Version Control & Collaboration Objectives

- To use Git and GitHub for proper version management.
- To organize code using branches, commits, and pull requests.

2.3.3 Deployment & DevOps Objectives

- To deploy the frontend using Vercel or GitHub Pages.
- To containerize the application using Docker for consistent environment setup.
- To implement automation pipelines using GitHub Actions or Jenkins.To demonstrate real-world CI/CD integration for seamless deployment.

2.3.4 Learning Objectives

- To understand full-stack development through a practical project.
- To gain hands-on experience in deploying and managing applications on cloud servers.
- To learn modern DevOps tools and workflows used in industry.

3. Technologies Used

The project incorporates a comprehensive set of modern development and deployment technologies to ensure functionality, scalability, and automation. The primary technologies utilized include:

- Flask for building the backend web application and handling server-side logic.
- Docker and Docker Compose for containerization, ensuring consistent environments across development and deployment stages.
- Git and GitHub for version control, collaboration, and maintaining the project repository.
- AWS EC2 for hosting and deploying the application on a cloud-based virtual server.
- PyTest for automated testing and validation of backend functionalities.
- Jenkins for implementing continuous integration and continuous deployment (CI/CD) pipelines.
- Monitoring Tools to track application performance, uptime, and system behavior.
- GitHub Actions for workflow automation, including testing, building, and deployment processes.

4. Flask

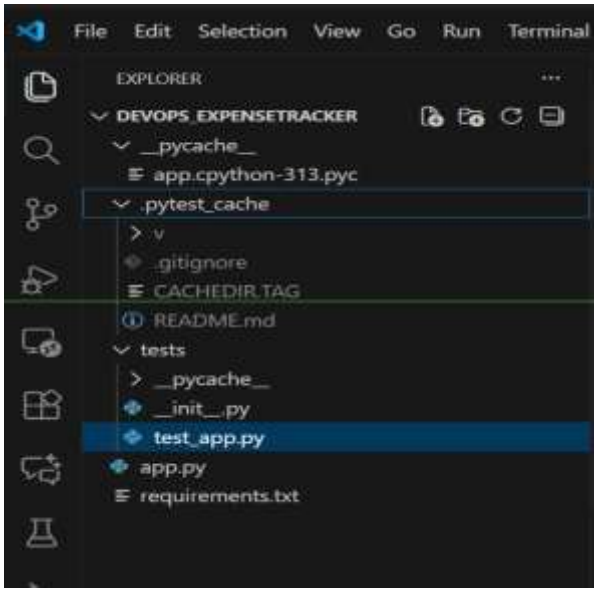
Flask is a lightweight, Python-based web framework used for building web applications and APIs.
It is called a micro-framework because:

- It does not force a strict project structure
- It is small and easy to learn
- It provides only the essentials: routing, templates, request handling
- Developers can add extensions when needed (DB, Auth, etc.)

Common Use Cases of Flask

- Building REST APIs
- Small-to-medium web applications
- Prototyping and backend services
- Machine Learning model deployment
- Dashboard/analytics tools

4.1 Folder Structure



4.2 Main Features

- Routing: Used to map URLs to specific backend functions for page rendering and data processing.
- Template Rendering: Jinja2 templates are used to generate dynamic HTML pages with backend data.
- Static File Handling: CSS, images, and JavaScript files are served through Flask's static/ directory.
- GET & POST Methods: Supports user input handling and page display through HTTP methods.
- Local Development Server: The app runs using python app.py for easy testing and debugging.
- Simple and Modular Structure: Clear separation of templates, static files, and logic, making the project easy to maintain

4.3 Technical Commands Used

1) The backend of the project was developed using Flask, and the logic is implemented inside the app.py file. The code begins with importing the necessary modules using the command:

from flask import Flask, render_template, request

This import statement provides access to Flask's core features, including routing, HTML rendering, and handling form inputs.

2) The application instance is initialized using the command:

app = Flask(__name__)

This creates the central Flask object responsible for managing all incoming requests and responses.

3) The first route defined in the project handles the homepage. It is created using the decorator:

@app.route("/")

and returns the main interface through the command:

return render_template("index.html")

This ensures that whenever a user visits the server address, the index.html file stored in the templates directory is displayed.

4) A second route is used to process user inputs. This route accepts form submissions and is defined with the command:

@app.route("/calculate", methods=["POST"])

Inside this route, the application retrieves form values using:

request.form.get("input_name")

These retrieved values are then passed into the core calculation logic of the project.

5) After processing, the final result is returned to the user using:

return render_template("result.html", output=value)

This completes the request–response cycle by sending the computed result back to the appropriate HTML page.

6) For local execution and debugging, the application is started using:

if __name__ == "__main__":

app.run(debug=True)

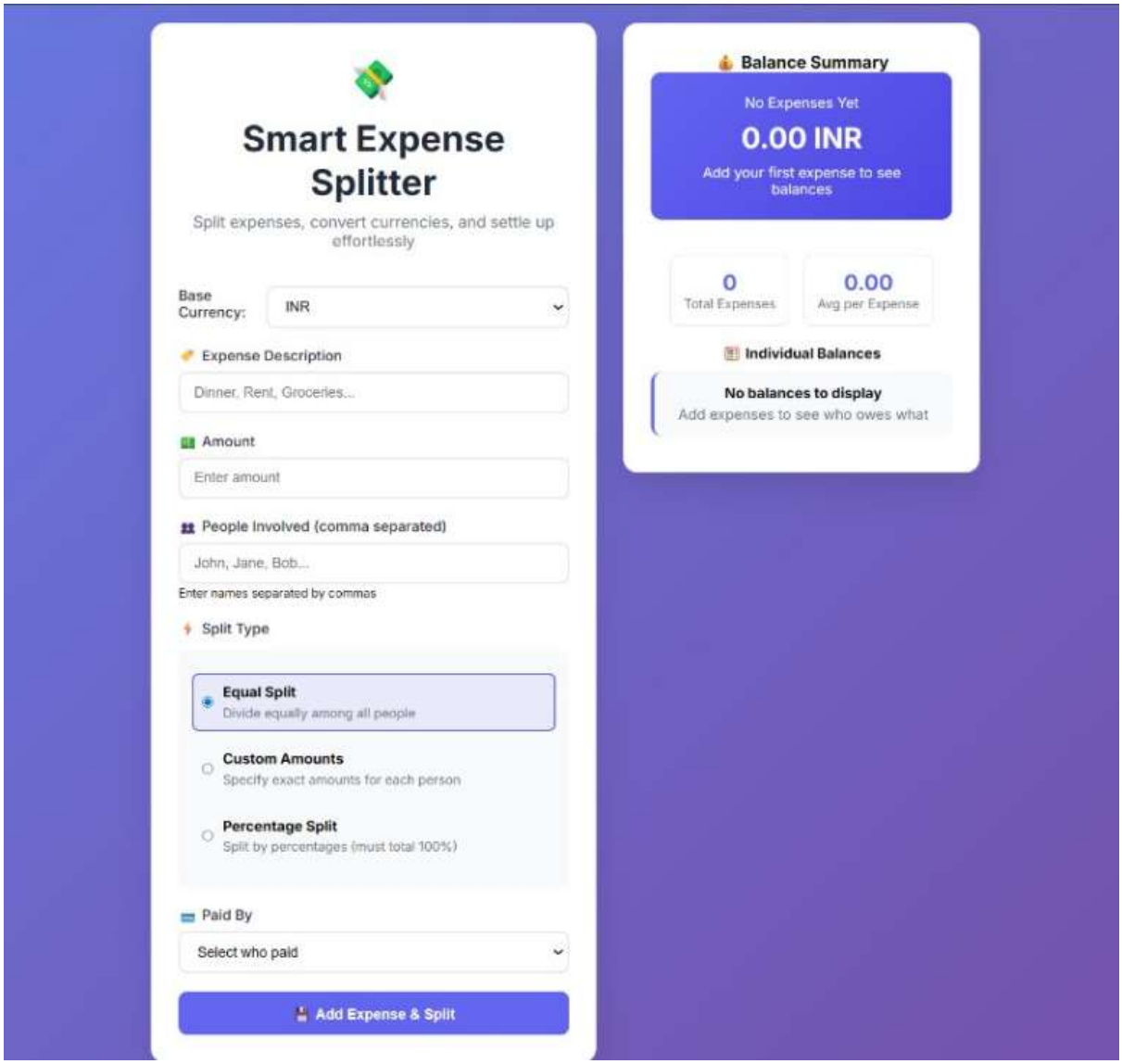
This command runs the development server in debug mode, enabling real-time error reporting and automatic reloads during modification.

7)The project also uses a requirements.txt file generated with the command:

pip freeze > requirements.txt

This file contains all the Python dependencies needed to run the application, ensuring that deployment environments install the exact versions of required libraries.

4.4 Illustration of the Tool Used



This website helps users split shared expenses easily.

A user enters: the expense description, total amount, names of people involved, how the amount should be split (equal, custom, or percentage),and who paid the bill. After submitting, the site automatically calculates who owes how much and shows a balance summary with total expenses and individual balances. It makes group expense management simple and organized.

5. Version Control: Git + GitHub

Git is a **version control system** used to track changes in code. It helps developers manage their project history, collaborate with others, and save different versions of their work.

Git is useful because:

- It keeps track of every change in your files
- You can go back to any previous version
- You can work on multiple branches safely
- You can collaborate with others without overwriting each other's work

Common Use Cases of Git

- Software development (any programming language)
- Team collaboration
- Tracking and managing code changes
- Maintaining backups
- Open-source contributions

5.1 Main Commands

1) git init

Feature: Initializes a new local Git repository in the project folder.

Usage in Project: Used at the beginning to convert the project folder (Flask application) into a version-controlled repository.

2. git add .

Feature: Stages all modified and new files for commit.

Usage in Project: Used every time files like app.py, templates, static files, and requirements.txt were updated.

3. git commit -m "message"

Feature: Saves the staged changes into the repository with a descriptive message.

Usage in Project: Used to record meaningful checkpoints while developing features (e.g., “Added HTML UI”, “Configured Flask routes”, “Updated Dockerfile”).

4. git branch -M main

Feature: Creates or renames the main branch and sets it as the primary branch.

Usage in Project: Standardized the repository to use main as the default branch before pushing to GitHub.

5. git remote add origin <repo-link>

Feature: Links the local project with a GitHub repository.

Usage in Project: Used to connect your local Flask project to the GitHub repository created online.

6. git push -u origin main

Feature: Pushes your local commits to GitHub for the first time and sets the upstream branch.

Usage in Project: Uploaded your project code (app.py, templates, static folder, Dockerfile, etc.) to GitHub.

7. git pull origin main

Feature: Fetches and merges updates from GitHub into the local machine.

Usage in Project: Used when deployment changes or documentation updates were pushed from another system.

8. git status

Feature: Shows the current working state of the repository—tracked, untracked, staged, modified files.

Usage in Project: Used repeatedly to verify which changes were ready for commit during development.

9. git clone <repo-link>

Feature: Copies an entire GitHub repository to a new machine.

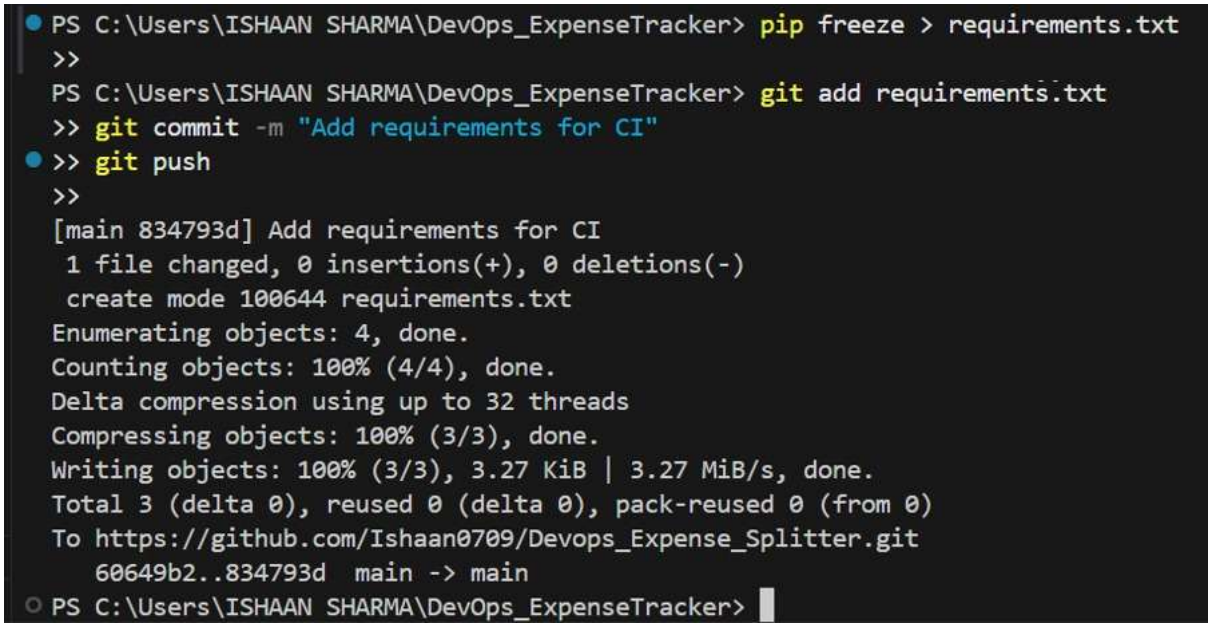
Usage in Project: Useful during testing or deploying the project on servers like AWS EC2.

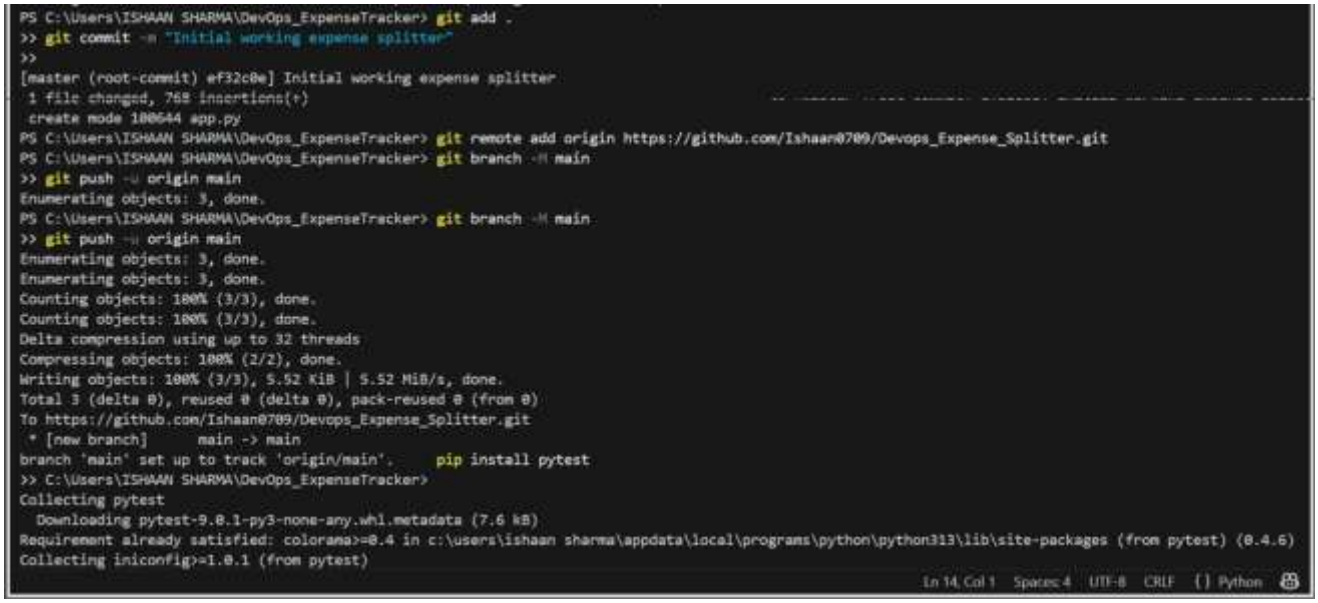
10. git push

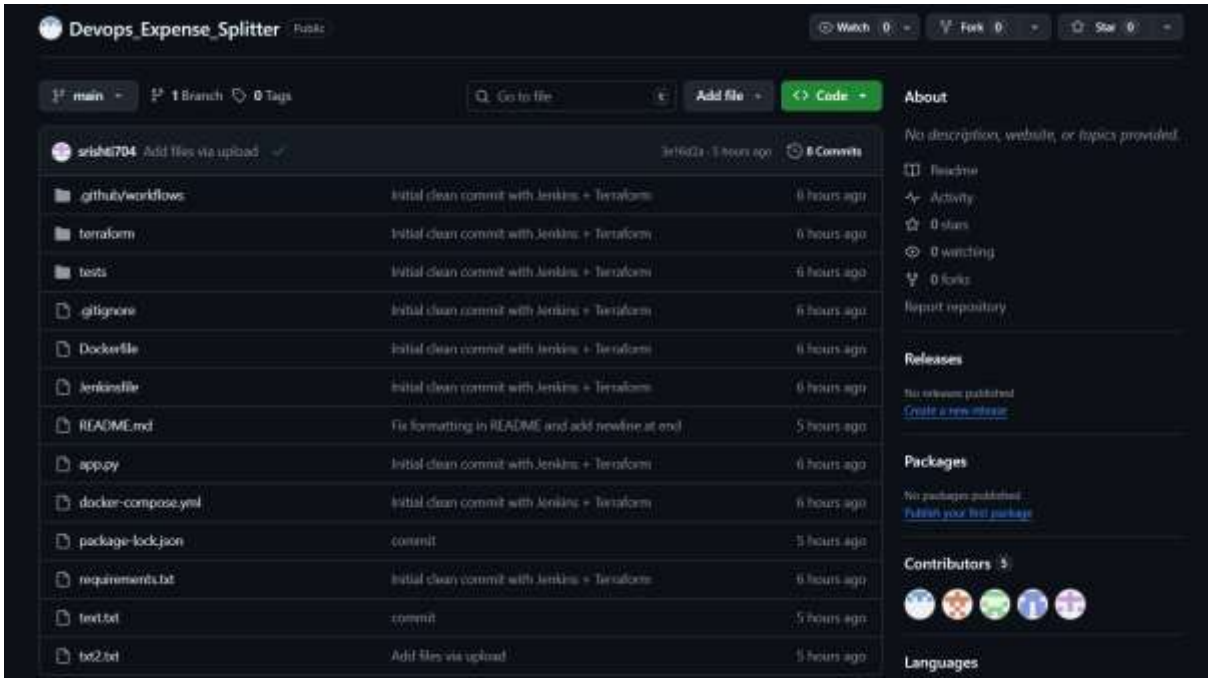
Feature: Uploads committed changes to GitHub.

Usage in Project: Used to continuously update the remote repository after new features or bug fixes.

5.2 Illustration of the Tool Used




PS C:\Users\ISHAAN SHARMA\DevOps_ExpenseTracker> git add .
>> git commit -m "Initial working expense splitter"
>>
[master (root-commit) ef32c0e] Initial working expense splitter
1 file changed, 768 insertions(+)
create mode 100644 app.py
PS C:\Users\ISHAAN SHARMA\DevOps_ExpenseTracker> git remote add origin https://github.com/Ishaan0709/Devops_Expense_Splitter.git
PS C:\Users\ISHAAN SHARMA\DevOps_ExpenseTracker> git branch -M main
>> git push -u origin main
Enumerating objects: 3, done.
PS C:\Users\ISHAAN SHARMA\DevOps_ExpenseTracker> git branch -M main
>> git push -u origin main
Enumerating objects: 3, done.
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Counting objects: 100% (3/3), done.
Delta compression using up to 32 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 5.52 KiB | 5.52 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Ishaan0709/Devops_Expense_Splitter.git
* [new branch] main -> main
branch 'main' set up to track 'origin/main'. pip install pytest
>> C:\Users\ISHAAN SHARMA\DevOps_ExpenseTracker>
Collecting pytest
Downloading pytest-9.0.1-py3-none-any.whl.metadata (7.6 kB)
Requirement already satisfied: colorama>=0.4 in c:\users\ishaan sharma\appdata\local\programs\python\python313\lib\site-packages (from pytest) (0.4.6)
Collecting iniconfig>=1.0.1 (from pytest)
Ln 14, Col 1 Spaces: 4 UTF-8 CRLF {} Python


Devops_Expense_Splitter
Public
Watch 0
Fork 0
Star 0
main
1 Branch
0 Tags
Go to file
Add file
Code
srishti704 Add files via upload
8 Commits
.github/workflows
terraform
tests
.gitignore
Dockerfile
Jenkinsfile
README.md
app.py
docker-compose.yml
package-lock.json
requirements.txt
text.txt
txt2.txt
About
No description, website, or topics provided.
Releases
Packages
Contributors 5
Languages

6. PyTest

PyTest is a testing framework for Python that makes it easy to write and run tests. It helps developers ensure their code works correctly and catches bugs early.

pytest is useful because:

- It can automatically discover and run tests
- Provides clear error messages when tests fail
- Supports fixtures to set up reusable test data
- Can run tests in any Python project, big or small
- Integrates with CI/CD pipelines for automated testing

Common Use Cases of pytest

- Unit testing (testing individual functions or classes)
- Integration testing (testing how parts of the system work together)
- Regression testing (making sure old features still work)
- Continuous integration setups (automated testing on GitHub Actions, GitLab, etc.)

6.1 Main Features

In the Expense Tracker project:

- pytest was used to test the backend routes (like /add-expense, /delete-expense)
- It verified the status codes (like 200 OK or 404 Not Found)
- It checked that calculations and summaries in the database were correct
- Automated tests helped catch bugs before deploying the app to AWS
- Tests could be run locally or in CI/CD to ensure the app is stable

6.2 Commands

1. Testing of Core Calculation Logic

PyTest was implemented to verify the correctness of the main functions used for splitting expenses, such as:

Equal Split Calculation, Custom Amount Split, Percentage Split, Balance Summary Calculation. Example Test Command: pytest
Feature Used: Simple assert statements to validate expected outputs.

2. Testing Flask Routes:The Flask application routes were tested using PyTest's Flask test client, which allows route testing without running the server.

Example: client = app.test_client()

response = client.post("/add_expense", data=test_data)

assert response.status_code == 200

Feature Used:

✓ Route behavior verification

✓ Checking correct HTTP status responses

✓ Ensuring form input is processed correctly

3. Using Fixtures for Test Setup: Test fixtures were used to create a reusable Flask test client across multiple test cases.

Example:

```
@pytest.fixture
def client():
    app.testing = True
    return app.test_client()
```

Feature Used:

✓ Clean and reusable testing environment

✓ Centralized test setup

4. Input Validation Testing: PyTest was used to ensure that invalid or incomplete inputs are properly rejected.

Example:

with pytest.raises(ValueError):

    calculate_split(-50)

Feature Used:

✓ Exception handling validation

✓ Detecting invalid values, incorrect percentages, or missing names

5. Coverage and Execution

All test cases were executed using PyTest's command-line interface.

Commands Used:

pytest -v

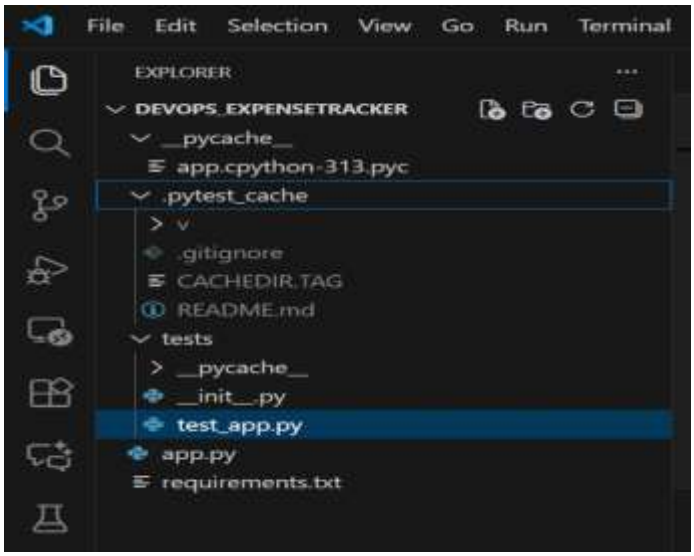
pytest --maxfail=1

Feature Used:

✓ Detailed test reporting

✓ Early failure stop to debug faster

6.3 Illustration of the Tool Used



7. GitHub Actions

GitHub Actions is a CI/CD (Continuous Integration / Continuous Deployment) tool built into GitHub.
It allows you to automate tasks like testing, building, and deploying your code whenever you make changes.

GitHub Actions is useful because:

- It can automatically run tests on every push or pull request
- Helps catch errors early before deployment
- Supports custom workflows using YAML files
- Can deploy code to servers or cloud platforms automatically
- Integrates with other tools like Docker, AWS, and Slack

Common Use Cases of GitHub Actions

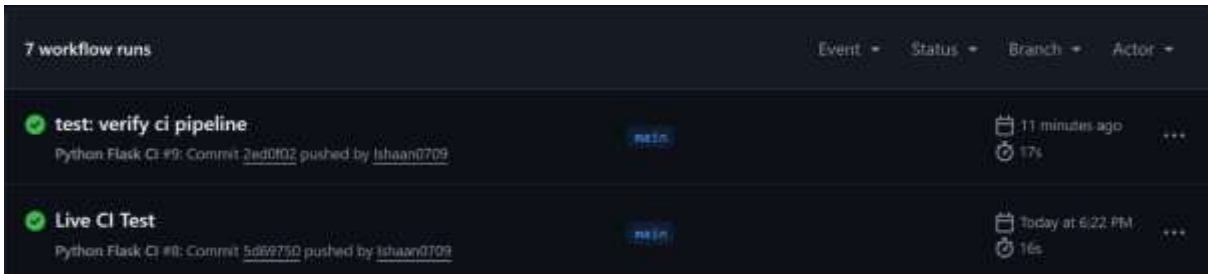
- Run automated tests on Python, Node.js, Java, or any language
- Build and package applications
- Deploy websites or APIs to cloud servers
- Automate repetitive tasks like linting, code formatting, or notifications

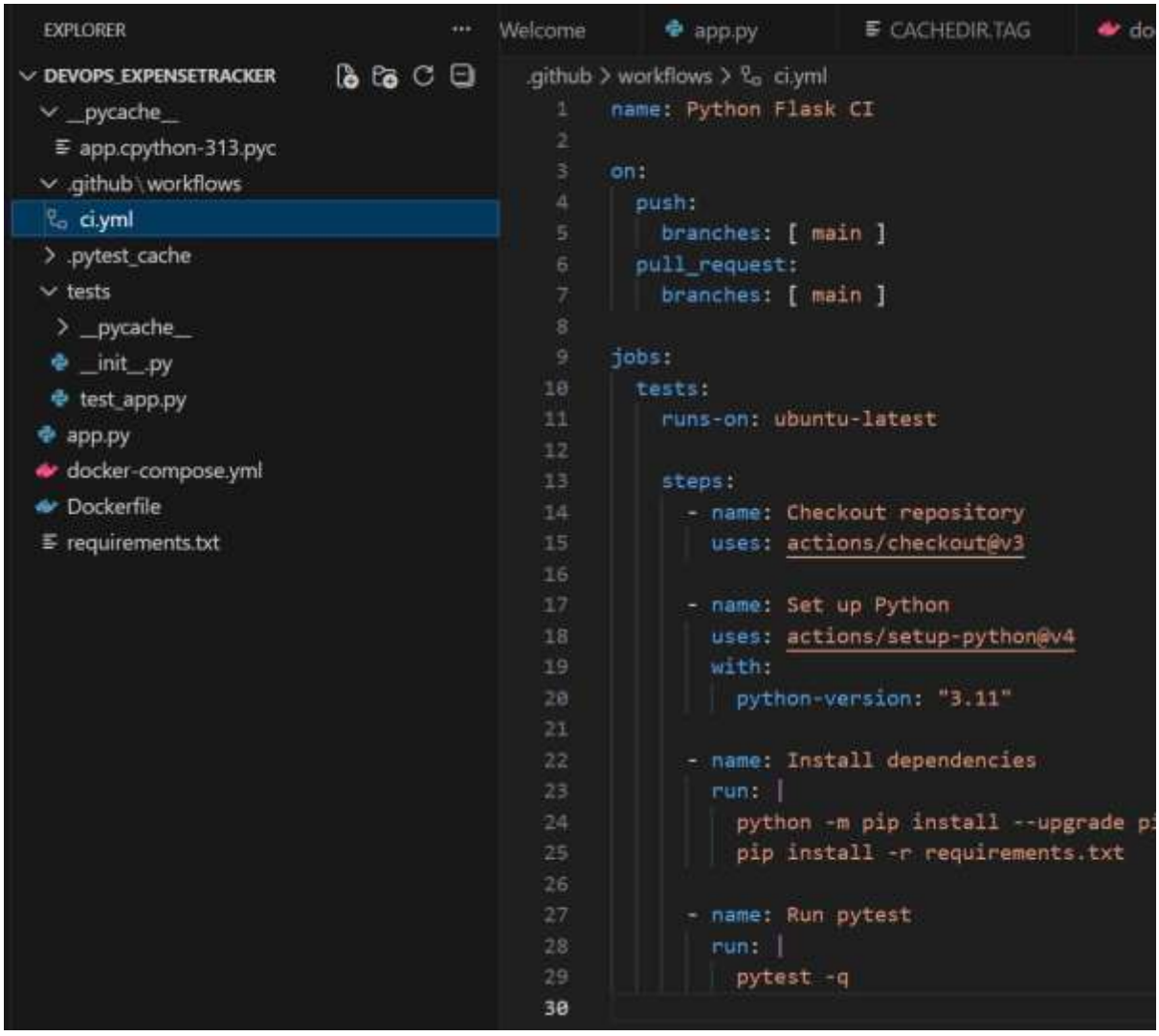
7.1 Main Features

In the Expense Tracker project:

- We set up a workflow YAML file to define automated actions
- On every push to GitHub, GitHub Actions automatically:
    - Installed Python dependencies
    - Ran pytest tests to check the backend logic
- This ensured the app remained stable after each code change
- It helped maintain reliability, so we could safely deploy to AWS

7.3 Illustration of the Tool Used





8. Docker & DockerCompose

Docker is a platform that packages applications with all their dependencies into containers, ensuring the app runs the same way on any machine. This eliminates “it works on my machine” issues.

Docker Compose is a tool that helps run multiple containers together (like the app and a database) using a single configuration file (docker-compose.yml). It makes it easy to start, stop, and manage multi-container setups.

Common Uses: Deploy web apps, run apps with databases, manage development environments, and automate container orchestration.

In this project:

- Docker packaged the Flask app with Python and all dependencies into a container, making it portable and consistent.
- Docker Compose allowed the app and SQLite database (if needed) to run together seamlessly, simplifying local testing and deployment.

7.1 Main Features

1. Containerization

Feature: Runs your app inside isolated, portable containers.

Usage: Your Python app runs inside a container using a Dockerfile like:

FROM python:3.9

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

CMD ["python", "app.py"]

Ensures: Same environment everywhere (PC, AWS, GitHub Actions)

No dependency conflicts

2. Image Building

Feature: Docker converts your project into a reusable "image".

Usage in your project:

docker build -t my-python-app .

This image is used to run the app locally, deploy to AWS, or run in CI/CD.

3. Isolation

Feature: Each container has its own environment, packages, and filesystem.

Usage: Your app does not depend on laptop-installed Python packages — everything lives in Docker.

4. Port Mapping

Feature: Exposes container ports to your machine or AWS.

Usage: docker run -p 8000:8000 my-python-app

Allows browser access:

localhost → http://localhost:8000

AWS → http://EC2_PUBLIC_IP:8000

5. Environment Variables

Feature: Pass secure config to containers.

Usage example:

docker run -e API_KEY=12345 my-python-app

Used for configs like DB passwords, API keys, etc.

DOCKER COMPOSE – FEATURES & USAGE

1. Multi-Container Orchestration

Feature: Run multiple services together (e.g., app + database).

Usage: Your project uses a docker-compose.yml like:

```
version: "3.8"
services:
  app:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    depends_on:
      - db

  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: testdb
```

This runs:

Python app

MySQL database

Together with one command.

2. Single Command Startup

Feature: Start ALL services with one simple command.

Usage:

docker-compose up --build

Starts entire project without manual setup.

3. Service Dependencies

Feature: Ensures containers start in the correct order.

Usage:

depends_on:

- db

Your app waits for DB before starting.

4. Volume Mounting (Live Reload)

Feature: Syncs your project folder with the container.

Usage:

volumes:

- .:/app

This allows:

live code updates

No need to rebuild every time

Faster development

5. Automatic Networking

Feature: All containers get connected automatically on the same network.

Usage:

Your app connects to DB using service name:

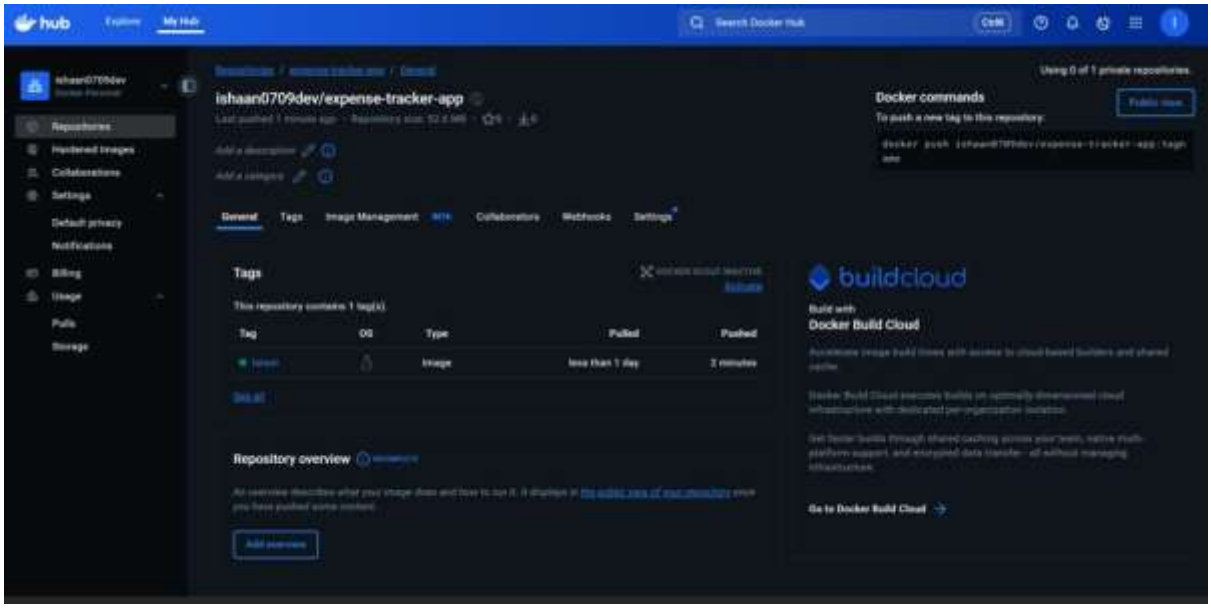
host="db"

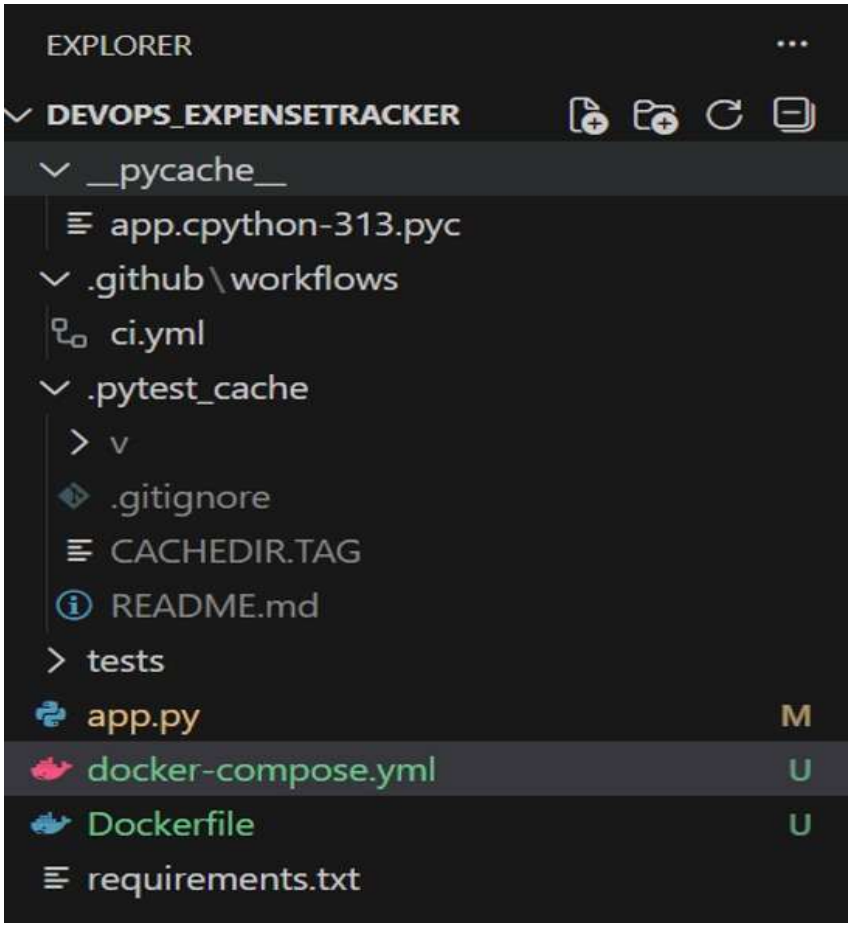
Works because Docker Compose creates:

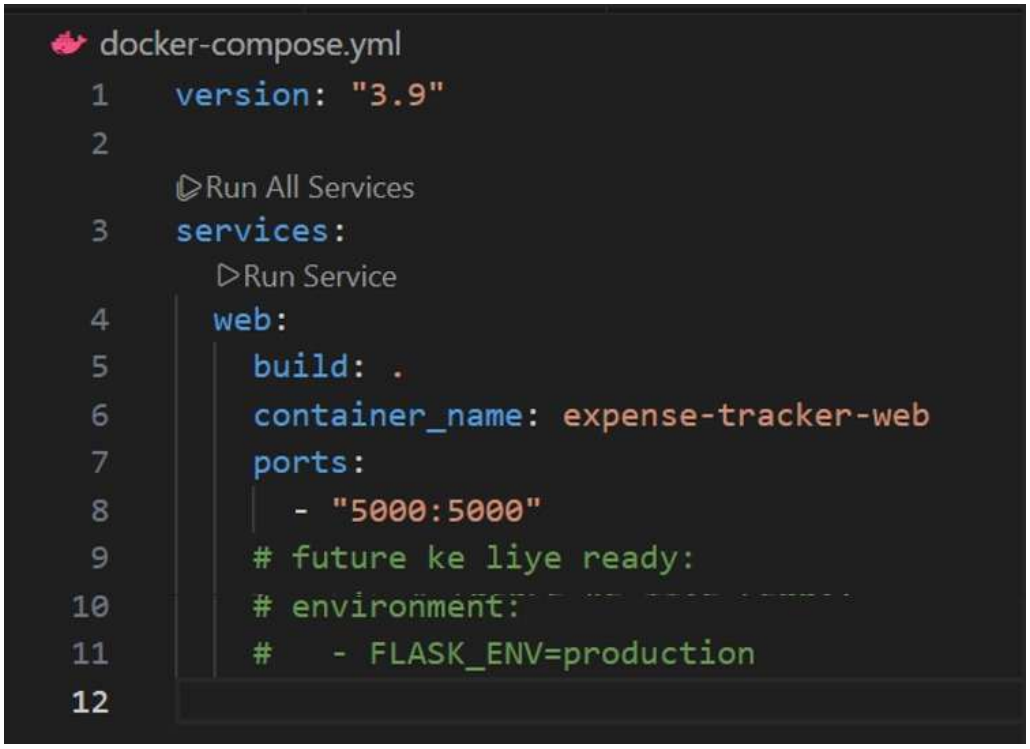
app <--> db

Private network, no manual config needed.

8.2 Illustration of the Tool Used




EXPLORER
DEVOPS_EXPENSETRACKER
__pycache__
app.cpython-313.pyc
.github\workflows
ci.yml
.pytest_cache
v
.gitignore
CACHEDIR.TAG
README.md
tests
app.py
M
docker-compose.yml
U
Dockerfile
U
requirements.txt


docker-compose.yml
version: "3.9"
Run All Services
services:
Run Service
web:
build: .
container_name: expense-tracker-web
ports:
- "5000:5000"
future ke liye ready:
environment:
- FLASK_ENV=production

9. AWS(EC2)

AWS EC2 is a cloud service that provides virtual servers (“instances”) to run applications online. Instead of using your local machine, you can host your project on a cloud server that’s always on and accessible from anywhere.

Common Uses:

- Hosting web applications
- Running backend services or databases
- Testing/deploying projects in a production-like environment
- Scaling applications without buying physical hardware

In this project:

- EC2 was used to host the Expense Tracker Flask app online.
- The app runs inside the EC2 instance, making it accessible via a public URL.
- Security rules (inbound ports) were configured to allow HTTP/Flask traffic.
- Combined with Docker, EC2 ensured the app runs consistently in the cloud just like locally.

9.1 Main Features

1. Scalable Compute Capacity

Feature: EC2 allows you to choose the amount of CPU, RAM, and storage your app needs.

Usage: In your project, EC2 enables you to choose an appropriate instance type (e.g., t2.micro, t2.small) depending on your app’s traffic and requirements. For small apps, you might use a smaller instance, but for heavy apps, you can scale up.

2. Flexible Instance Types

Feature: EC2 provides different instance types based on various workloads (e.g., general-purpose, compute-optimized).

Usage: For your app, a t2.micro or t2.small instance might be chosen as it's lightweight and fits the project requirements for small-to-medium-scale applications.

3. Pay-As-You-Go Pricing

Feature: EC2 uses a pay-per-use model. You are charged for only the time and resources your instance uses.

Usage: With your project, you only pay for the hours your instance runs. This helps reduce costs, as you can start and stop your instance based on need.

4. Automatic Scaling (Auto Scaling)

Feature: EC2 can automatically scale your application's compute capacity when the traffic spikes or drops.

Usage: In your project, you can set Auto Scaling to increase or decrease the number of EC2 instances running your app depending on the traffic.

5. Security Groups & Networking

Feature: EC2 uses security groups (firewalls) to control the incoming and outgoing traffic to the instances.

Usage: For your app, you can configure security groups to allow HTTP (port 80) for web traffic and SSH (port 22) for remote access.

6. Elastic IP Addresses

Feature: EC2 offers Elastic IP addresses that remain static and can be reassigned to other instances.

Usage: You can assign an Elastic IP to your instance to make sure your app always has a fixed IP address, even if the instance is restarted.

7. Integration with Other AWS Services

Feature: EC2 integrates easily with other AWS services like S3 (for file storage), RDS (for databases), and CloudWatch (for monitoring).

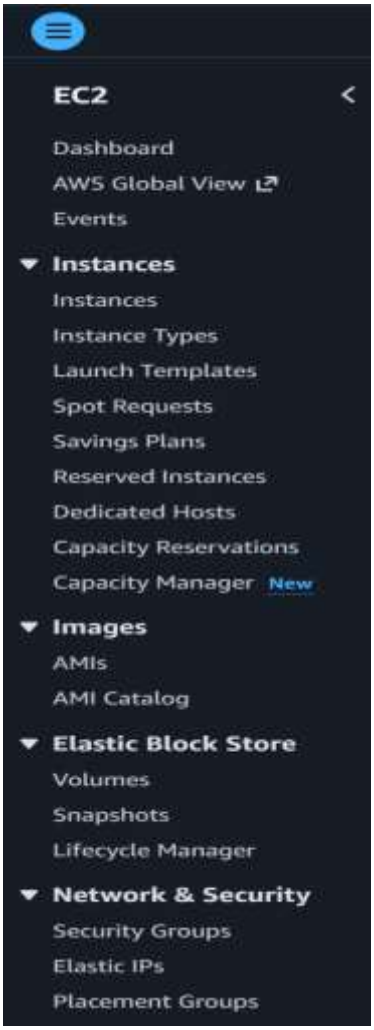
Usage: For example, your app can store data in S3, manage its database with RDS, and use CloudWatch to monitor performance.

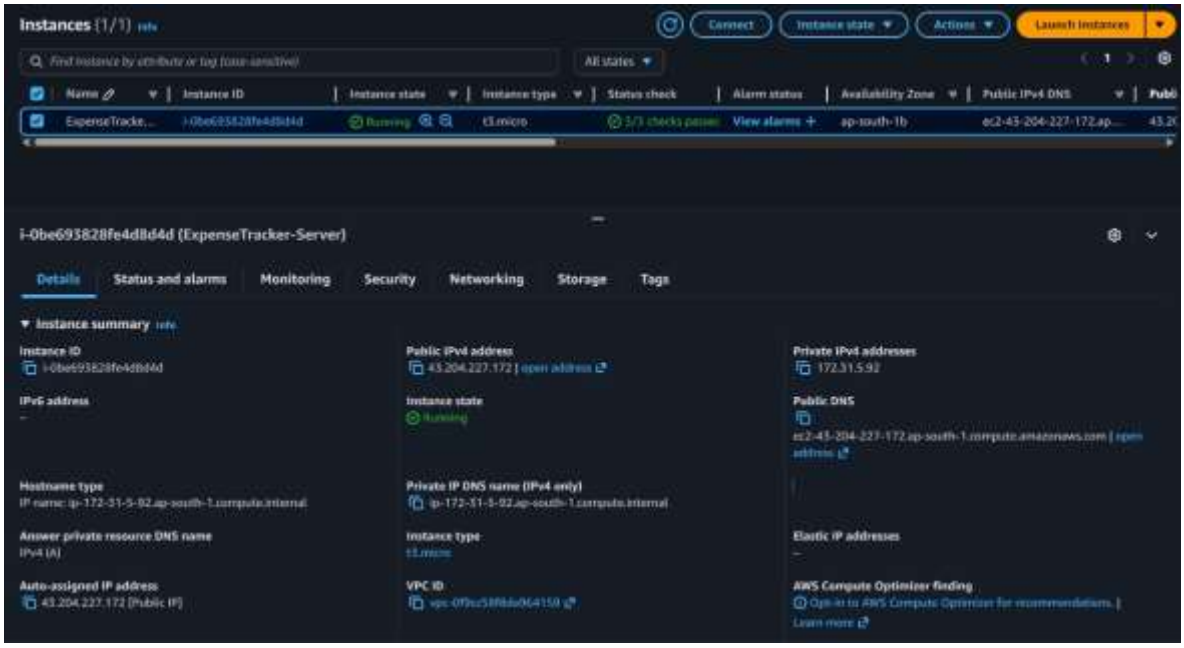
8. SSH Access for Secure Management

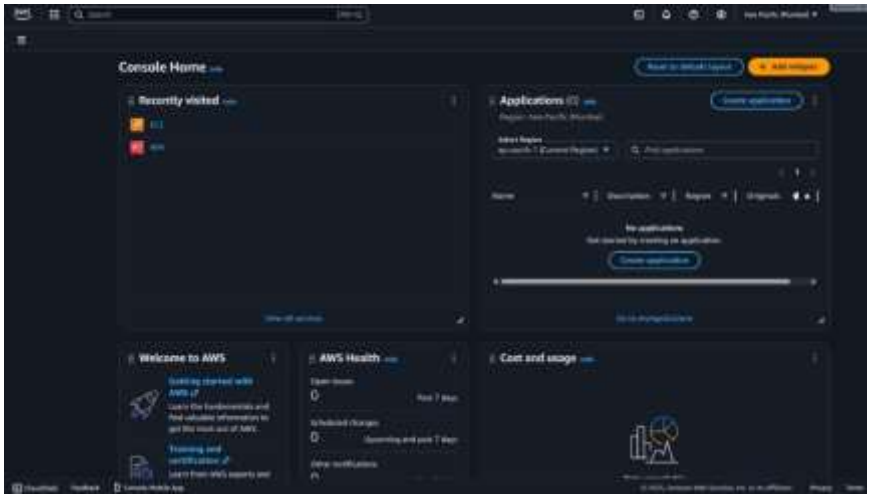
Feature: EC2 provides SSH access to instances for remote configuration and management.

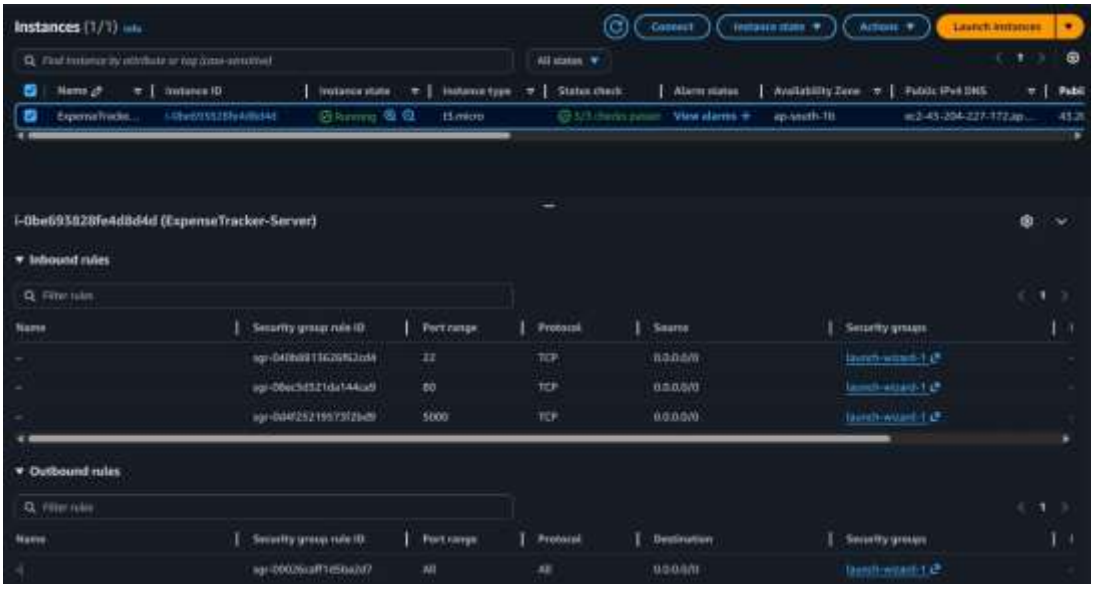
Usage: You can SSH into your EC2 instance using the private key (.pem file) to install dependencies, configure the environment, or troubleshoot.+

9.2 Illustration of the Tool Used




Instances (1/1)
Connect
Instance state
Actions
Launch instances
Find instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public IPv4 DNS
ExpenseTracke...
i-0be693828fe4d8d4d
Running
t3.micro
3/3 checks passed
View alarms +
ap-south-1b
ec2-43-204-227-172.ap...
i-0be693828fe4d8d4d (ExpenseTracker-Server)
Details
Status and alarms
Monitoring
Security
Networking
Storage
Tags
Instance summary
Instance ID
i-0be693828fe4d8d4d
Public IPv4 address
43.204.227.172 | open address
Private IPv4 addresses
172.31.5.92
IPv6 address
Instance state
Running
Public DNS
ec2-43-204-227-172.ap-south-1.compute.amazonaws.com | open address
Hostname type
IP name: ip-172-31-5-92.ap-south-1.compute.internal
Private IP DNS name (IPv4 only)
ip-172-31-5-92.ap-south-1.compute.internal
Answer private resource DNS name
IPv4 (A)
Instance type
t3.micro
Elastic IP addresses
Auto-assigned IP address
43.204.227.172 (Public IP)
VPC ID
AWS Compute Optimizer finding
Opt-in to AWS Compute Optimizer for recommendations.
Learn more


Console Home
Reset to default layout
Add widgets
Recently visited
Applications
Create application
No applications
Get started by creating an application.
Create application
Welcome to AWS
AWS Health
Cost and usage


Instances (1/1)
Connect
Instance state
Actions
Launch instances
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public IPv4 DNS
ExpenseTracke...
t3.micro
View alarms +
ap-south-1b
i-0be693828fe4d8d4d (ExpenseTracker-Server)
Inbound rules
Filter rules
Name
Security group rule ID
Port range
Protocol
Source
Security groups
TCP
0.0.0.0/0
launch-wizard-1
Outbound rules
Destination

10. Terraform

Terraform is an infrastructure-as-code (IaC) tool that allows you to define, provision, and manage cloud resources using configuration files. It enables you to automate the creation and management of infrastructure (like servers, databases, networks) in a reproducible, efficient manner.

Common Uses:

- Automating the provisioning of cloud infrastructure (servers, networks, databases)
- Managing infrastructure resources in a version-controlled manner
- Ensuring that infrastructure is consistent across environments
- Scaling resources based on demand (e.g., creating EC2 instances or S3 buckets on AWS)

In this project:

- Terraform was used to automate the creation and management of the AWS EC2 instance hosting the Expense Tracker app.
- With Terraform, we wrote configuration files to define the resources (like EC2 instances, security groups, etc.), and then we used the terraform apply command to automatically create and manage these resources.
- This approach ensures that infrastructure can be easily replicated, modified, and tracked in version control (GitHub), making the app's deployment consistent and repeatable.

10.1 Main Features

1. Infrastructure as Code (IaC): Terraform allows you to define and manage infrastructure using code, which ensures repeatable, version-controlled, and automated resource provisioning. This approach makes it easier to scale, modify, and maintain infrastructure across different environments.

2. Resource Management: Terraform simplifies the management of various AWS resources such as EC2 instances, security groups, and VPCs. By using Terraform, you can define the entire infrastructure configuration,

ensuring that all resources are created and managed consistently. It also provides easy updates and modifications to the infrastructure without manual intervention.

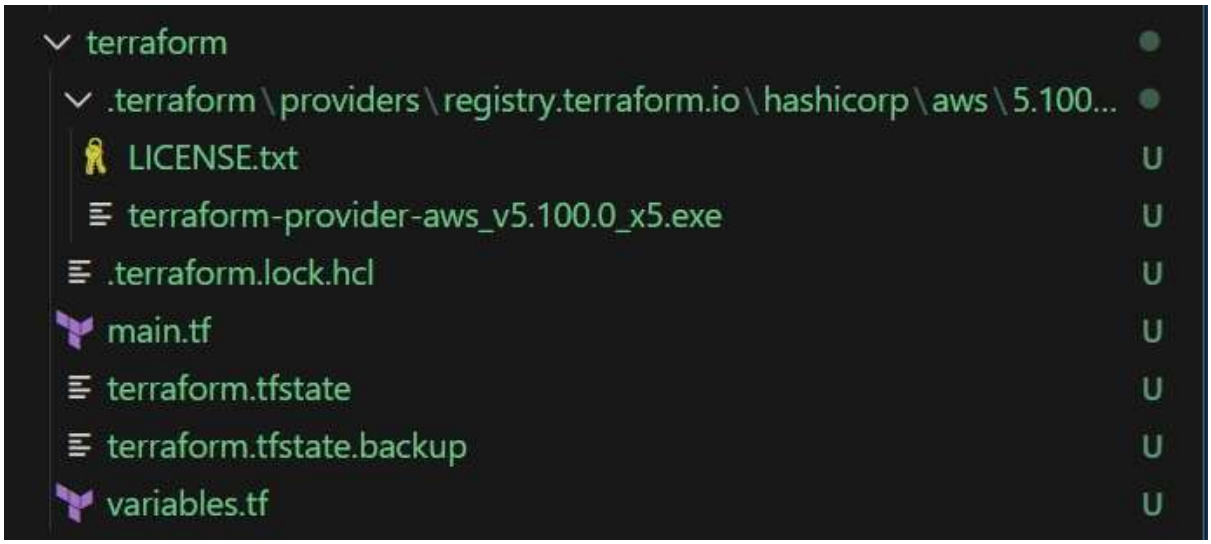
3. State Management: Terraform tracks the current state of the infrastructure through a state file, which allows it to compare the existing setup with the desired state. When running terraform plan, Terraform generates an execution plan that shows which resources need to be created, updated, or destroyed. This ensures that any changes are precisely controlled and predictable.

4. Automating Infrastructure with User Data: Terraform can automate the provisioning of resources, including configuring EC2 instances with specific tasks such as installing software or running applications. This is done using the user_data feature, which allows a script to be executed automatically when the EC2 instance is launched, ensuring that all necessary applications and services are installed and running without manual setup.

5. Multi-Environment Consistency: Terraform promotes consistency across multiple environments (e.g., development, staging, and production) by using variables and modules. This allows you to create reusable configurations that can be applied to different environments with minimal adjustments. It simplifies the management of infrastructure across different stages of development.

6. Version Control Integration: Terraform configurations can be stored in version control systems such as Git, ensuring that infrastructure changes are tracked and auditable. This makes it easier to collaborate on infrastructure, review changes, and revert to previous versions if needed. Using Git to store Terraform configurations provides transparency and allows the infrastructure to be versioned along with the application code.

10.2 Illustration of the Tool Used

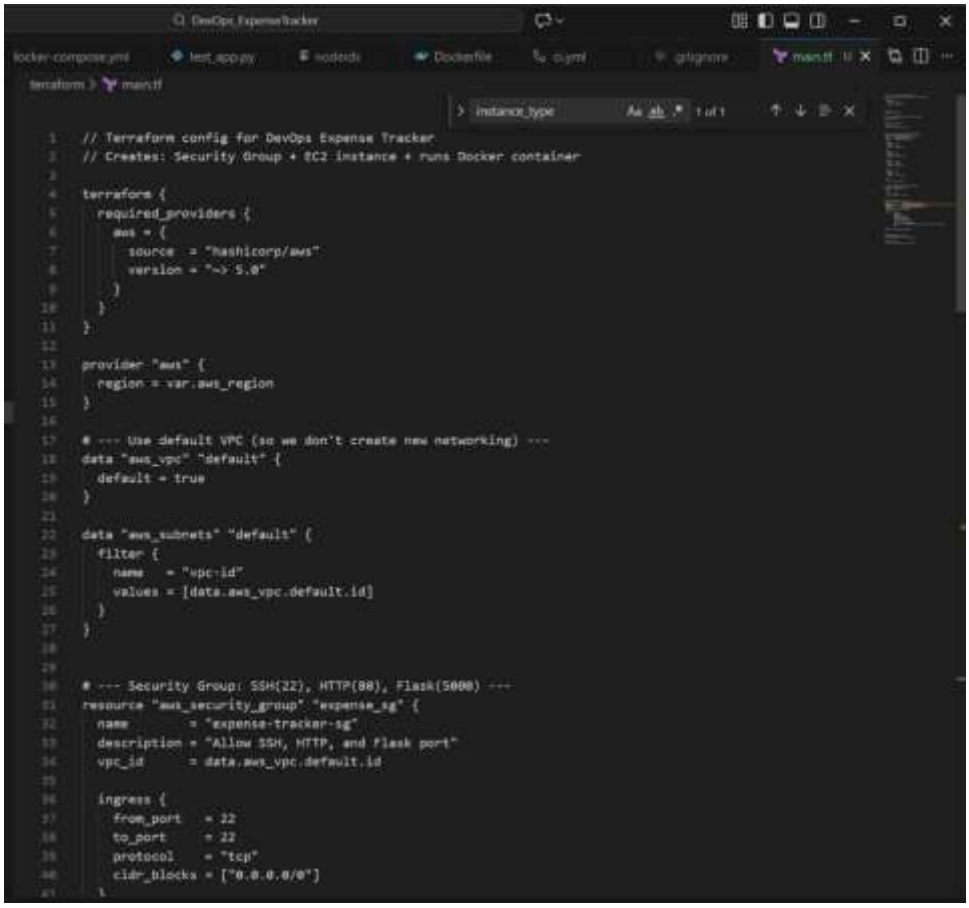



```
PS C:\Users\Iterraform plan vOps_ExpenseTracker\terraform>
>>
data.aws_vpc.default: Reading...
data.aws_ami.amazon_linux_2023: Reading...
data.aws_vpc.default: Read complete after 1s [id=vpc-0f9cc58f8da964159]
data.aws_subnets.default: Reading...
aws_security_group.expense_sg: Refreshing state... [id=sg-09731e08f1ff9102f]
data.aws_subnets.default: Read complete after 0s [id=ap-south-1]
data.aws_ami.amazon_linux_2023: Read complete after 1s [id=ami-02d05f76acbed4e3e]

Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.expense_server will be created
  + resource "aws_instance" "expense_server" {
      + ami                                  = "ami-02d05f76acbed4e3e"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = true
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t3.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "ishaan-key"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
```



```
// Terraform config for DevOps Expense Tracker
// Creates: Security Group + EC2 instance + runs Docker container

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

# --- Use default VPC (so we don't create new networking) ---
data "aws_vpc" "default" {
  default = true
}

data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}


# --- Security Group: SSH(22), HTTP(80), Flask(5000) ---
resource "aws_security_group" "expense_sg" {
  name        = "expense-tracker-sg"
  description = "Allow SSH, HTTP, and Flask port"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
data.aws_vpc.default: Reading...
data.aws_ami.amazon_linux_2023: Reading...
data.aws_ami.amazon_linux_2023: Read complete after 1s [id=ami-0188073bacbed4e3e]
data.aws_vpc.default: Read complete after 1s [id=vpc-
data.aws_subnets.default: Reading...
aws_security_group.expense_sg: Refreshing state... [id=sg-
data.aws_subnets.default: Read complete after 0s [id=ap-south-1]

Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.expense_server will be created
  + resource "aws_instance" "expense_server" {
      + ami                                  = "ami-0188073bacbed4e3e"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = true
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t3.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "jenkin-key"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
```

11. Jenkins

Jenkins is an open-source automation server used primarily for continuous integration (CI) and continuous delivery (CD). It automates the process of building, testing, and deploying software, making it easier to integrate changes and monitor the development process.

Common Uses:

- Continuous Integration (CI): Automatically building and testing code changes every time they are pushed to a version control system like Git
- Continuous Delivery (CD): Automatically deploying the code to production or staging environments after passing tests
- Automating repetitive tasks: Running unit tests, code linting, deployment pipelines
- Monitoring and reporting: Automatically sending notifications when a build fails, when tests fail, or when deployment is successful

In this project:

- Jenkins was used to automate the testing and deployment pipeline.
- Every time a change was pushed to GitHub, Jenkins automatically triggered a build and ran the PyTest tests to ensure that the app was working correctly.
- Once the tests passed, Jenkins deployed the app to the AWS EC2 instance, automating the entire deployment process.
- Jenkins was configured with webhooks to listen for changes in the GitHub repository and trigger the necessary tasks (build, test, deploy).

11.1 Main Features

1) Continuous Integration (CI): Jenkins is primarily used for continuous integration, where developers' changes are automatically tested and integrated into the project’s main branch. This helps in identifying integration issues early, ensuring that new code works well with the existing codebase.

Usage: In this project, Jenkins automatically pulls the latest changes from the GitHub repository whenever a commit or pull request is made. It then runs tests using pytest to verify the correctness of the code before any further steps like Docker image creation and deployment are executed.

2) Automated Testing: Jenkins can automate the execution of unit tests, integration tests, and functional tests, ensuring the quality and correctness of the code. By integrating testing into the CI pipeline, Jenkins ensures that only code that passes tests gets deployed.

Usage: In this project, Jenkins runs pytest as part of the pipeline to automatically test the Flask application every time there is a code change, ensuring that the app's functionality remains intact.

3) Docker Image Building and Deployment: Jenkins automates the process of building and deploying Docker images, making it a critical tool for containerized applications. By integrating Jenkins with Docker, you can automatically build new images whenever code changes are detected and push them to a Docker registry.

Usage: Jenkins automatically builds a new Docker image for the Expense Tracker app after the tests pass. It then pushes the new image to DockerHub, from where it can be pulled and deployed to the EC2 instance on AWS.

4)Pipeline as Code: Jenkins supports the concept of defining build pipelines as code, typically through a Jenkinsfile. This allows you to define all the stages of the CI/CD process, including code checkout, testing, building, and deployment, in a version-controlled file.

Usage:The pipeline for this project is defined in a Jenkinsfile, which specifies the various stages such as checkout, testing, Docker image building, and deployment. This ensures that the process is repeatable, versioned, and consistent across different environments.

5) Integration with Version Control Systems: Jenkins integrates seamlessly with version control systems like GitHub, Bitbucket, and GitLab, allowing it to automatically trigger jobs when there are changes in the code repository.

Usage: For this project, Jenkins is integrated with GitHub. When a commit is pushed or a pull request is created, Jenkins automatically triggers the CI pipeline to build, test, and deploy the application.

6) Monitoring and Reporting: Jenkins provides comprehensive monitoring and reporting capabilities, allowing you to view the status of builds, tests, and deployments. Jenkins can send notifications about build status, failures, and test results via email or other communication channels.

Usage: After each build, Jenkins provides feedback on whether the tests passed or failed. This helps the development team quickly address any issues before moving to the next stage of the pipeline.

Scalability: Jenkins supports distributed builds, allowing you to run builds across multiple machines (agents) to improve performance and scalability. This is useful in large teams or for complex projects requiring parallel execution of tasks.

Usage:For this project, Jenkins can be scaled to handle additional tasks such as parallel tests or builds, ensuring the system remains responsive as the project grows.

7) Extensibility via Plugins: Jenkins offers a large selection of plugins that extend its functionality. These plugins can help with everything from integration with third-party services to providing advanced reporting features.

Usage:Jenkins integrates with several plugins in this project, including:

Docker Plugin for Docker container management

GitHub Plugin for integration with GitHub repositories

Email Extension Plugin for sending notifications after build completions

11.2 Illustration of the Tool Used

```
PS C:\Users\ISHAAN SHARMA> docker run -d --name jenkins -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts
Unable to find image 'jenkins/jenkins:lts' locally
lts: Pulling from jenkins/jenkins
a37b858bb47a: Pull complete
c27509c3e53b: Pull complete
65e4ba8066bc: Pull complete
e4beac64dffa: Pull complete
13cc39f8244a: Pull complete
5dc87232677a: Pull complete
33300af18dd0: Pull complete
744b4792e883: Pull complete
dc2a77f462ea: Pull complete
05a7d9a8b608: Pull complete
7718ff514822: Pull complete
8d2a75b252b2: Pull complete
Digest: sha256:7b1c378278279c8688efd6168c25a1c2723a6bd6f0420beb5ccefabee3cc3bb1
Status: Downloaded newer image for jenkins/jenkins:lts
d72e596f9dcb911360aa3f6245a5cedfd0c5e0a45e359d0b357cfc3d209574e3
```

12. Monitoring

- Consistent Monitoring Across Environments: Docker ensures consistency between different environments (e.g., local development, staging, and production). By using Docker for monitoring, we were able to keep track of the same performance metrics across all environments, ensuring that the app behaved similarly both locally and in the cloud.
- Efficient Resource Utilization: With Docker's isolation, it was important to monitor the application's resource usage to ensure that it was efficiently using CPU, memory, and disk space. Using docker stats and cAdvisor, we were able to optimize resource usage, which is crucial when running the application on cloud infrastructure like AWS EC2.
- Scalability and Load Management: One of Docker's advantages is the ease of scaling applications by adding more containers. Monitoring was critical for determining when to scale the app by adding more instances of the Docker container to handle increased traffic. With proper monitoring, we ensured that the app could handle varying loads without performance degradation.
- Reliability in Cloud Deployment: By monitoring the Docker container on the EC2 instance, we ensured that the Expense Tracker app was always running as expected. If there was any issue, like the container stopping unexpectedly or using excessive resources, it could be detected early, and corrective action could be taken immediately.

13. Result

The DevOps Expense Tracker project successfully implemented a fully automated pipeline using modern DevOps tools, ensuring the smooth operation of both the application and its underlying infrastructure.

Infrastructure Automation with Terraform:

Terraform was used to provision and manage AWS resources such as EC2 instances, Security Groups, and VPCs, ensuring that the infrastructure is consistently deployed with minimal manual intervention. The infrastructure was successfully created, and the EC2 instance was configured to run the Flask

application inside a Docker container.

CI/CD Pipeline with Jenkins:

A Jenkins pipeline was established to automate the process of building, testing, and deploying the application. The pipeline pulled the latest code from GitHub, ran tests using pytest, built the Docker image, and pushed it to DockerHub. The image was then deployed to the AWS EC2 instance. The Jenkins pipeline successfully handled all these tasks, allowing for seamless continuous integration and delivery.

Dockerization of the Application:

The Flask application was containerized using Docker, making it portable and easy to deploy. The Docker image was built and pushed to DockerHub, where it could be pulled and run on any machine, including the EC2 instance on AWS.

Monitoring Setup with Docker Stats and cAdvisor:

The monitoring aspect was handled by Docker’s built-in docker stats command for basic container statistics. Additionally, cAdvisor was used for more advanced monitoring, providing real-time resource usage graphs for the containers. This ensured that the health and performance of the application could be continuously monitored.

Application Deployment on AWS:

The application was successfully deployed to AWS using EC2, with proper security groups configured to allow necessary traffic (SSH, HTTP, and Flask). The application was accessible through the EC2 instance's public IP, confirming that the cloud deployment was successful.

14. Challenges Faced

- EC2 Instance Type Compatibility: Challenge: The initial EC2 instance type (t2.micro) was not eligible for the AWS free tier in certain regions, which resulted in deployment errors.
- Docker Configuration and Networking: Challenge: When running Docker containers on AWS EC2, managing port forwarding and container networking proved to be tricky, especially when configuring it for both Flask and Docker in a secure manner.
- Jenkins Integration with GitHub: Challenge: Integrating Jenkins with GitHub required the proper setup of webhooks and ensuring that Jenkins had the necessary access to the GitHub repository to automatically trigger builds.
- Terraform State Management: Challenge: Managing Terraform's state file (which tracks the resources deployed) across multiple environments was initially confusing, especially in terms of collaboration and versioning.
- Container Health and Monitoring: Challenge: Implementing monitoring and ensuring that the containers were performing as expected required setting up a reliable monitoring tool.

14. Conclusion

- The project successfully demonstrated a complete DevOps pipeline, incorporating best practices of automation, continuous integration, and deployment using a variety of tools:
- Terraform automated the infrastructure provisioning on AWS, ensuring consistency and scalability across environments.
- Jenkins facilitated automated testing and deployment, improving the speed and reliability of development workflows.
- Docker helped containerize the Flask application, ensuring that it could be deployed consistently across environments.
- Monitoring was implemented using Docker stats and cAdvisor, allowing for real-time tracking of application performance.

This project has proven the effectiveness of using modern DevOps tools to streamline development processes, improve collaboration, and enhance the scalability and maintainability of applications. By integrating infrastructure automation, continuous delivery, and monitoring, the project provided a robust solution to deploy and maintain a cloud-based web application.