Portfolio Data Aggregator Test Suite

Software Engineers:

- *Cody Strange – Software Engineer*

Abstract

The Portfolio Data Aggregator Test Suite is a backend-focused project that simulates the validation and monitoring of a financial data aggregation API. It demonstrates core software testing principles including schema validation, edge case handling, mock-based testing, and performance analysis. The suite targets two critical endpoints—`/api/portfolio` and `/api/assets`—with structured test coverage using Pytest, data integrity enforcement using Pydantic, and HTTP mocking via the `responses` library.

This project incorporates continuous integration via GitHub Actions to automate testing and coverage reporting. Additionally, it includes performance testing using Locust to evaluate API responsiveness under simulated user load. All code is modular and maintainable

Organization

Coding Standards

Commenting

Functions

- Brief description of function
- Parameters
- Return Values

Classes

- Brief description of class

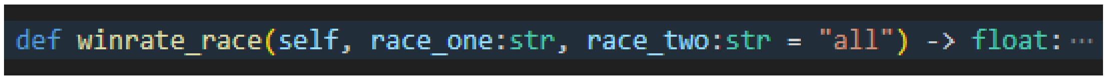
ChatGPT

- I highly recommend letting ChatGPT do most of the commenting
- Double check any comments by ChatGPT

Naming Conventions

- Classes: CapWords
- Functions: snake_case
- Variables: snake_case
- Constants: ALLCAPS
- Files/Folders: snake_case

Type Safety

- Every function should have the parameters types listed and the return type of the function listed
- 

```python
def winrate_race(self, race_one:str, race_two:str = "all") -> float:···
```

Software Requirements

Programming Languages and Frameworks

- Programming Languages
  - Python
- Test Frameworks & Libraries
  - Pytest
  - Response or Pytest-httpserver
  - Pydantic or jsonschema
  - Pytest-cov
  - Flake8 or ruff
  - Requests or httpx
- Data Handling
  - CSV File Export

Development Tools

- VS Code

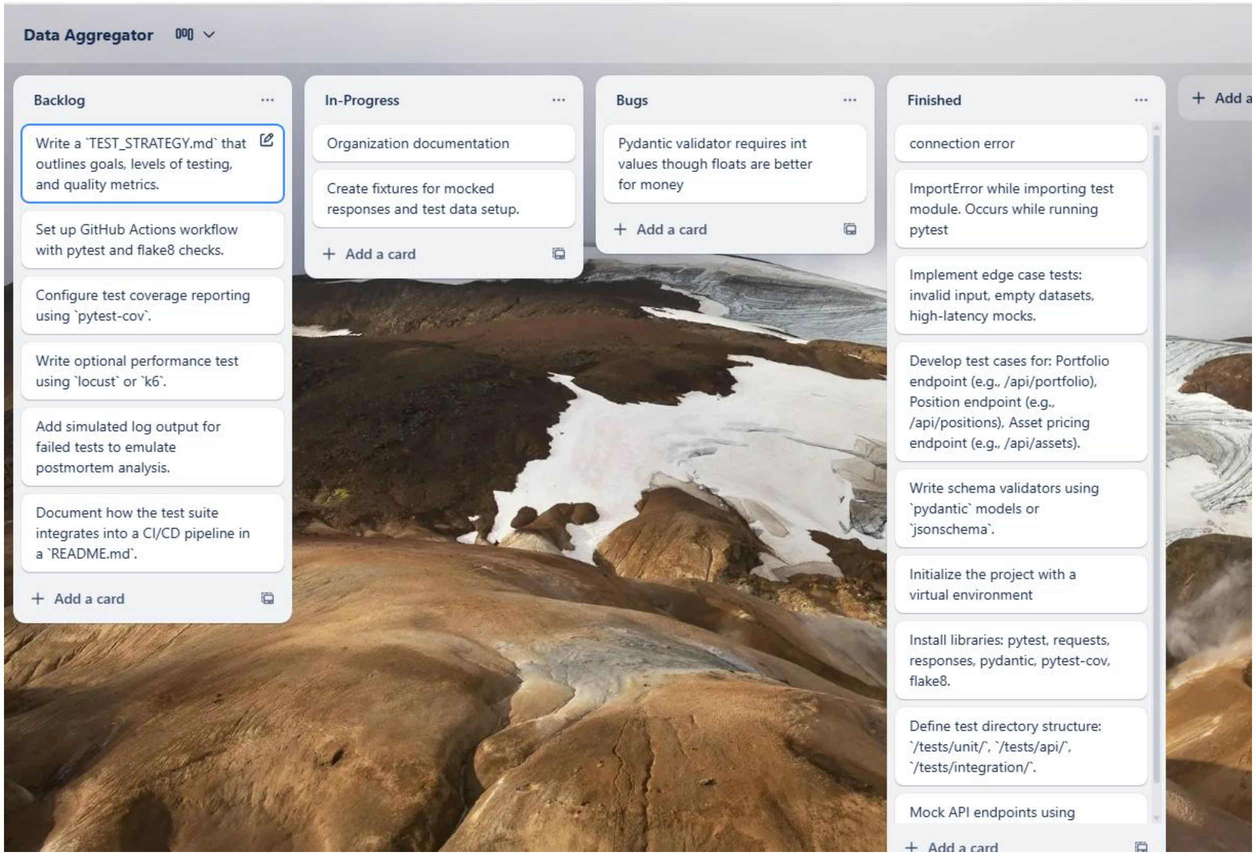
- Venv
- Node.js + npm
- Browser

Version Control
- Git
- Github

Virtual Environment
- Venv

Additional Tools
- Lucidchart
- ChatGPT

Backlog
Description: This contains the tasks that we have completed each iteration as well as what we are currently working on, what known bugs exist, and what remains to be done. We add to the backlog as we discover new tasks but ignore them until we finish what we are working on.



Backlog
- Write a `TEST_STRATEGY.md` that outlines goals, levels of testing, and quality metrics.
- Set up GitHub Actions workflow with pytest and flake8 checks.
- Configure test coverage reporting using `pytest-cov`.
- Write optional performance test using `locust` or `k6`.
- Add simulated log output for failed tests to emulate postmortem analysis.
- Document how the test suite integrates into a CI/CD pipeline in a `README.md`.

In-Progress
- Organization documentation
- Create fixtures for mocked responses and test data setup.

Bugs
- Pydantic validator requires int values though floats are better for money

Finished
- connection error
- ImportError while importing test module. Occurs while running pytest
- Implement edge case tests: invalid input, empty datasets, high-latency mocks.
- Develop test cases for: Portfolio endpoint (e.g., /api/portfolio), Position endpoint (e.g., /api/positions), Asset pricing endpoint (e.g., /api/assets).
- Write schema validators using `pydantic` models or `jsonschema`.
- Initialize the project with a virtual environment
- Install libraries: pytest, requests, responses, pydantic, pytest-cov, flake8.
- Define test directory structure: `/tests/unit/`, `/tests/api/`, `/tests/integration/`.
- Mock API endpoints using

Requirements Gathering

Functional Requirements

Description: These requirements are the specific goals that our application should be able to meet.

- Test Portfolio Retrieval Endpoint
    - The test suite must verify that the /api/portfolio endpoint returns
        - A valid JSON response
        - Correct HTTP status codes (200 OK, 404 for missing portfolios)
        - Accurate portfolio metadata (name, ID, owner, total value)
        - A list of position IDs that are part of the portfolio
- Test Asset Pricing Endpoint
    - The /api/assests enpoint must:
        - Return current pricing for a list of asset symbols
        - Include data fields such as asset symbol, last price, data, and exchange
        - Handle unknown symbols gracefully (return 404 or empty result)
- Validate JSON Schema for All Endpoints
    - All responses must match defined Pydantic or JSON schema models
    - Schema mismatches must be detected and cause test failure

Non-Functional Requirements

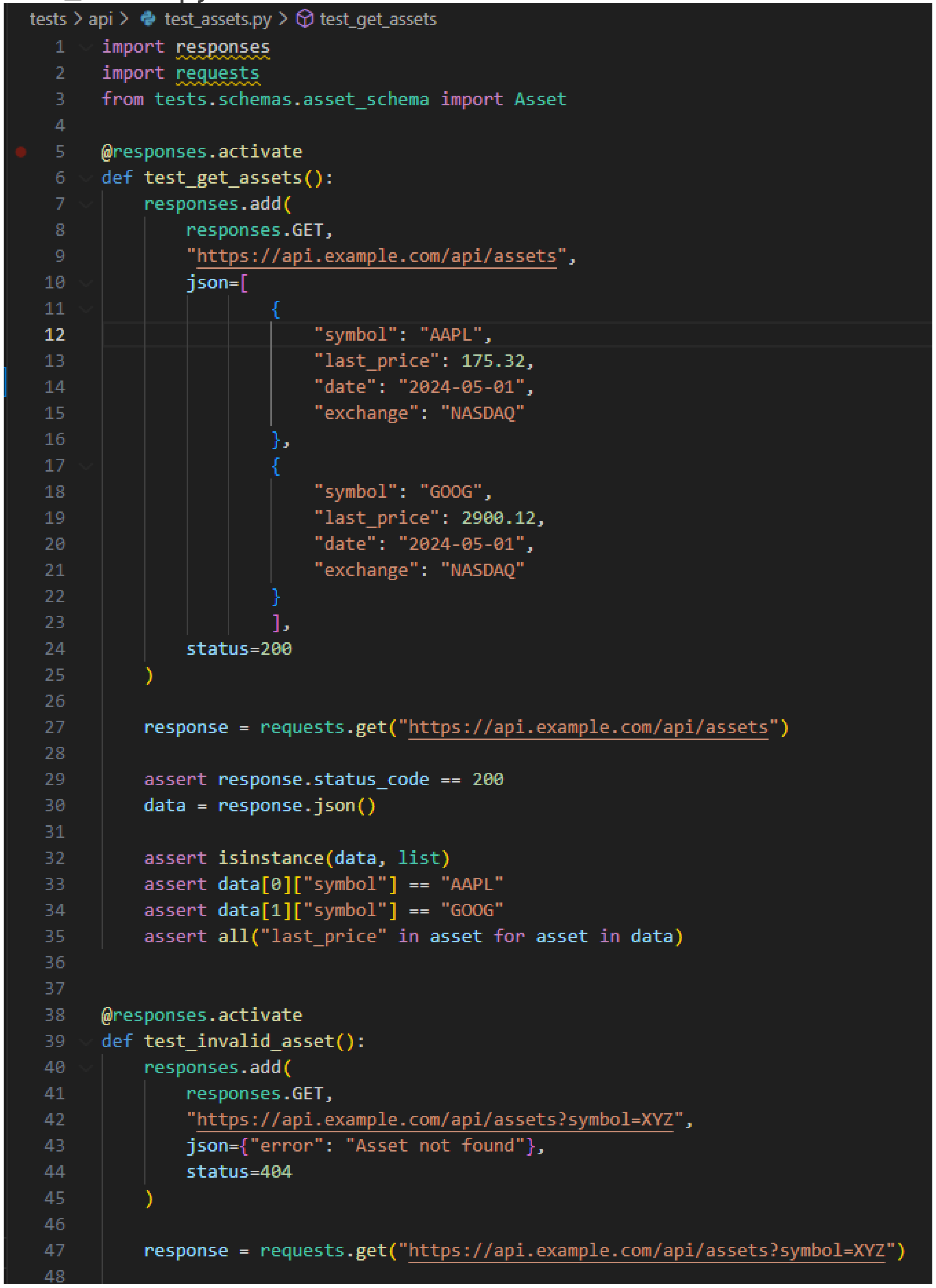
Description: These requirements are the specific goals that our application should be able to meet.

- Performance and Latency Testing
    - The suite must simulate high request volume for /api/portfolio using locust or k6
    - Measure response times and ensure they remain under a defined threshold (<500ms for 95% of request)
- Error and Exception Coverage
    - The system must return appropriate status codes for:
        - Malformed requests (400)
        - Unauthorized access (401)
        - Resource not found (404)
        - Server errors (500)
- Automated CI integration
    - The test suite must be runnable in a CI pipeline and return:
        - Exit codes indicating pass/fail
        - Coverage reports
        - Logs for failed tests

Development

Code Snippets

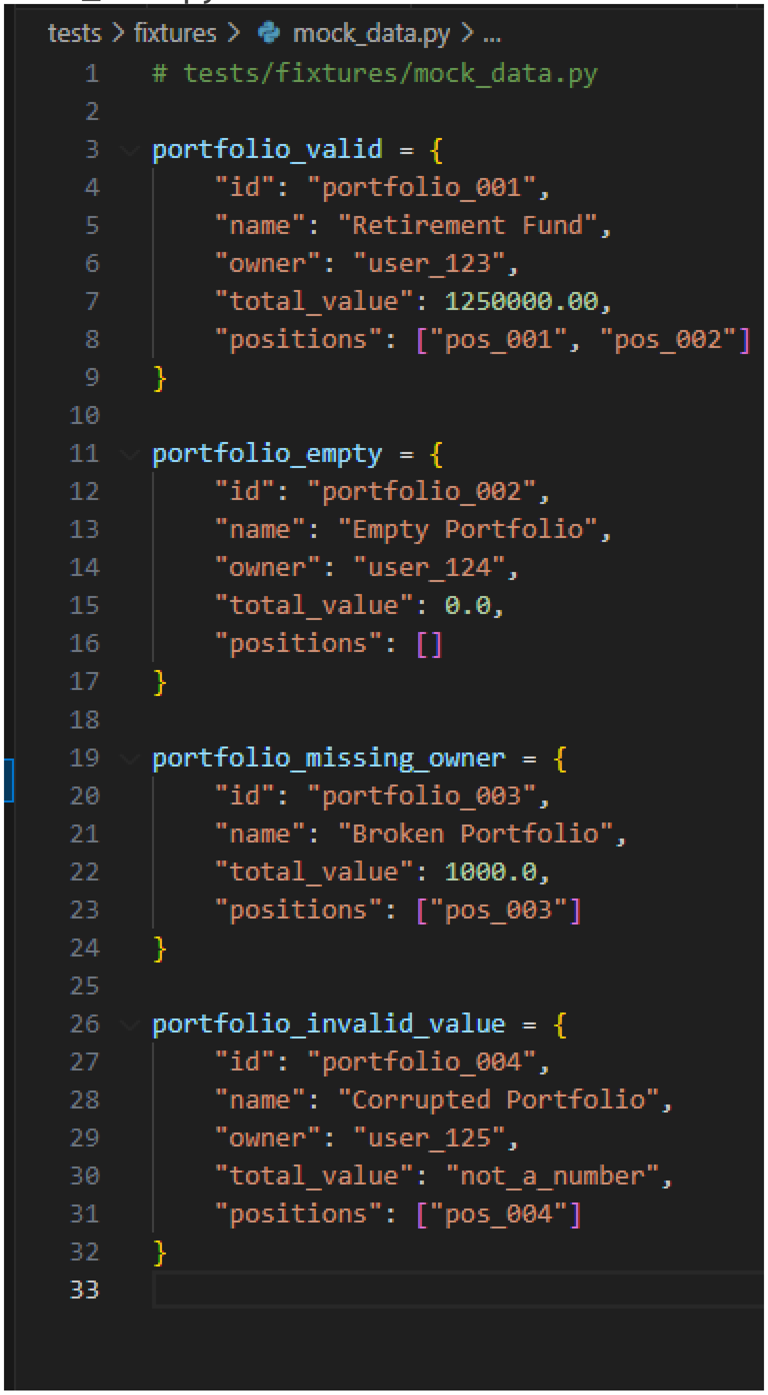
Test_assets.py



tests > api > test_assets.py > test_get_assets

```python
import responses
import requests
from tests.schemas.asset_schema import Asset

@responses.activate
def test_get_assets():
    responses.add(
        responses.GET,
        "https://api.example.com/api/assets",
        json=[
                {
                    "symbol": "AAPL",
                    "last_price": 175.32,
                    "date": "2024-05-01",
                    "exchange": "NASDAQ"
                },
                {
                    "symbol": "GOOG",
                    "last_price": 2900.12,
                    "date": "2024-05-01",
                    "exchange": "NASDAQ"
                }
                ],
        status=200
    )

    response = requests.get("https://api.example.com/api/assets")

    assert response.status_code == 200
    data = response.json()

    assert isinstance(data, list)
    assert data[0]["symbol"] == "AAPL"
    assert data[1]["symbol"] == "GOOG"
    assert all("last_price" in asset for asset in data)


@responses.activate
def test_invalid_asset():
    responses.add(
        responses.GET,
        "https://api.example.com/api/assets?symbol=XYZ",
        json={"error": "Asset not found"},
        status=404
    )

    response = requests.get("https://api.example.com/api/assets?symbol=XYZ")

```

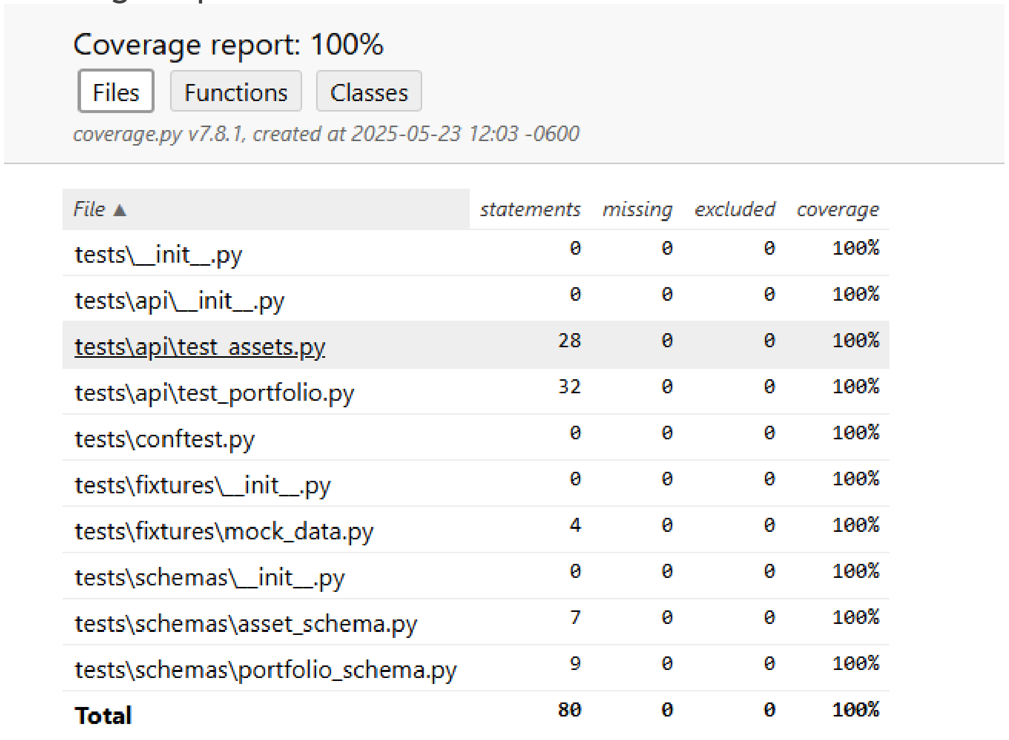
Mock_data.py



```python
# tests/fixtures/mock_data.py

portfolio_valid = {
    "id": "portfolio_001",
    "name": "Retirement Fund",
    "owner": "user_123",
    "total_value": 1250000.00,
    "positions": ["pos_001", "pos_002"]
}

portfolio_empty = {
    "id": "portfolio_002",
    "name": "Empty Portfolio",
    "owner": "user_124",
    "total_value": 0.0,
    "positions": []
}

portfolio_missing_owner = {
    "id": "portfolio_003",
    "name": "Broken Portfolio",
    "total_value": 1000.0,
    "positions": ["pos_003"]
}

portfolio_invalid_value = {
    "id": "portfolio_004",
    "name": "Corrupted Portfolio",
    "owner": "user_125",
    "total_value": "not_a_number",
    "positions": ["pos_004"]
}
```

Outputs

Coverage Report



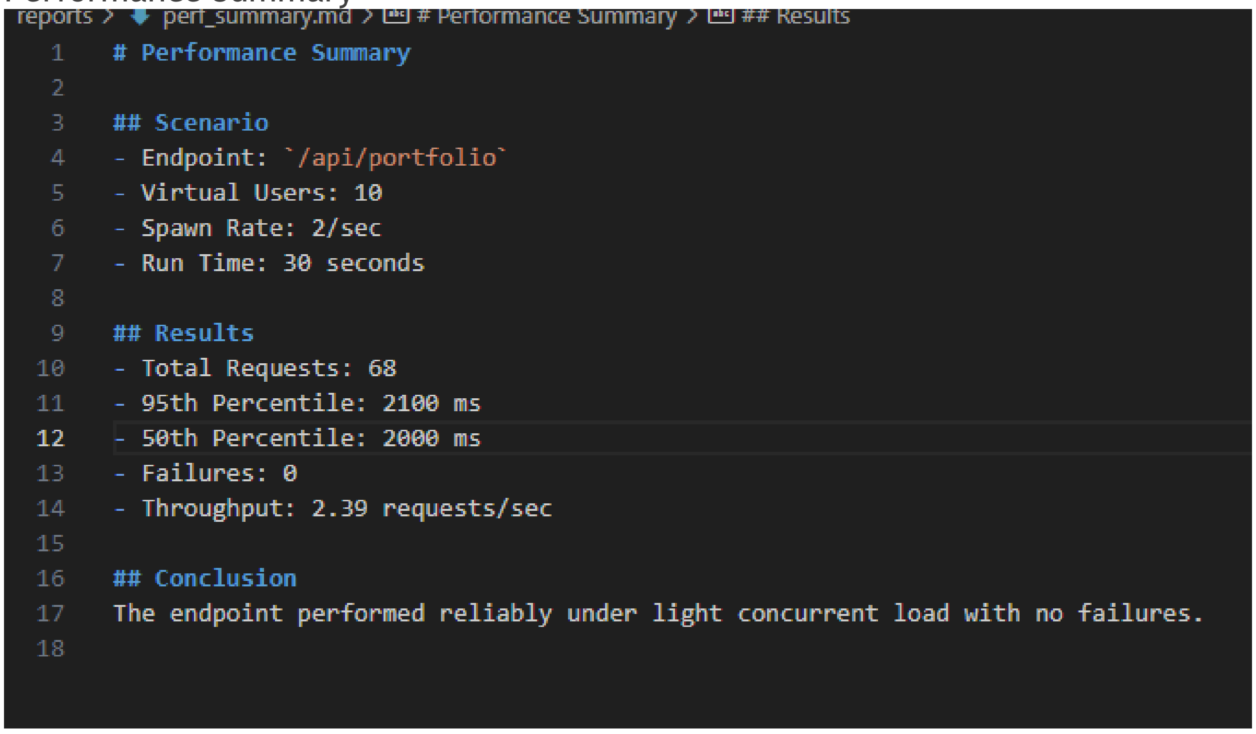
Coverage report: 100%

Files | Functions | Classes

coverage.py v7.8.1, created at 2025-05-23 12:03 -0600

File ▲	statements	missing	excluded	coverage
tests\__init__.py	0	0	0	100%
tests\api\__init__.py	0	0	0	100%
tests\api\test_assets.py	28	0	0	100%
tests\api\test_portfolio.py	32	0	0	100%
tests\conftest.py	0	0	0	100%
tests\fixtures\__init__.py	0	0	0	100%
tests\fixtures\mock_data.py	4	0	0	100%
tests\schemas\__init__.py	0	0	0	100%
tests\schemas\asset_schema.py	7	0	0	100%
tests\schemas\portfolio_schema.py	9	0	0	100%
Total	80	0	0	100%

Performance Summary



reports > perf_summary.md > # Performance Summary > ## Results

```
# Performance Summary

## Scenario
- Endpoint: `/api/portfolio`
- Virtual Users: 10
- Spawn Rate: 2/sec
- Run Time: 30 seconds

## Results
- Total Requests: 68
- 95th Percentile: 2100 ms
- 50th Percentile: 2000 ms
- Failures: 0
- Throughput: 2.39 requests/sec

## Conclusion
The endpoint performed reliably under light concurrent load with no failures.
```

Review

Overview

This project was a deep dive into backend-focused test automation and quality assurance. It involved designing a structured, schema-validated, and performance-aware test suite for a mock financial portfolio API. The goal was to simulate a real-world backend testing workflow, including automated validation, CI integration, test coverage tracking, and performance benchmarking.

What I Learned

- How to design testable API schemas using Pydantic for strict data validation.
- How to write modular, reusable Pytest tests that validate endpoints and edge cases.
- How to use the `responses` library to mock backend APIs for unit and integration tests.
- How to track code coverage with `pytest-cov` and interpret reports for quality insights.
- How to build a complete CI pipeline using GitHub Actions that runs tests and uploads artifacts.
- How to use Locust to simulate real-world load and analyze latency under concurrent traffic.

Struggles I Faced

- Understanding when to use mock data directly vs. validating responses through schemas.
- Learning the best practices for structuring a multi-layered test suite (api, fixtures, schemas).
- Dealing with test import errors due to missing `__init__.py` files or bad `PYTHONPATH` config.
- Locust's output format and tuning performance tests to simulate realistic usage patterns.
- GitHub Actions artifact upload errors due to incomplete coverage config or missing reports.

Successes

- Created a clean, well-structured test suite that includes schema validation, error handling, and edge case coverage.
- Built reusable mock data fixtures and used them across multiple test cases for consistency.
- Integrated continuous testing and coverage metrics with GitHub Actions workflows.
- Simulated real-world load using Locust and documented performance insights.
- Generated a professional README, test strategy, and performance summary to present the project clearly.