Dockerized Parallel Test Automation Framework with Allure Reporting

1. Project Overview

This project is designed to implement an automated testing framework that leverages Docker containers to run multiple API test suites in parallel. It aims to provide a scalable and isolated testing environment, ensuring comprehensive testing of various endpoints simultaneously. The project includes integrated Allure Reporting to visualize test results, making it easier to analyze failures and monitor test outcomes.

1.1 Purpose

The purpose of this project is to enable parallel test execution using Docker and to create detailed test reports using Allure. The goal is to reduce overall test execution time while maintaining test isolation and generating consolidated test reports for analysis.

1.2 Scope

The scope of this project includes:

- - Setting up a Dockerized testing environment with multiple containers executing API tests in parallel.
- - Implementing parallel testing using pytest-xdist to reduce overall execution time.
- - Using Allure Reports to generate detailed, interactive, and user-friendly test reports.
- - Handling environment variables and volume management for result storage and synchronization across containers.
- - Merging multiple Allure results into a single, consolidated report for better analysis and review.

2. Software Architecture

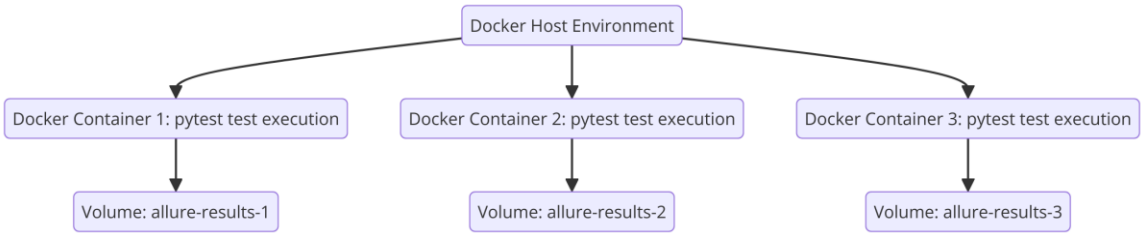
The architecture is built around a modular, containerized structure, enabling easy scalability and isolated execution environments. The key components include Docker containers for parallel test execution, volume management, and centralized Allure report generation.

2.1 Architecture Overview

Each Docker container runs an independent instance of the test suite using pytest and allure-pytest. The containers are created and managed using docker-compose, allowing for efficient orchestration and scaling.

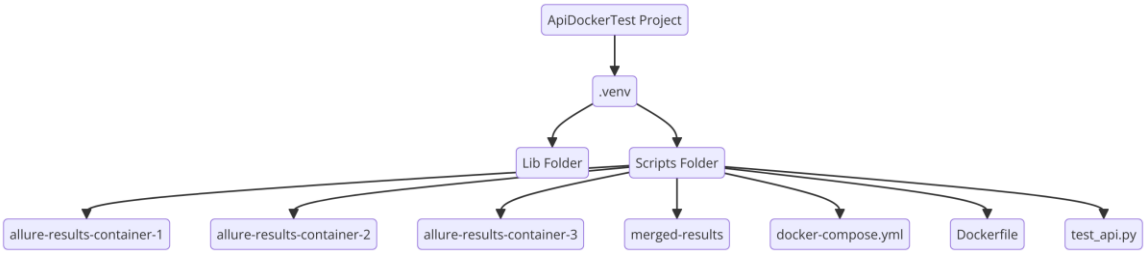
2.2 Architecture Diagram

The architecture diagram shows the interaction between Docker containers and the final consolidated Allure report:



3. Technical Implementation

3.1 Project Structure



3.2 Dockerfile Configuration

FROM python:3.8-slim

#Set the working directory inside the container

WORKDIR /app

#Copy the test script inside the container

COPY test_api.py .

#Install all the necessary library files

RUN pip install pytest pytest-xdist allure-pytest requests

```
# Define proper environment Variable
ENV RESULTS_DIR="/app/allure-results"

# Use the environment variable directly without shell syntax

CMD ["sh", "-c", "pytest -n 3 --alluredir=${RESULTS_DIR}"]
```

3.3 Docker Compose Configuration

```
version: '3.7'
services:
 test_service_1:
   build: .
   container_name: unittest_parallel_1
   environment:
    - RESULTS_DIR=/app/allure-results
   volumes:
    - ./allure-results-container1:/app/allure-results
 test_service_2:
   build: .
   container_name: unittest_parallel_2
   environment:
    - RESULTS_DIR=/app/allure-results
   volumes:
    - ./allure-results-container2:/app/allure-results
 test_service_3:
   build: .
   container_name: unittest_parallel_3
   environment:
    - RESULTS_DIR=/app/allure-results
   volumes:
    - ./allure-results-container3:/app/allure-results
```

3.4 Test Suite Implementation

```
import pytest
import requests
import allure


@allure.title("Test Api for multiple GET requets")
```

```
@pytest.mark.parametrize("url",[
  'https://jsonplaceholder.typicode.com/posts',
  'https://jsonplaceholder.typicode.com/comments',
  'https://jsonplaceholder.typicode.com/albums',
  'https://jsonplaceholder.typicode.com/todos'
])

def test_get_calls(url):
  with allure.step(f"sending GET request to {url}"):
    response = requests.get(url)
    allure.attach(f"Response: {response.text}",name="API Response",
attachment_type=allure.attachment_type.TEXT)
  with allure.step("Check if the status code is 200"):
    assert response.status_code == 200
```

4. Conclusion and Recommendations

The project successfully demonstrates how to leverage Docker for parallel test execution and Allure for comprehensive reporting. Future enhancements could include integration with a CI/CD pipeline (e.g., Jenkins or GitLab CI) for automated test execution and reporting.