Simulated\_robotic\_workstation

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
* 

## Software Requirements

Programming Languages and Frameworks

* Programming Languages
  + Python
* Python Libraries
  + Opencv-python
  + Matplotlib
  + Pygame
  + Ikpy
  + Flask
  + Sqlite3

Development Tools

* VS Code
* Venv

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.*

*A screenshot of a computer

AI-generated content may be incorrect.*

# Requirements Gathering

## Functional Requirements

Description: *These define what the system should do, its features and behaviors*

* Perception
  + The system will load and process 2D workspace images
  + The system will detect and classify predefined tools in an image
  + The system will return the type and x,y coordinates of each detected tool
* Motion Planning
  + The system will calculate a valid trajectory to reach the tool using inverse kinematics
  + The system will simulate the robotic arm movement from its base to the target tool location
  + The system will visually represent the motion on a 2D canvas
* Infrastructure and Interface
  + The system will allow the user to trigger tool detection and motion planning via a CLI or GUI
  + The system will log detection events and motion steps to a file or database
  + The system will display logs via a local web dashboard or console interface

## Non-Functional Requirements

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

* Performance
  + The tool detection process will complete within 2 seconds for a 1080p image
  + Motion planning and visualization will respond within 1 second after detection
* Maintainability
  + The codebase will follow consistent structure and include comments for all functions
  + Each module will be loosely coupled and independently testable

# Development

## Code Snippets

### Test\_assets.py

A screen shot of a computer program

AI-generated content may be incorrect.

### Mock\_data.py

A screen shot of a computer program

AI-generated content may be incorrect.

## Outputs

### Coverage Report

A screenshot of a report

AI-generated content may be incorrect.

### Performance Summary A screenshot of a computer AI-generated content may be incorrect.

# 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.