Smart Fuel Allocation API Test Plan

For

Syntech - Smart Fuel Allocation API Testing Exercise

by

Paul Smith

October 21, 2025

Table of Contents

1. INTRODUCTION

1.1 TEST OBJECTIVES

The primary objective of this testing is to validate that the Smart Fuel Allocation API operates according to the specified user story and acceptance criteria. The testing will validate that the system, as a cohesive unit, performs its core function.
Specific objectives include validating that the API:
• Provides delivery recommendations (volume and schedule) that accurately reflect the inputs (demand forecasts, current tank levels, and constraints).
• Supports the business goal of minimizing delivery costs, avoiding overstocking, and reducing fuel shortages.
• Correctly implements the three optimization modes ("Cost," "Availability," and "Balanced").
• Handles exception scenarios by returning appropriate alerts and explanatory notes when fallback logic is used (e.g., missing data or no available trucks).
• Adheres to all functional requirements defined in the Acceptance Criteria.

1.2 SCOPE OF TESTING

The scope of testing is limited to the single external RESTful API endpoint, `POST /api/allocate`. Testing will include the validation of inputs, request handling, and the structure and correctness of the API's output based on the provided user story requirements (Acceptance Criteria).

1.3 SYSTEM OVERVIEW

The Smart Fuel Allocation API is a RESTful API within the fictional FuelSync system. Its purpose is to assist fuel operations managers in planning fuel deliveries by analyzing station data (current tank levels, usage, and expected demand). The API provides recommendations on the delivery volume and schedule.

1.4 DEFINITIONS/ACRONYMS

Term	Definition
Requirement	Something that the system should do or be, based on user or business needs.
Regression Testing	Testing performed to ensure that previously functional parts of the software continue to work correctly after a change.
System Integration Testing (SIT)	Validating internal and external system interfaces to ensure the system works as a cohesive whole, built according to requirements.

Fallback Logic	Backup rules the system uses to provide helpful recommendations when resilient data is missing or stale.

1.5 REFERENCES

The Test Plan is based on the requirements and constraints detailed in the Smart Fuel Allocation API Testing Exercise document.

2. APPROACH

2.1 ASSUMPTIONS/CONSTRAINTS

2.1.1 Assumptions

• A base URL for the API endpoints is accessible for testing.
• The API endpoints are generally stable and ready for validation against the user story requirements.
• All necessary Python dependencies and the execution environment are installed and configured.

2.1.2 Constraints

• The testing relies only on input data specified in the request fields (`POST /api/allocate` inputs). External system interfaces (like delivery truck scheduling or historical data storage) are assumed to be working correctly, but their internal logic is not under test.

2.2 COVERAGE

Test coverage will focus on the specified requirements (Acceptance Criteria).

2.2.1 Requirements

All user requirements as specified in the Acceptance Criteria for the Smart Fuel Allocation API will be tested, specifically focusing on validating correct recommendations, handling of optimization modes, and error/alert responses.

2.3 TEST TOOLS

The primary test tool will be a **Python script** designed to send requests and validate API responses. Necessary Python libraries (e.g., `requests` for sending HTTP requests, `json` for handling responses) will be utilized.

2.4 TEST TYPE

The following types of testing will be performed:

• **Functional Testing:** Executing test cases based on the requirements to ensure the API allocates fuel correctly based on inputs (e.g., verifying `recommended_volume` correctness).
• **Boundary/Negative Testing:** Verifying how the API handles extreme, missing, or invalid inputs (e.g., `tank_level_percent` > 100, or mandatory fields missing).
• **Regression Testing:** If fixes are applied, previously passing tests must be rerun to ensure no new defects were introduced.

2.5 TEST DATA

Test data will be created specifically for this exercise to validate key scenarios outlined in the Acceptance Criteria. This data will include:
• **Standard Success Cases:** Valid combinations of inputs to check all optimization modes.
• **Boundary Cases:** Testing `delivery_window` edges and `tank_level_threshold` limits.
• **Failure/Alert Cases:** Specific data intended to trigger expected alerts and fallback logic (e.g., simulating stale telemetry data or high demand with low tank levels).
• **Unit Conversion Cases:** Data ensuring consistency when `demand_unit` is "gallons" vs. "liters".

3. PLAN

3.3 MAJOR TASKS AND DELIVERABLES

The major tasks align with the assignment deliverables:

Task	Deliverable(s)
Outline Approach	This Test Plan and Strategy
List Assumptions/Questions	List of assumptions/questions about the API
Develop Test Script	Python script that sends requests and validates responses
Document Submission	README file with instructions and summary of findings

3.4 ENVIRONMENTAL NEEDS

3.4.1 Test Environment

The testing environment requires a setup capable of executing the Python script and accessing the API.
Software:
• Python 3.x environment
• Required Python libraries (e.g., `requests`)
• Access to the Smart Fuel Allocation API (base URL)

4. FEATURES TO BE TESTED

Features are defined by the required API functionality, primarily validated via the `POST /api/allocate` endpoint.

1. **Core Recommendation Logic:**
    - Verify recommendations reflect demand forecasts and current tank levels.
    - Verify recommendations respect the specified `delivery_window` and `tank_level_threshold`.
    - Verify unit consistency: `recommended_volume` and `recommended_unit` match `demand_unit` (gallons/liters).
2. **Optimization Modes:**
    - Test standard behavior for "Cost" optimization (focus on cost minimization).
    - Test standard behavior for "Availability" optimization.
    - Test standard behavior for "Balanced" optimization.
3. **Response Structure and Data Integrity:**
    - Validate that all required fields are present in the response: `station_id`, `recommended_volume`, `recommended_unit`, `suggested_delivery_time`, `alerts`, and `notes`.
    - Validate that `recommended_volume` and `suggested_delivery_time` have correct data types (number and ISO timestamp/null).
4. **Resilience and Fallback Handling (Alerts/Notes):**
    - Validate that the API returns **alerts** for specific issues (e.g., stale telemetry data).
    - Validate that **explanatory notes** are provided when fallback logic is triggered (e.g., "Fallback heuristic triggered").
    - Validate the "no delivery possible" scenario: `recommended_volume` must be 0, `suggested_delivery_time` must be `null`, and appropriate alerts must be returned.

5. FEATURES NOT TO BE TESTED

Testing will not include the internal database structure, underlying algorithms, or other services within the fictional FuelSync system that interface with the Smart Fuel Allocation API.

6. TESTING PROCEDURES

6.1 TEST EXECUTION

6.1.1 Test Cases

For each requirement (Acceptance Criterion), the Python test script will execute a set of pre-defined test cases. Each test case will send a specific request payload and validate the received API response against the expected results.

6.2 PASS/FAIL CRITERIA

A test case is considered to **pass** if the observed results are equal to the expected results defined by the Acceptance Criteria. To pass the overall testing exercise, the Python script must demonstrate that the API meets the following criteria:
• Delivery volume and schedule recommendations reflect demand forecasts, tank levels, and constraints.
• The selected optimization mode ("Cost," "Availability," or "Balanced") is correctly applied.
• When "Cost" optimization is used, the recommendations minimize delivery costs.
• The `suggested_delivery_time` respects the `delivery_window`.
• The API returns appropriate alerts and explanatory notes when issues or fallback logic are triggered.
• All required fields are present in the API response.
• If delivery is impossible, `recommended_volume` is 0 and `suggested_delivery_time` is null, with appropriate alerts.

6.3 DEFECT MANAGEMENT

Any test case that results in observed results not matching the expected results will be considered a defect. Given the scope of this exercise, defects should be logged within the summary findings in the README.

7. RISKS AND CONTINGENCIES

Risk Description	Risk Level	Contingency Plan
Logic Complexity: The optimization logic is highly complex (analyzing demand, tank level, cost, and availability simultaneously).	Moderate to High	Thorough data generation must include edge cases and clear expected outputs for all three optimization modes.
Fallback Misbehavior: Fallback heuristic logic might provide recommendations that are factually incorrect or fail to log the required alerts/notes.	Moderate	Prioritize negative testing scenarios to confirm that alerts are triggered correctly and that volume/time are set to zero/null when required.
API Environment Instability: The assumed base URL/API is unavailable or unreliable during test execution.	Low (Assumption)	Note the inability to proceed with test execution in the README and document specific environment issues.