Software Testing Document

for

GCPS Bus Monitoring Through Kafka – Team 2

Sarah Fashinasi, Amali McHie, Tyler Hood, Alex Baker, Jeffrey Sanderson

Version 1.0

11 / 17 / 2024

Table of Contents

1. Introduction **3**

1.1. Test Objectives 3
1.2. Test Strategies 3
1.3. Scope of Tests 4
1.4. Testing Environment 4

2. Test Scenarios **4**

2.1. Client Scenarios 5
2.2. User Scenarios 5

3. Test Cases **5**

3.1. Data Producer & Consumer 5
3.2. Kafka Connector 6
3.3. User Interface 7
3.4. Database Querying 7

1. Introduction

This document outlines the testing framework and approach for the GCPS Bus Monitoring project, detailing the methodologies, scope, and test cases implemented during the software development lifecycle. The purpose of testing is to ensure the system meets client requirements, functions reliably across all environments, and delivers a high-quality user experience.

1.1 Test Objectives

One of the main objectives of testing is to ensure that all requirements requested by the client are covered in full and are implemented correctly. The requirements that are stated in the project proposal from the client should be fully represented in the team's final product, and as such should be feature rich and contain the solution the client asked for. Part of this objective is to also identify areas of the program that diverged from their intended requirement, allowing the team to correct the fault and cover all requirements correctly. Another objective for testing this product is to identify and cover edge cases that may result in unexpected behaviors that interfere with the functionality of the system. These logical errors within the system should be found throughout the testing process and fixed accordingly to reinforce the quality of the software.

The last main objective of testing is to locate and patch bugs relating to quality issues, such as slow performance and visual errors. Testing should reveal inadequacies in both the front-end user interface and the back-end data generation and database logic. Identifying issues in these areas allows for early debugging and an improvement in quality, making this aspect a primary purpose of testing the program.

1.2 Test Strategies

Several strategies will be employed to ensure comprehensive test coverage for this project. Unit testing, the most common form of testing, will verify the functionality of individual features as they are developed, focusing on user interfaces, data generation logic, data processing logic, and Kafka data streams. This approach helps identify and address errors in functions, data connections, or system interfaces early in development. Integration testing will be conducted to ensure seamless functionality across components developed by different team members, particularly at the front-end, back-end, and database junctions. This is especially critical as updates are pushed through GitHub to maintain system stability after integration.

White box testing will be a key strategy for analyzing and resolving errors in back-end components, including database queries, data generation, and data processing. Since back-end code tends to be error-prone, this approach allows the team to locate and fix logical issues efficiently, unlike black box testing, which is less practical in this context. Finally, compatibility testing will verify that the containerized application operates consistently across major operating systems, including Linux (Red Hat 9), Windows, and macOS, ensuring the solution meets platform portability requirements.

1.3 Scope of Tests

Testing will strictly cover the front-end, back-end, and database for this project, limited only to the source files found in the SWE-Capstone-GCPS repository. Code that is in the API's used in this project such as Kafka are not in the scope of testing, only the areas where the API's are used. As previously mentioned, testing includes the use of several operating systems to ensure portability and consistent functionality. The majority of testing will be performed in the areas where data is processed, data is transferred through the Kafka stream, data is queried into the database, and where the front-end interprets recent data.

1.4 Testing Environment

Testing will be conducted within the development environment using Visual Studio Code for development and debugging, GitHub for version control and collaboration, and Docker for containerization and cross-platform compatibility testing.

2. Test Scenarios

The following test scenarios represent general actions that the client and users of the system will perform.

2.1 Client Scenarios

- The client can query the database to find all information on the tracking events that are produced from the bus assets.
- The client can query the database to view all telemetry data collected from bus assets.
- The client can query the database to connect the telemetry data to the event that produced the data.
- The client can use the Docker container to install the system on any machine.

2.2 User Scenarios

- The user can view all buses currently being tracked.

3. Test Cases

3.1 Data Producer & Consumer

Test Case	Steps	Intended Result	Real Result	Status
Produce valid location and speed events.	Edit the producer so that it generates valid data for the location and speed events, then run main.py.	The events are processed and placed into the database	Accurate events and telemetry data are inserted into the database.	Pass
Produce valid location and speed events within a geofenced location.	Modify producer for geofenced data and run main.py.	The events are produced, then processed to add the named location to the set of data that is then inserted into the database.	The events and telemetry data enter the database correctly, including the geofence name.	Pass
Produce invalid location and	Edit the producer so that it generates data that is missing the	The events are produced but not inserted into the	Data does not appear in the database and an	Pass

speed events due to value that cannot be null.	Asset_ID value, then run main.py.	database, and they produce an error log.	error log was created.	
Produce invalid location and speed events	Edit the producer so that it generates data that has a 0 for the Is_Valid value, then run main.py.	The events are produced and enter the database but are marked as invalid.	Events appear in the database as expected.	Pass
Check latency of data insertion.	Produce valid data, then in the database check the difference between Happened_At_Time and Row_Modified_Time.	The difference between the 2 values is less than 1 second.	The average difference in time was 352 ms.	Pass

3.2 Kafka Connector

Test Case	**Steps**	**Intended Result**	**Real Result**	**Status**
Verify that producer sends data to Kafka stream.	Run producer.py and view the output.	The output of producer contains data that is sent to the Kafka stream.	The output data is sent to the Kafka stream.	Pass
Verify that consumer receives data from producer.	Run producer.py and consumer.py and view the outputs of both.	The consumer output contains the same data that the producer makes and receives that data through Kafka.	The outputs are the same and the data runs through Kafka.	Pass

Verify that the consumer subscribes to the Kafka stream.	Run main.py and view the output.	Main.py starts producer.py and consumer.py and the code affirms that the consumer gets the data through Kafka.	The consumer receives the data through Kafka and inserts it into the database.	Pass

3.3 User Interface

Test Case	Steps	Intended Result	Real Result	Status
Verify the accuracy of the data of each bus.	On the website, click on the bus and verify that the information associated matches the database.	The bus data should be up to date with the database every 5 seconds and should reflect the latest information each time.	The bus has up to date data that refreshes every 5 seconds.	Pass
Verify accuracy of route line.	On the website, click on the bus and verify that the line drawn for the bus matches its planned route.	The route on the map is accurate to the bus route stored.	The route is accurate.	Pass
Move around the map.	On the website, move the map around and zoom in and out of it.	The bus pin adjusts with the map and has consistent locations.	The pins move with the map as expected.	Pass

3.4 Database Querying

For the following test cases, the user is in the MySQL terminal and has already run the command USE GCPSBusMonitoring.

Test Case	**Steps**	**Intended Result**	**Real Result**	**Status**
Asset_Telemetry table can be viewed.	Select the Asset_Telemetry table using the SQL command.	The table of data is given, including the most recently inserted data.	Table displays containing all data, which also includes the data in the past second.	Pass
Event_Instances table can be viewed.	Select the Event_Instances table using the SQL command.	The table of data is given, including the most recently inserted data.	Table displays containing all data, which also includes the data in the past second.	Pass
Event_Instances and Asset_Telemetry can be queried together.	Select the Event_Instances and Asset_Telemetry tables using a join command at the Event_ID value.	A table displays that has all columns from both tables, and shows that each event is connected to a row of telemetry data.	The table displays containing all data.	Pass