


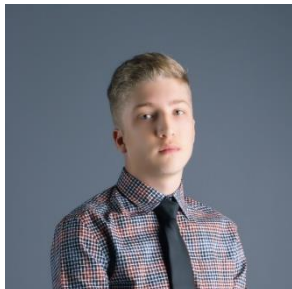



12: GCPS Live Bus Monitoring System

CS 4850 - Section 01 – Fall 2024

Aug 27, 2024

		
Sam Bostian (Team Leader)	Michael Rizig (Developer)	
		
Charlie McLarty (Developer)	Allen Roman (Developer)	Brian Pruitt (Documentation)

Team Members:

Name	Role	Cell Phone / Alt Email
Sam Bostian (Team Lead)	Developer	321.292.4693 sboastian@students.kennesaw.edu
Michael Rizig	Developer	678.668.3294 mrizig@students.kennesaw.edu
Charlie McLarty	Developer/QA	470.303.9544 cmclarty21@gmail.com
Brian Pruitt	Documentation	404.207.6548 bpruitt9@students.kennesaw.edu
Allen Roman	Developer	470.249.0421

		aroman14@students.kennesaw.edu
--	--	--------------------------------

Software Design Document

GCPS Live Bus Monitoring System

Version 0.1

August 31, 2024

Software Development Team:

Sam Bostian, Charlie McLarty, Brian Pruitt, Michael Rizig

Supervisors: Ed Van Ness and Sharon Perry

Gwinnett County Public Schools

Kennesaw State University

Revisions

Version	Primary Author	Description of Version	Changes	Date Completed
0.1	Sam Bostian, Charlie McLarty, Brian Pruitt, Michael Rizig	Initial Release		

Table of Contents

Revisions	2
1. Introduction and Overview	4
1.1 Purpose	4
1.2 Introduction	4
1.3 System Overview.....	4
1.4 Definitions, Acronyms and Abbreviations	5
2. Design Considerations	5
2.1 Assumptions	5
2.2 Dependencies.....	5
2.2.1 Kafka [1]	5
2.2.2 Microsoft SQL Server	6
2.2.3 Podman Container	6
2.1. General Constraints	6
2.2. Development Methods.....	7
3. Architectural Strategies	7
4. System Architecture	8
4.1 High-Level Architecture Overview	8
4.2 Partitioning of Responsibilities	9
4.3 Interaction Between Components.....	9
5. Detailed System Design	10
5.1. Classification	10
5.2. Definition.....	10
5.3. Constraints.....	10
5.4. Resources.....	10
5.5. Interface/Exports	10
6. Bibliography	10

1. Introduction and Overview

1.1 Purpose

This document describes the design and implementation of the software solution for Gwinnett County Public School's issue with using polling to gather bus data. A detailed description for how to set up an Apache Kafka consumer, the python solution used to handle and process data, the relationship database management, containerization of the solution and a system for monitoring and handling errors.

1.2 Introduction

This project aims to enhance the efficiency and safety of school bus operations for Gwinnett County Public Schools (GCPS) by transitioning from the current Samsara REST API polling method to a Kafka-based event streaming solution. Currently, GCPS uses Samsara REST APIs to poll data on school buses, including locations, speeds, and vehicle health, every 5 seconds. However, this method introduces latency, potential bottlenecks and a large consumption of API calls. Due to Gwinnett Counties' large fleet, of approximately 2000 buses, they are currently in the top 1% of Samsara's REST API usage. The current polling system contributes to a 15 second delay from when the event takes place.

1.3 System Overview

To address these issues, this project involves developing a stand-alone application on a Linux, using Red Hat Enterprise Linux distribution, a subscriber will consume events made from the Samsara Kafka Connector. The application will process real time events, perform data validation, and sort valid and invalid records into their own separate relationship SQL server database, where they can be accessed by various applications. Additionally, the solution will be containerized using Podman for consistent deployment, with optional monitoring functionality to track performance and manage backlogs. To test the applications' capabilities simulated real-time data will be streamed to simulate the load of GCPS.

1.4 Definitions, Acronyms and Abbreviations

Acronym	Meaning
API	Application Programming Interface
GCPS	Gwinnett County Public Schools
JSON	JavaScript Object Notation
KRaft	Apache Kafka Raft
Podman	Pod Manager
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux
SQL	Structured Query Language
WSL	Windows Subsystem for Linux

2. Design Considerations

2.1 Assumptions

	Date Identified	Assumption	Validation By:	Date Completed	Valid	Comments
1	9/1/31	2000 Buses Streamed	Sam Bostian	-	<input type="checkbox"/>	-

2.2 Dependencies

2.2.1 Kafka ^[1]

- Kafka: 0.10.1.0 or later release
- Kafka Cluster:

Serverless-like Kafka	Kafka as a Service	Self-managed Kafka	Local Kafka
Upstash Apache Kafka	Confluent Cloud	Confluent Platform	Test Containers
Amazon MSK Serverless	Red Hat OpenShift Streams	Red Hat AMQ Streams (Strimzi)	Quarkus Dev Services (Red Panda)
	Heroku Apache Kafka	Cloudera Data Platform	EmbeddedKafkaCluster

[2]

- Operating System: WSL2 with a Linux OS or Any Linux OS

- Java: Java 8 or 11 , Java Zookeeper or Kafka Raft

2.2.2 Microsoft SQL Server

- Operating System: Windows 10 TH1 1507 or Windows Server 2016 or greater
- .NET Framework:

2.2.3 Podman Container

- Operating System: Windows, MacOS, or Linux
- Minimum Requirements:
2 physical CPU cores
2 GB of free memory
35 GB of storage space

Describe any assumptions or dependencies regarding the software and its use. These may concern such issues as:

- Related software or hardware
- Operating systems
- End-user characteristics
- Possible and/or probable changes in functionality

2.1. General Constraints

Describe any global limitations or constraints that have a significant impact on the design of the system's software (and describe the associated impact). Such constraints may be imposed by any of the following (the list is not exhaustive):

- Hardware or software environment
- End-user environment
- Availability or volatility of resources
- Standards compliance
- Interoperability requirements
- Interface/protocol requirements
- Data repository and distribution requirements
- Security requirements (or other such regulations)
- Memory and other capacity limitations
- Performance requirements
- Network communications
- Verification and validation requirements (testing)
- Other means of addressing quality goals
- Other requirements described in the requirements specification

2.2. Development Methods

Briefly describe the method or approach used for this software design. If one or more formal/published methods were adopted or adapted, then include a reference to a more detailed description of these methods. If several methods were seriously considered, then each such method should be mentioned, along with a brief explanation of why all or part of it was used or not used.

3. Architectural Strategies

This section outlines the key design decisions and strategies that influence the overall architecture of the system. Each decision is made with careful consideration of the project's goals, constraints, and future extensibility.

Programming Language:

- **Python/C#:** The system will use Python or C# for consuming data from Kafka and processing it. These languages were chosen because of their robust support for handling JSON data, extensive libraries for data manipulation, and ease of integration with both Kafka and SQL Server.
- **SQL Server:** Chosen for its compatibility with the existing infrastructure at GCPS and its strong support for handling large volumes of relational data. SQL Server's advanced features such as indexing, stored procedures, and analytics capabilities are crucial for this project.
- **Kafka:** Kafka is used for its ability to handle large streams of data in real-time with low latency. Kafka's distributed architecture provides the scalability and fault tolerance needed for this project.

Reasoning: The choice of Python/C# and SQL Server balances the need for ease of development with the requirements for processing and storing large volumes of data. Kafka's event streaming capabilities align with the project's goal of reducing latency in telemetry data processing.

Alternatives Considered: Alternatives like using MySQL or PostgreSQL instead of SQL Server were considered but rejected due to the existing reliance on Microsoft technologies at GCPS.

Concurrency and Synchronization

Multi-threading: The system will use multi-threading to process multiple Kafka streams concurrently, ensuring that data is processed in near real-time. GCPS is the largest school system in Georgia with nearly 2000 busses being monitored concurrently.

Reasoning: Multi-threading allows the system to handle high volumes of data without bottlenecks.

4. System Architecture

This section provides a high-level overview of the system architecture, explaining how the system's functionality is partitioned and how responsibilities are assigned to different subsystems or components. The system is designed to handle the ingestion, processing, and storage of telemetry data in a scalable and maintainable way.

4.1 High-Level Architecture Overview

The system architecture is designed around the following major components:

1. Kafka Event Streaming Subsystem

- **Purpose:** This subsystem is responsible for ingesting telemetry data from Samsara via the Kafka event streaming platform.
- **Components:**
 - **Kafka Producer:** Produces telemetry events from the Samsara system and publishes them to a Kafka topic.
 - **Kafka Broker:** Manages the Kafka topic and ensures that data is distributed to consumers.
 - **Kafka Consumer:** Consumes the telemetry events from the Kafka topic for further processing.

2. Data Processing Subsystem

- **Purpose:** This subsystem processes the raw telemetry data received from Kafka, including validation and transformation of the data.
- **Components:**
 - **Validation Module:** Validates incoming data based on predefined rules (e.g., ensuring GPS coordinates are within a valid range).
 - **Transformation Module:** Transforms the data into a format suitable for storage in the SQL Server database.
 - **Error Handling Module:** Handles any errors encountered during data processing, including logging and retry mechanisms.

3. Data Storage Subsystem

- **Purpose:** This subsystem is responsible for storing the processed telemetry data in a relational database.
- **Components:**
 - **SQL Server Database:** The primary data storage system, where validated and transformed telemetry data is stored in a normalized format.
 - **Data Access Layer (DAL):** Provides an interface for storing and retrieving data from the SQL Server database.

4. Deployment and Containerization Subsystem

- **Purpose:** This subsystem ensures that the application can be consistently deployed across various environments using containerization technology.
- **Components:**
 - **Docker/Podman Container:** Encapsulates the entire application, including all dependencies, to ensure consistent behavior across environments.

- **CI/CD Pipeline:** Manages the automated deployment of the containerized application, ensuring that updates are deployed smoothly and reliably.

5. Monitoring and Logging Subsystem

- **Purpose:** This subsystem tracks the performance and health of the system, providing insights into system operations and alerting administrators to potential issues.
- **Components:**
 - **Logging Service:** Captures logs from various components for auditing and troubleshooting purposes.
 - **Monitoring Dashboard (Optional):** Provides real-time visualization of system metrics, such as event processing rates and error rates.

4.2 Partitioning of Responsibilities

The architecture divides responsibilities among these subsystems to achieve modularity, scalability, and maintainability:

- **Kafka Event Streaming Subsystem:** This subsystem handles the ingestion of telemetry data, which is crucial for ensuring that data is captured in near real-time with minimal latency.
- **Data Processing Subsystem:** By isolating data validation and transformation into its own subsystem, the architecture ensures that these operations can be easily updated or replaced without affecting other parts of the system.
- **Data Storage Subsystem:** The separation of data storage into its own subsystem allows for flexibility in database management and scalability, ensuring that the system can handle increasing volumes of data.
- **Deployment and Containerization Subsystem:** This subsystem ensures that the application can be deployed reliably and consistently across different environments, which is essential for maintaining system integrity.
- **Monitoring and Logging Subsystem:** This subsystem provides ongoing visibility into system operations, allowing administrators to quickly identify and address any issues that arise.

4.3 Interaction Between Components

- **Data Flow:** Telemetry data flows from the Kafka Event Streaming Subsystem to the Data Processing Subsystem, where it is validated and transformed. Valid data is then passed to the Data Storage Subsystem for storage in the SQL Server database.
- **Error Handling:** Errors encountered during data processing are logged by the Error Handling Module, and depending on the nature of the error, the data may be retried or logged as invalid.
- **Deployment:** The entire system is encapsulated within a Docker/Podman container, which is managed by the CI/CD pipeline for consistent deployment across different environments.
- **Monitoring:** The Logging Service captures logs from all subsystems, which are then visualized in the Monitoring Dashboard to provide real-time insights into system performance.

5. Detailed System Design

Most components described in the System Architecture section will require a more detailed discussion. Other lower-level components and subcomponents may need to be described as well. Each subsection of this section will refer to or contain a detailed description of a system software component. The discussion provided should cover the following software component attributes:

5.1. Classification

The kind of component, such as a subsystem, module, class, package, function, file, etc.

5.2. Definition

The specific purpose and semantic meaning of the component. This may need to refer to the requirements specification.

5.3. Constraints

Any relevant assumptions, limitations, or constraints for this component. This should include constraints on timing, storage, or component state, and might include rules for interacting with this component (encompassing preconditions, postconditions, invariants, other constraints on input or output values and local or global values, data formats and data access, synchronization, exceptions, etc.)

5.4. Resources

A description of any and all resources that are managed, affected, or needed by this entity. Resources are entities external to the design such as memory, processors, printers, databases, or a software library. This should include a discussion of any possible race conditions and/or deadlock situations, and how they might be resolved.

5.5. Interface/Exports

The set of services (resources, data, types, constants, subroutines, and exceptions) that are provided by this component. The precise definition or declaration of each such element should be present, along with comments or annotations describing the meanings of values, parameters, etc. For each service element described, include (or provide a reference) in its discussion a description of its important software component attributes (Classification, Definition, Responsibilities, Constraints, Composition, Uses, Resources, Processing, and Interface).

6. Bibliography

- [1] <https://kafka.apache.org/documentation/#design>
- [2] <https://developers.samsara.com/docs/data-connectors>