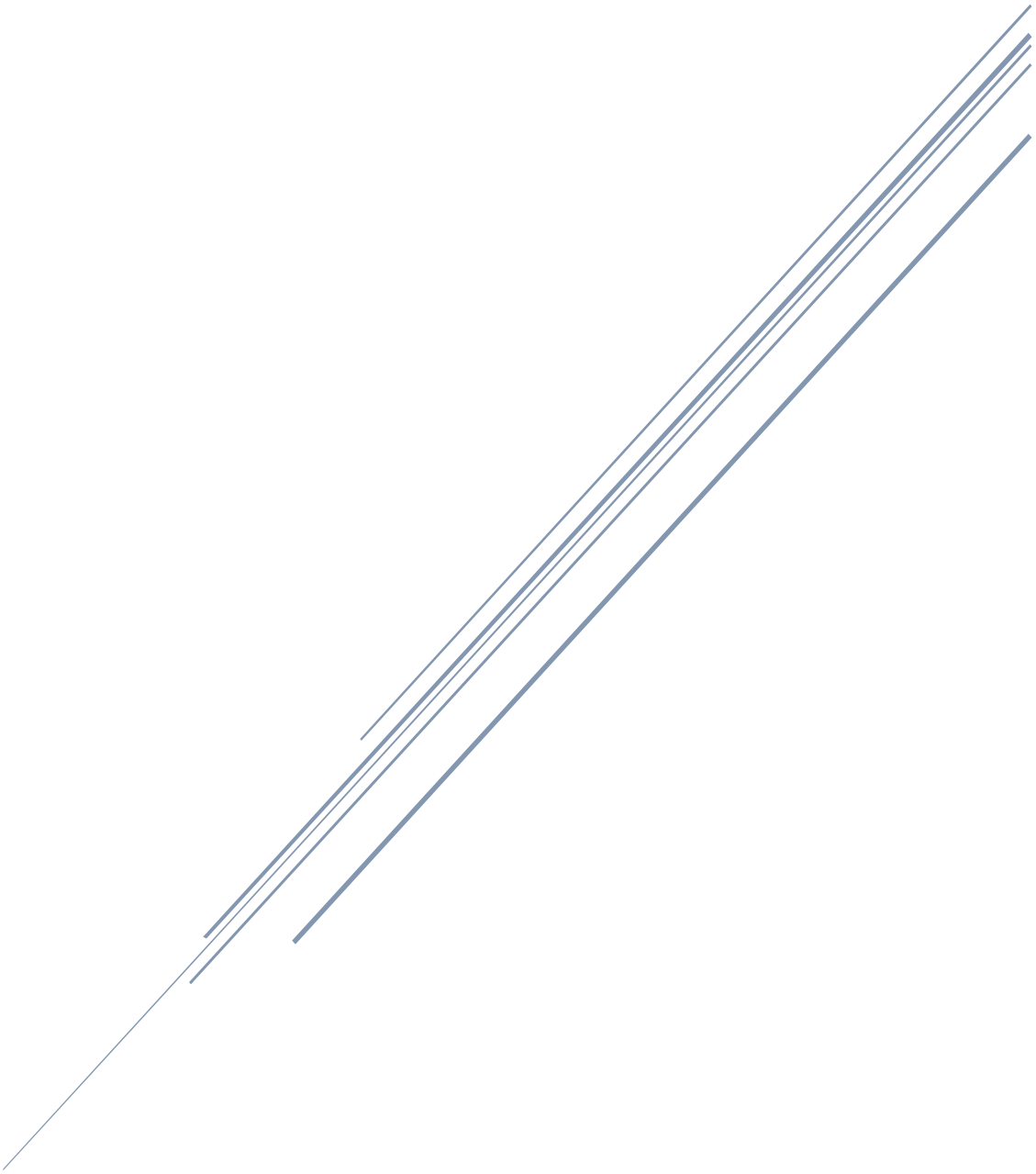
SENSOR DATA INTERPRETER

SYSTEM DESIGN DOCUMENT



Content

Version History Page

Version	Author	Reason	Date
1.0	Mert Karaçam	Initial Draft	26.11.2025

1. Introduction

The Sensor Data Interpreter is a small backend service developed to process real-time sensor data coming from different vehicle types. The company already streams all raw sensor messages into a central message queue, but the messages are not being interpreted or stored in a meaningful way. This application fills that gap by providing a clean and reliable pipeline: reading each message, understanding what type of data it contains, applying the necessary rules, and saving the output into the system database.

Every incoming message is treated as if it was pulled from the central queue, but for testing and demo purposes a REST endpoint (/api/messages) can also be used to ingest messages manually. From there, the service categorizes the message as either **statistical** or **operational**. Statistical messages are stored directly. Operational messages require additional work: the service loads rule definitions from the configuration table, performs calculations such as battery-charge checks or voltage-change thresholds, and generates alarms when the rules are satisfied. All results are written to the database in a consistent way.

The goal of this document is to give a clear, straightforward explanation of the system for developers who have never seen the project before. It covers the architecture, processing flow, database structure, API behavior, and the reasoning behind some of the design choices. The focus is not only on making the code understandable, but also on explaining the thinking behind it so that future developers can extend or modify the system with confidence.

2. Project Plan

2.1 Clarify Requirements

Before designing or implementing the Sensor Data Interpreter, the first step was to clearly understand what the system is expected to do and under which constraints it must operate. The requirements were grouped into two categories: **functional** and **non-functional**.

2.1.1 Functional Requirements

1. Message Ingestion

- The system must consume sensor messages from a central message queue.
- For testing/demo purposes, messages can also be sent manually via the REST endpoint: **POST /api/messages**

2. Message Classification

- Every incoming message must be identified as either **statistical** or **operational** based on whether it contains status_changes.

3. Statistical Data Processing

- Pure statistical messages should be validated and stored directly.
- They must remain independent of operational rules or external lookups.

4. Operational Data Processing

- Operational messages require an additional enrichment step:
    - Load rule thresholds from configuration table.
    - Perform calculations (battery charge validation, voltage delta checks).
    - Generate alarms when a rule is violated.
    - Persist the interpreted operational record.

5. Data Persistence

- The system must save:
    - Raw sensor-related fields,
    - Computed metrics,
    - Alarm flags,
    - Location information,
    - Timestamps.

6. Expose Query APIs

The system must provide endpoints for data retrieval:

- **Location history**
  GET /api/location-history - returns device location history within a time range.
- **Operational data**
  GET /api/operational-data - query operational records by device ID or time interval.
- **Statistical data**
  GET /api/statistical-data - query statistical records by type or time interval.

2.1.2 Non-Functional Requirements

1. Real-Time or Near Real-Time Performance

- The processing pipeline must remain stable even when message volume increases (100 -> 5000 msg/s).

2. Zero Message Loss

- The queue does not support re-reading.
- Every message must be processed exactly once, without dropping or duplicating.

3. Scalability

- The architecture should allow scaling the listener horizontally if needed.

4. Simplicity & Extensibility

- New rules, new sensor types, and new message fields must be easy to add.
- Layers (controller -> service -> repository) must be cleanly separated.

5. Testability

- Core modules (router, services, calculator, Controllers, ExceptionHandler) must be unit-testable.

6. Error Handling

- The system must return meaningful errors.
- All exceptions should be captured by a global exception handler.

7. Observability

- Logs must allow developers to trace message flow and identify failures quickly.

2.2 Capacity & Load Estimation

Since the Sensor Data Interpreter is designed to process continuous, high-volume sensor streams, it is important to estimate how much data the system may need to handle. These values are **approximations**, not strict limits, but they help justify certain design decisions such as the need for lightweight message routing, fast I/O operations, and efficient persistence.

1. Expected Message Throughput

According to the problem description, the size of the message stream can vary significantly over time:

- **Normal load:** ~100 messages/second
- **Peak load:** ~5,000 messages/second

For planning purposes, the system is designed assuming:

- Peak load may last for several minutes at a time
- Message volume can fluctuate abruptly
- The system must not drop or skip messages

2. Message Size Estimate

A typical sensor message contains:

- Device identifiers
- Environmental metrics (temperature, pressure, humidity…)
- Status changes (only in operational messages)
- Location (geometry object)
- Timestamps

Average approximate size: **1–2 KB per message**

3. Daily Data Volume

If peak or near-peak load sustains, approximate daily traffic becomes:

- **Normal day:**
  ~8.6 million messages
  ~9–17 GB raw JSON data

- **Peak traffic sustained:**
  ~430 million messages
  ~430–860 GB raw JSON data

4. Read/Write Ratio

The system is **write-intensive**:

- **Writes:**
  Every message ingested results in a DB write -extremely high write frequency
- **Reads:**
  API calls such as
  /api/location-history,
  /api/operational-data,
  /api/statistical-data
  are occasional and mostly used for analytics, debugging or reporting.

Therefore:

- **Write ratio:** ~95%
- **Read ratio:** ~5%

This is why the design favors:

- Simple repository operations
- Proper indexing
- Lightweight DTO mappings

5. Network Bandwidth

Given peak throughput:

- **5,000 msg/s × 2 KB ≈ 10 MB/s**
- Hourly peak bandwidth ≈ 36 GB/hour
- Not all of this is stored; operational data is smaller after processing.

6. Database Storage Estimation

The persisted data is significantly smaller than the incoming raw JSON:

- Only interpreted and calculated fields are stored
- Unneeded raw sensor data is not duplicated

Rough approximation:

- Each DB row ~300–600 bytes (after trimming raw JSON)
- With 100 million records - ~30–60 GB total
- Acceptable for a standard relational database with proper partitioning/indexing (if needed in future versions)

For this assignment, storage remains small and manageable.

2.3 API Design

The API layer of the Sensor Data Interpreter provides a simple and practical interface for interacting with the system. All endpoints are designed to support either message ingestion or data querying. Since this project simulates a real message queue, the ingestion endpoint also acts as the main entry point of the pipeline.

API design intentionally follows a clean, predictable pattern:

- Base path starts with /api
- Controllers are grouped by responsibility
- Query parameters follow ISO date formats
- Responses use lightweight DTOs
- Swagger documentation is available for exploration and testing

Below is an overview of the API surface.

2.3.1 Message Ingestion API

POST /api/messages

Purpose:
Simulates queue ingestion by accepting a raw sensor message and forwarding it to the processing pipeline.

Description:

- Accepts a JSON payload representing a sensor message
- Validates required fields (e.g., type)
- Routes the message to statistical or operational handlers
- Returns a simple acknowledgment string after processing

Request Body:
SensorMessageDTO (type, humidity, pressure, geometry, statusChanges, etc.)

Response:

- 200 – Message processed and persisted
- 400 – Invalid request body or missing fields
- 500 – Unexpected server error

2.3.2 Location History API

GET /api/location-history

Purpose:
Returns all event locations of a given device within a specified time range.

Description:
This endpoint satisfies the requirement in the original assignment to deliver location history for the selected device.

Parameters:

- deviceId (required)
- startTime (ISO datetime, required)
- endTime (ISO datetime, required)

Response:
List of EventLocationDTO → latitude, longitude, timestamp.

2.3.3 Operational Data API

GET /api/operational-data

Purpose:
Fetch operational records that were enriched and validated by the rule engine.

Description:
Used for debugging, validation, or reviewing generated alarms.

Parameters:

- deviceId (optional)
- startTime (optional, ISO datetime)
- endTime (optional, ISO datetime)

Response:
List of OperationalData entities (battery charge, voltage values, rule evaluations, alarm flags).

2.3.4 Statistical Data API

GET /api/statistical-data

Purpose:
Returns stored statistical sensor data.

Description:
Provides filtered statistical data for analytics and verification.

Parameters:

- type (sensor type, optional)
- startTime (optional, ISO datetime)
- endTime (optional, ISO datetime)

Response:
List of StatisticalData entities.

2.4 Database Design

The Sensor Data Interpreter stores its processed messages using a relational data model designed to support high-frequency writes, predictable query patterns, and strict structural clarity. Sensor messages differ significantly depending on the presence of operational fields, so the schema separates statistical records, operational records, and rule configuration data into **three independent, purpose-specific tables**.

Each table represents a self-contained domain concept. This separation improves maintainability, reduces processing complexity, and avoids unnecessary coupling between different message types.

No foreign key constraints are defined between these tables. Instead, the system uses a **snapshot-based persistence model** in which each message is fully evaluated and stored with all necessary information at the time of processing. This aligns with the nature of sensor streams, where each record must stand alone and historical states must remain immutable.

2.4.1 Data Model

1. operational_data

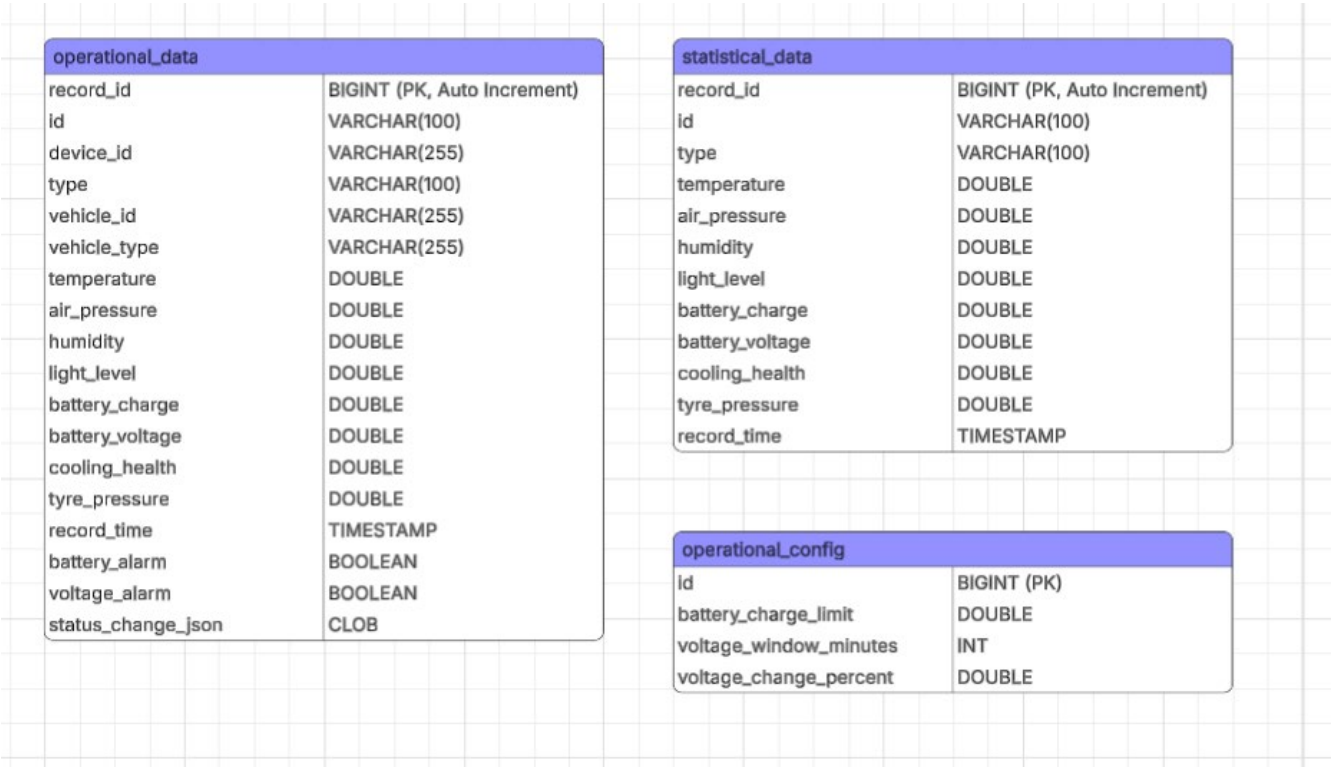
Holds enriched operational messages that contain status changes, additional metadata, voltage and battery readings, and alarm flags.
This table is also the single source of truth for location history.

2. statistical_data

Stores raw statistical sensor readings that do not include statusChanges.
These records represent simple measurements and are not linked to operational logic.

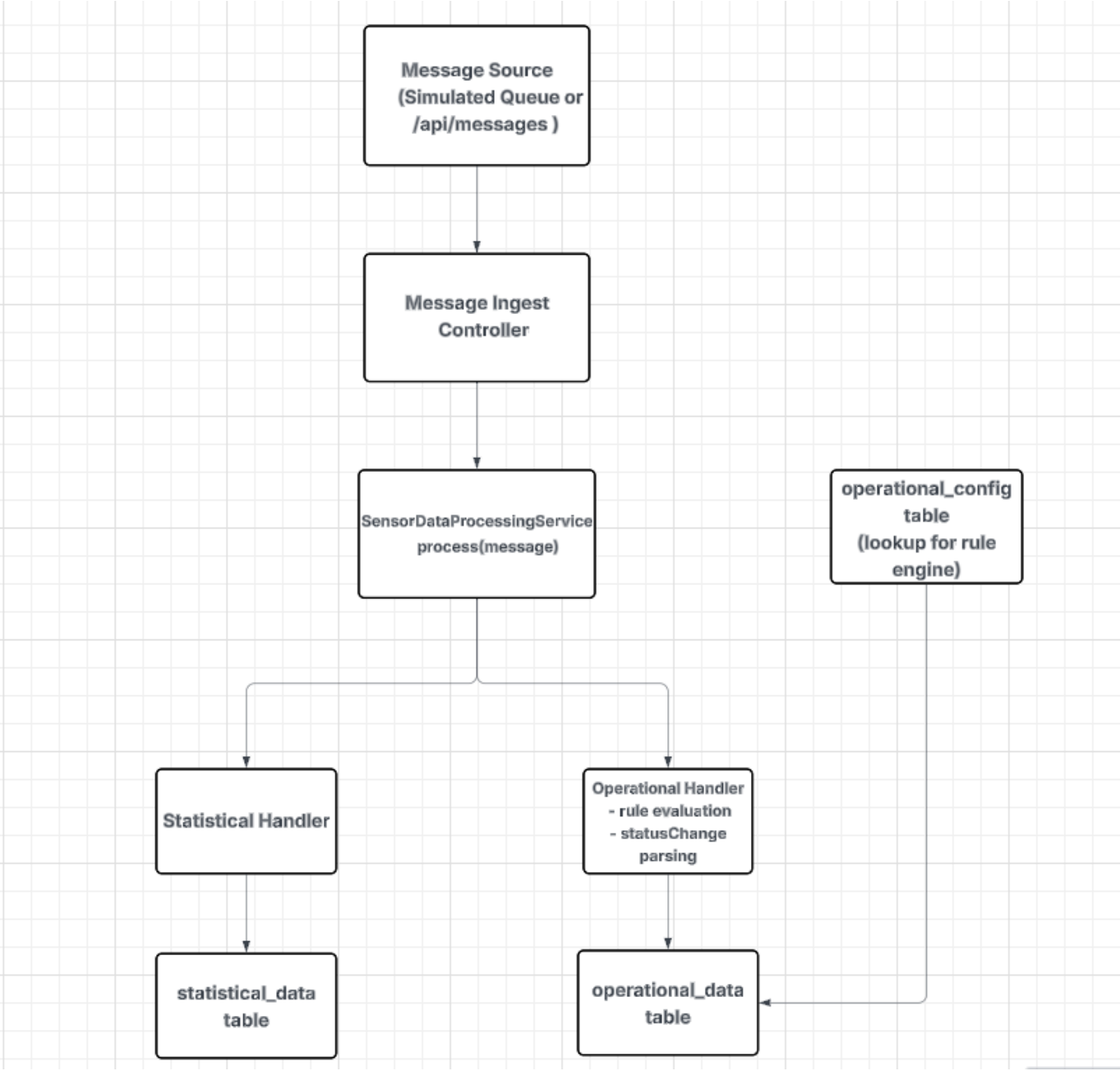
3. operational_config

A lookup table containing threshold values and tuning parameters used by the rule evaluation engine.
These configurations are read dynamically at runtime and are not tied to any individual record.



2.5 High-Level Architecture Design

The Sensor Data Interpreter follows a modular and layered architecture. Each component has a clear responsibility: receiving messages, classifying them, applying rules, and persisting the interpreted data. Query operations are handled through lightweight REST controllers that expose processed data back to the client.



2.6 Dive Into Key Components

1. Message Ingest Controller

Receives incoming messages via /api/messages and forwards them to the processing service.
Contains no business logic.

2. Message Router

Determines the message type by checking the presence of statusChanges and routes it to the appropriate handler.

3. Service-Level Dispatcher (processMessage)

Inside SensorDataProcessingServiceImpl, the processMessage() method acts as the central dispatcher.

It inspects the message structure and directs it to either handleOperational() or handleStatistical().

This keeps the controller lightweight and centralizes decision-making in the service layer.

4. Statistical Handler

Validates statistical messages, maps them to the entity, and saves them into the statistical_data table.

5. Operational Handler

Parses operational messages, loads configuration values, evaluates alarms, and persists the result in the operational_data table.

6. Metric Calculator

Performs voltage delta and other operational calculations needed by the rule engine.

7. Config Access

Reads rule parameters from the operational_config table and provides them to the operational handler.

8. REST Query Controllers

Expose stored data through:

- /api/location-history
- /api/statistical-data
- /api/operational-data

2.7 Scalability, Fault Tolerance & Reliability

1. Scalability

- The system can scale horizontally by running multiple instances of the message-processing service.
- Since each message is independent, instances can process messages in parallel without coordination.
- Database tables support high write volume through simple inserts and indexed filters.

2. Fault Tolerance

- Message ingestion is stateless, so failures do not impact previously processed messages.
- Operational rule evaluation uses static config values, keeping processing predictable even if the config table is temporarily inaccessible.
- Exceptions are centralized through a global exception handler to prevent partial or inconsistent writes.

3. Reliability

- Each message is processed as a complete snapshot, ensuring consistent historical records.
- No cross-table dependencies reduce the risk of relational failures.
- Validation and alarm logic happen before persistence, guaranteeing that stored data is complete and final.

4. Logging & Observability

- Key processing steps are logged (ingestion, routing, rule evaluation, alarms).
- Errors are captured with meaningful messages, making debugging straightforward.

3. Developer Guide

3.1 Project Structure

1. controller/

- Contains all REST API endpoints
- Handles incoming HTTP requests
- Forwards messages to the processing service
- /api/messages forwards incoming sensor messages to SensorDataProcessingServiceImpl for routing and processing.
- /api/location-history uses LocationHistoryServiceImpl to retrieve operational location data.
- /api/operational-data and /api/statistical-data directly query the corresponding repositories. These two endpoints are included for **demo and validation purposes**, and no additional business logic is required, so introducing a service layer would not provide functional value.

2. service/

- Contains the central message processing entry point
- SensorDataProcessingServiceImpl decides whether a message is statistical or operational
- Delegates processing to the appropriate handler

3. service/impl/

- Includes the actual handlers:
    - **StatisticalHandler** – processes statistical messages(handleStatistical)
    - **OperationalHandler** – processes operational messages, applies rules, calculates alarms(handleOperational)

4. calculator/

- Contains **MetricCalculator**, which performs operational calculations (e.g., voltage delta, threshold checks)

5. mapper/

- Responsible for converting incoming DTOs into JPA entities
- Ensures the persistence layer remains independent from request models

6. repository/

- Spring Data JPA repositories for:
    - operational_data
    - statistical_data
- Provides CRUD operations for all tables

7. model/

- JPA entities mapped to database tables:
  - OperationalData
  - StatisticalData
- Reflects the database schema used by H2

8. dto/

- Data transfer objects for:
  - incoming sensor messages
  - responses returned by API endpoints

9. config/

- Application configuration components:
  - H2 database setup
  - Config rule loading (from operational_config)
  - CORS configuration
  - Swagger/OpenAPI configuration for cloud execution,
  - OperationalConfigEntity / OperationalConfigRepository(kept here because operational rules are configuration parameters, not part of the main sensor-data domain)

10. exception/

- Global exception handler for consistent API error responses
- Captures validation errors, parsing issues, and runtime exceptions

3.1 Requirements

- Java **17+**
- Maven **3.8+**
- Internet (to download dependencies on first build)
- No external DB needed -> **H2 In-Memory** is auto-started
- IDE(Eclipse, IntelliJ, VS Code)

3.2 How to Run the Application

3.2.1 Navigate to the project directory

Open a terminal and move to the root folder of the project (the folder containing pom.xml).

1. Build the Project

```
mvn clean install
```

This will:

- download dependencies
- compile the code
- run unit tests
- prepare the application for execution

3. Start the application

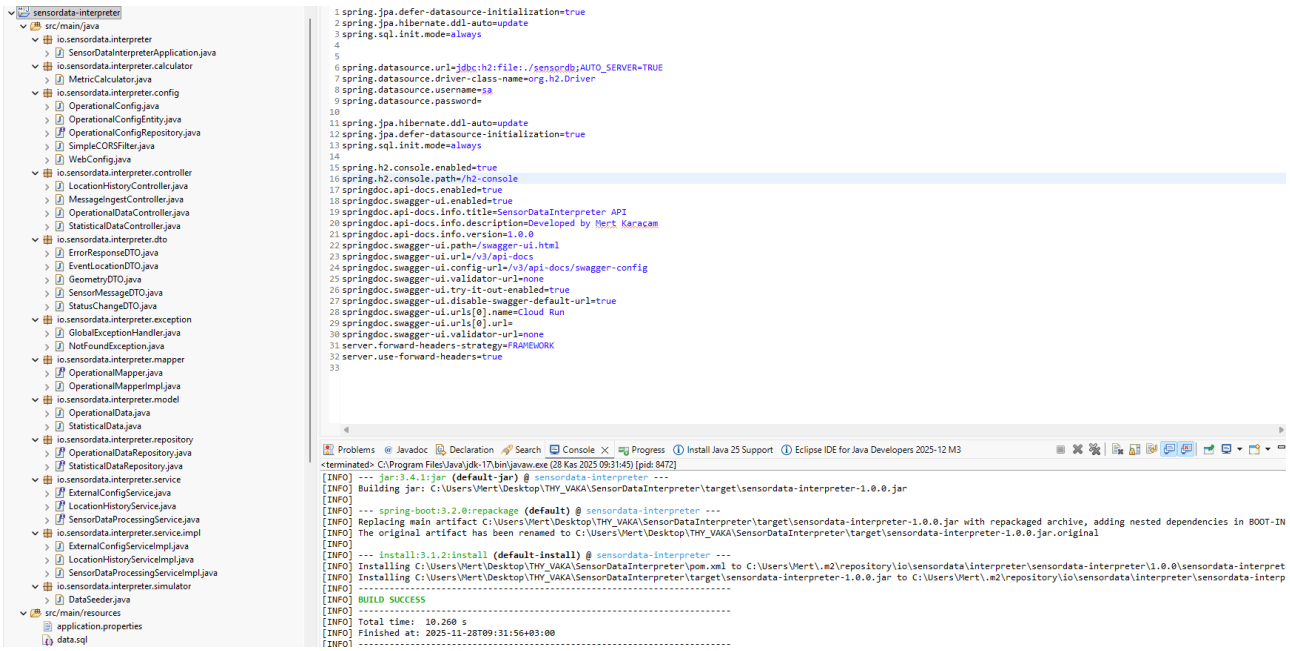
```
mvn spring-boot:run
```

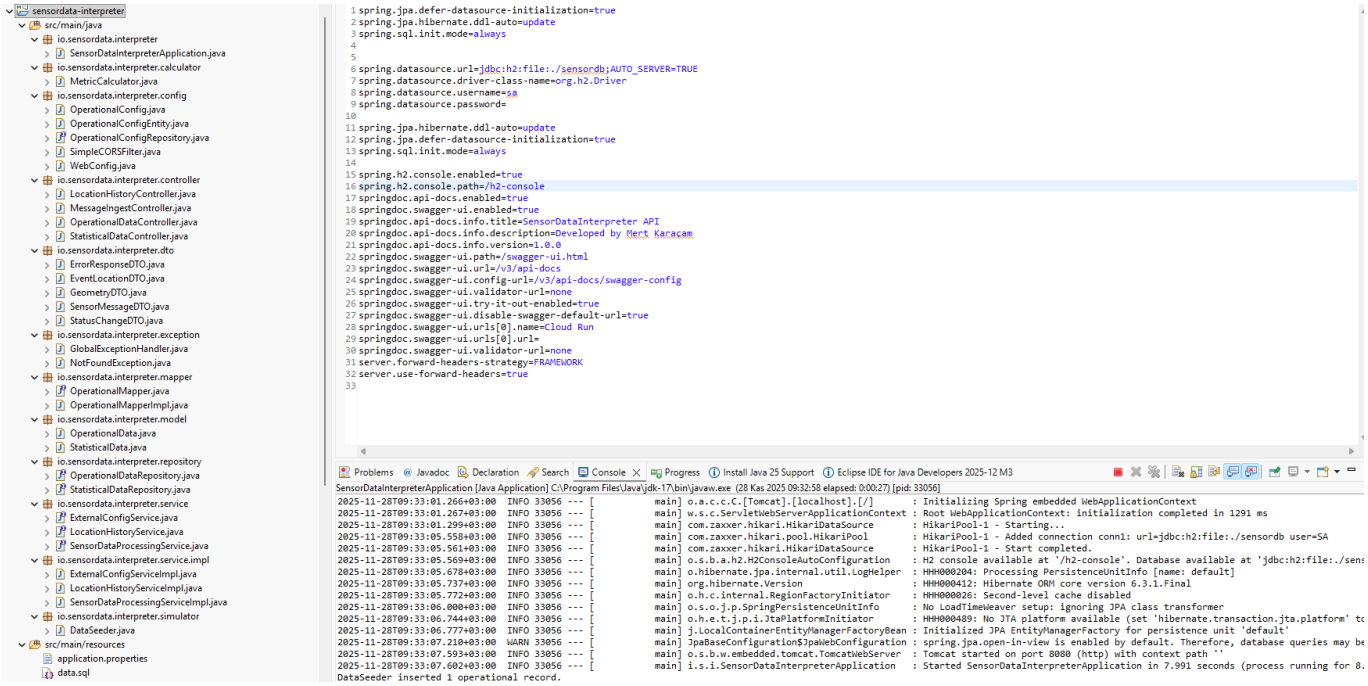
The application will start on: http://localhost:8080

4. Open Swagger UI
   http://localhost:8080/swagger-ui/index.html

Or You can run the project directly from your IDE without using the terminal.

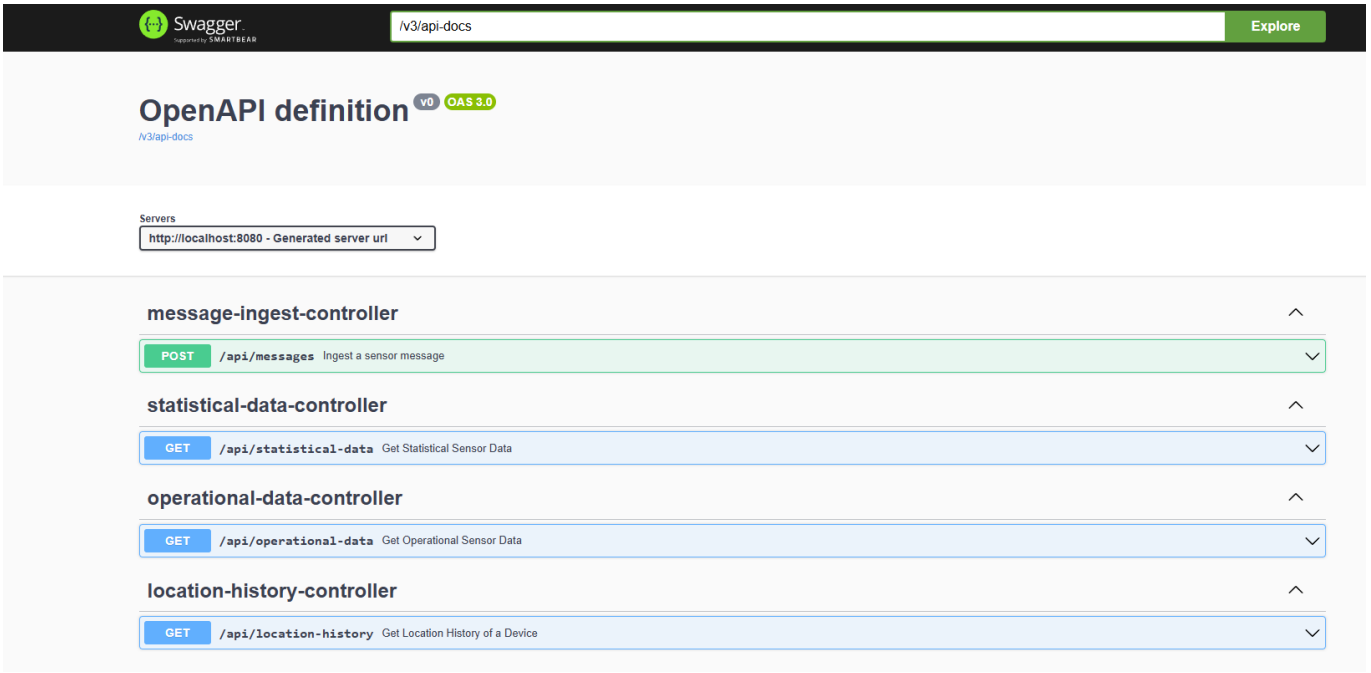
- Open file in IDE. SensorDataInterpreter
- Right-click file ->**Run As**-> **Maven Install**
- Right-click file -> **Run As-> Java Application(SensorDataInterpreterApplication.java)**
- The application will start on: http://localhost:8080
  Open Swagger UI http://localhost:8080/swagger-ui/index.html





From Swagger UI, you can:

- send sample messages using **POST /api/messages**
- inspect processed operational and statistical data
- retrieve location history



1. POST /api/messages

The /api/messages endpoint accepts both *operational* and *statistical* messages.
Below are two ready-to-test samples you can paste directly into Swagger or Postma

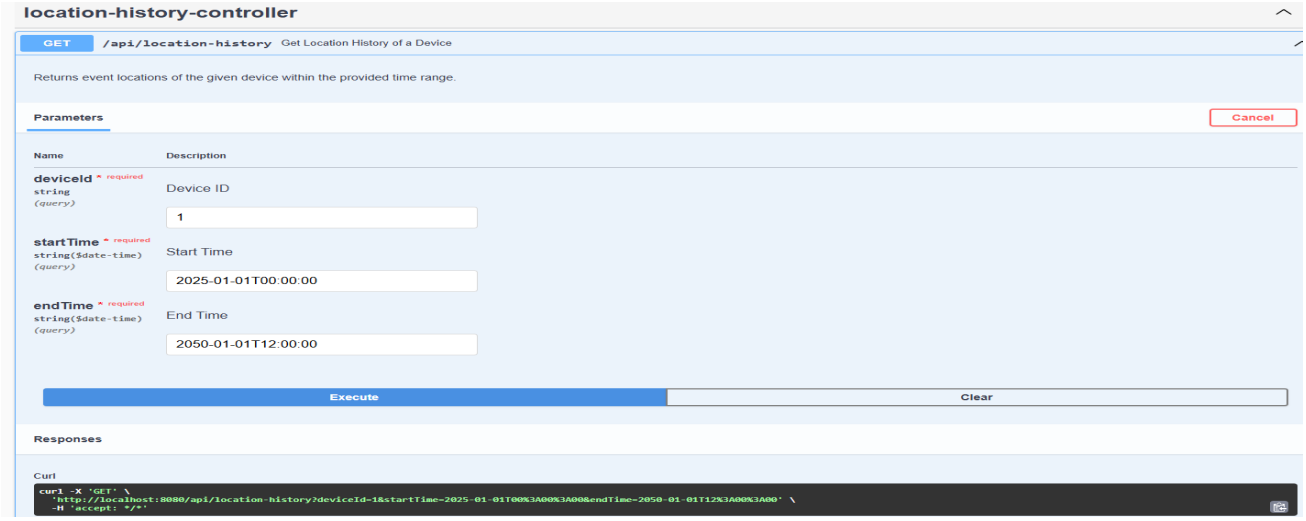
Operational Message Example:

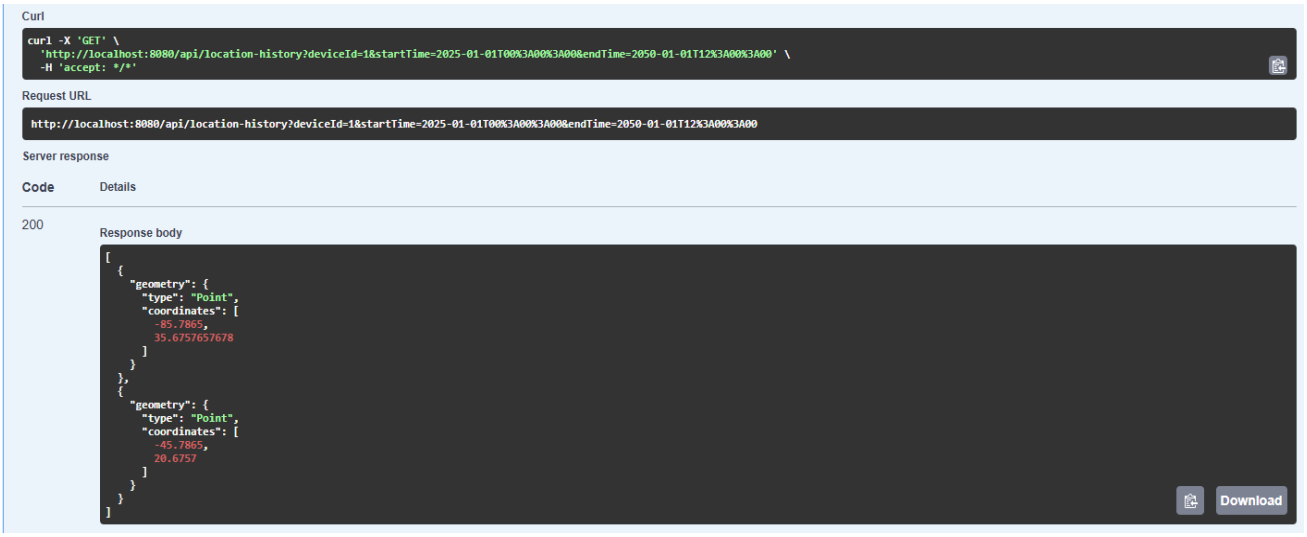

```
{
  "id": "01",
  "type": "DEV1",
  "temperature": 26.09,
  "airPressure": 101573,
  "humidity": 12.09,
  "lightLevel": 45145,
  "batteryCharge": 12.09,
  "batteryVoltage": 58,
  "coolingHealth": 78.0,
  "tyrePressure": 2.1,
  "statusChanges": [
    {
      "deviceId": "1",
      "vehicleId": "56790077",
      "vehicleType": "TPS678",
      "propulsionType": ["electric"],
      "eventType": "available",
      "eventTypeReason": "user_drop_off",
      "eventTime": 15472345678,
      "eventLocation": {
        "geometry": {
          "type": "Point",
          "coordinates": [-85.7865, 35.6757657678]
        }
      }
    }
  ]
}
```

Statistical Message Example

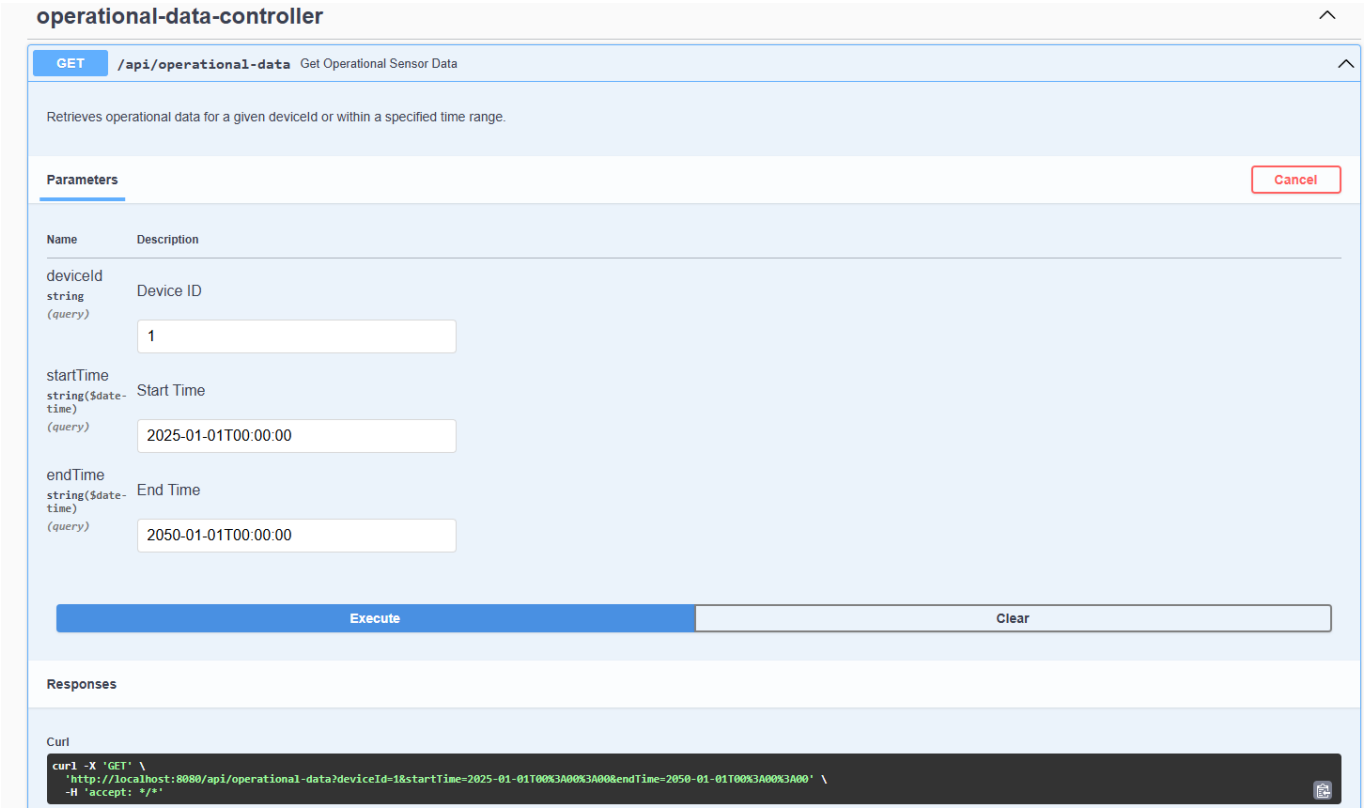

```
{
 "id": "02",
 "type": "DEV3",
 "temperature": 26.09,
 "airPressure": 101573,
 "humidity": 12.09,
 "lightLevel": 45145,
 "batteryCharge": 12.09,
 "batteryVoltage": 58,
 "coolingHealth": 78.0,
 "tyrePressure": 2.1
}
```


2. GET /api/location-history

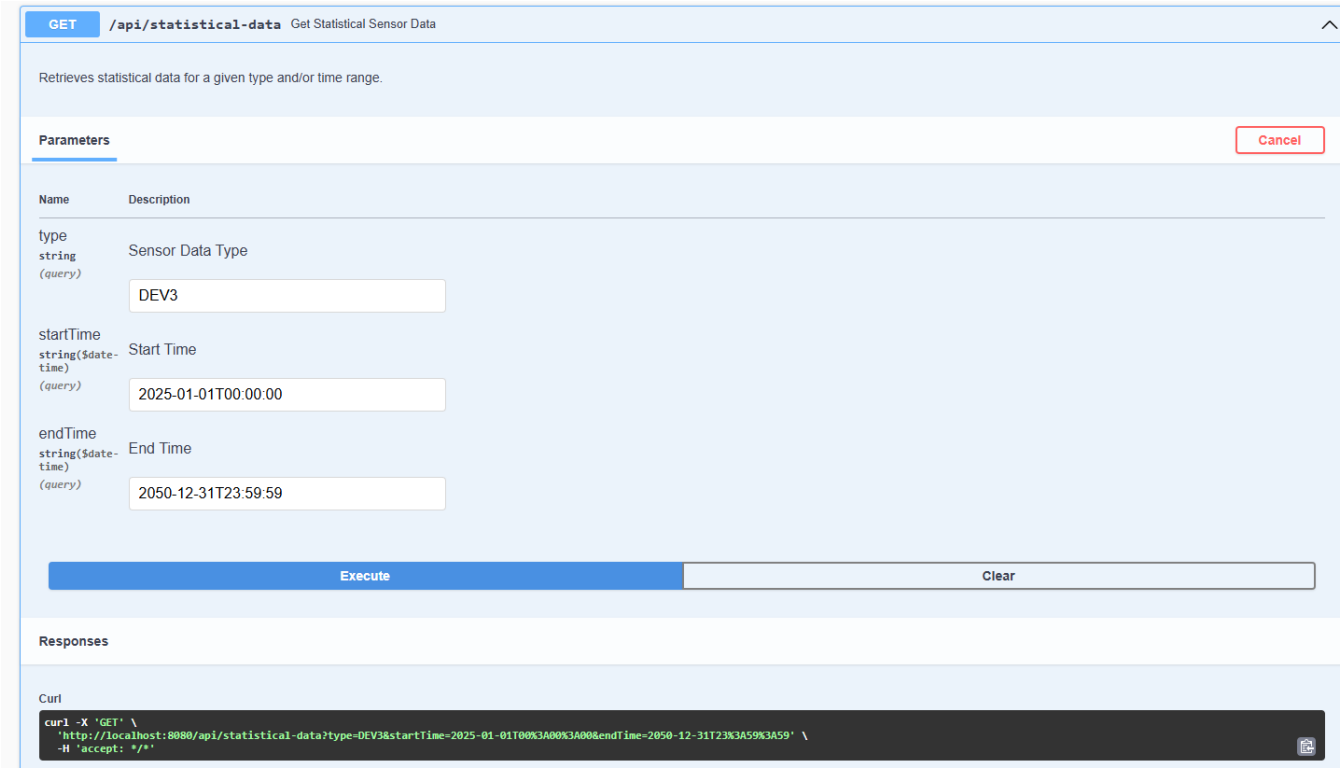


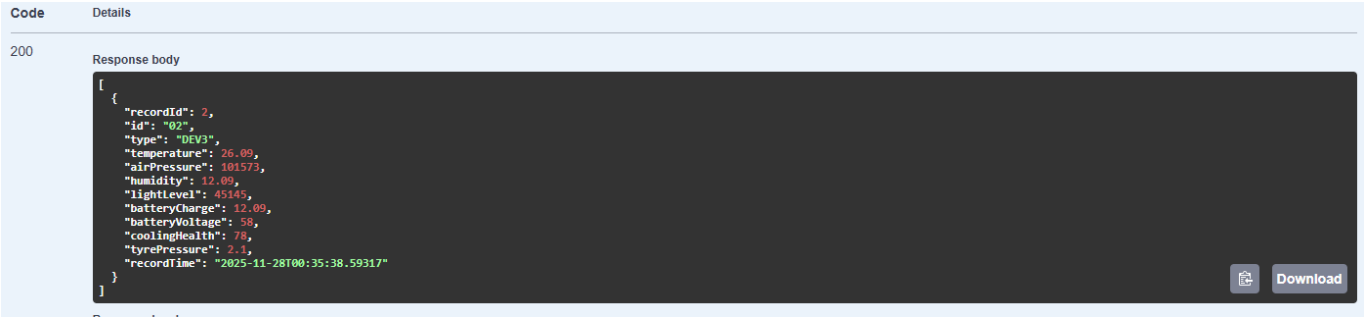


3. GET /api/operational-data



5. GET /api/statistical-data





3.2.2 Public API Documentation & Live Swagger Access

The following links provide direct access to the deployed application and its API documentation.
You can either test the live Cloud Run deployment or view the API definition through SwaggerHub.

1. Live Swagger UI (Cloud Run Deployment)

Directly interacts with the running backend.
URL:
https://sensordatainterpreter-802895490980.europe-west1.run.app/swagger-ui/index.html#

2. SwaggerHub API Documentation

URL:
https://app.swaggerhub.com/apis-docs/mertkaracamm/SensorDataInterpreterAPI/1.0.0

3.1 Testing

The project includes unit tests for the main components responsible for message processing and decision-making. The goal is to ensure that message routing, rule evaluation, data mapping, and controller behavior all work as expected.

1. Message Processing (Routing Logic)

Tests verify that SensorDataProcessingServiceImpl.process() correctly determines whether a message should be handled as:

- **Statistical** (no statusChanges)
- **Operational** (contains statusChanges)

Assertions check that the correct handler is invoked for each case.

2. Statistical Handler Tests

These tests validate that:

- statistical messages are mapped to StatisticalData correctly
- validation runs properly
- records are saved using the StatisticalDataRepository
- timestamps and numeric values are persisted as expected

3. Operational Handler Tests

Tests cover operational-specific logic:

- statusChanges JSON is parsed correctly
- rule values are loaded from operational_config
- battery alarm is triggered under the configured limit
- voltage delta calculation behaves correctly
- the final OperationalData entity contains the computed alarm flags

Metric calculations are mocked only when needed so that business logic stays isolated.

4. MetricCalculator Tests

The voltage delta computation is tested independently:

- correct percentage calculation
- correct threshold comparison
- edge cases (zero values, negative deltas, large jumps)

This ensures the alarm logic remains stable even if upstream logic changes.

5. REST Controller Tests

Using mock services, the following is verified:

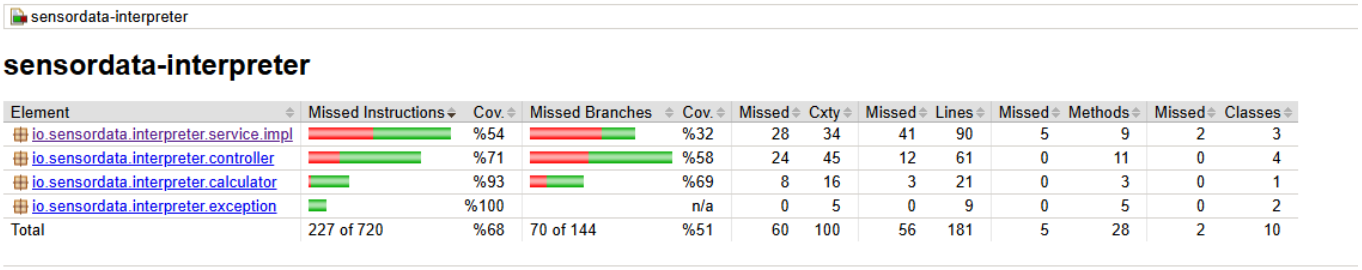
- /api/messages accepts a valid JSON payload and forwards it to the processing service
- query endpoints return the expected HTTP response format
- invalid requests (missing parameters, wrong date format, etc.) return proper error messages

6. Global Exception Handling

Tests confirm that common failure scenarios are mapped to clean, structured error responses:

- missing fields → 400 Bad Request
- processing failures → 500 Internal Server Error
- date parsing errors → consistent error format

3.1.1 Test Coverage



4. Future Improvements

Real message queue integration: Replace the simulated POST ingestion with Kafka or RabbitMQ for true real-time streaming.

Pluggable rule engine: Make operational rules modular so new rules can be added without changing existing code.

Basic anomaly detection: Add simple ML or statistical models to flag unusual sensor patterns.

Geospatial features: Enhance location data with route tracking or geo-fence alerts.

Monitoring & metrics: Add Prometheus/Grafana to track throughput, alarm counts, and system health.