REVALUATING A SYSTEM IMPLEMENTATION TO EMBRACE OPEN SOURCE TIME SERIES DATABASES

A dissertation submitted in partial fulfilment of

the requirements for the degree of

BACHELOR OF SCIENCE in Computer Science

in

The Queen's University of Belfast

by

Dominicus Adjie Wicaksono

'Date of Submission'

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE**

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

|  |  |  |  |
| --- | --- | --- | --- |
| Student Name: | Dominicus Adjie Wicaksono | Student Number: | 40352799 |
| Project Title: | REVALUATING A SYSTEM IMPLEMENTATION TO EMBRACE OPEN SOURCE TIME SERIES DATABASES | | |
| Supervisor: | Charles Gillan | | |

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

**By submitting your dissertation you declare that you have completed the tutorial on plagiarism at** <http://www.qub.ac.uk/cite2write/introduction5.html> **and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.**

1. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

|  |  |  |  |
| --- | --- | --- | --- |
| *Student’s signature* | Dominicus Adjie Wicaksono | *Date of submission* |  |

# Acknowledgements

To those who have helped the author during the project and the preparation of the dissertation and to anybody who has given financial support.

# Abstract

This dissertation introduces a real-time heart rate monitoring system for ICUs that utilizes Apache Kafka for data streaming and employs various time-series databases for backend storage. The system emulates heart rate data acquisition at adjustable rates, facilitates master-standby failover, and displays real-time information for each patient bed. Performance benchmarking was executed on InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL under varying message loads. The results indicate that InfluxDB delivers the maximum throughput, whilst TimescaleDB demonstrates higher memory economy. VictoriaMetrics achieves equilibrium between both aspects. The system exhibits fault tolerance, low latency (around 150ms), and scalability, enhancing a robust healthcare telemetry infrastructure with prospects for real-world ICU implementation and wider medical IoT applications.

Table of Contents

[Acknowledgements 1](#_Toc195806753)

[Abstract 1](#_Toc195806754)

[Table of Contents 2](#_Toc195806755)

[1. Introduction and Problem Area 6](#_Toc195806756)

[1.1. Introduction 6](#_Toc195806757)

[1.2. Problem Statement 6](#_Toc195806758)

[1.3. Background 6](#_Toc195806759)

[1.4. Objectives 6](#_Toc195806760)

[1.5. Scope 7](#_Toc195806761)

[1.6. Justification 7](#_Toc195806762)

[2. System Requirements and Specification 7](#_Toc195806763)

[2.1. Overview 7](#_Toc195806764)

[2.2. Assumptions and Constraints 8](#_Toc195806765)

[2.3. Functional Requirements 8](#_Toc195806766)

[2.4. Non-Functional Requirements 9](#_Toc195806767)

[2.5. Use Case Definitions 9](#_Toc195806768)

[2.6. Software Specification 10](#_Toc195806769)

[2.7. System Interfaces 10](#_Toc195806770)

[2.8. User Characteristics 11](#_Toc195806771)

[3. Design 11](#_Toc195806772)

[3.1. Architectural Overview 11](#_Toc195806773)

[3.1.1. Pre-processing Layer 12](#_Toc195806774)

[3.1.2. Streaming Ingestion via Kafka 12](#_Toc195806775)

[3.1.3. Message Bus and Decoupling 12](#_Toc195806776)

[3.1.4. Consumer and Storage Layer 13](#_Toc195806777)

[3.1.5. Visualisation Layer 13](#_Toc195806778)

[3.1.6. Architectural Summary 13](#_Toc195806779)

[3.2. Data Flow and Sequence of Operations 14](#_Toc195806780)

[3.2.1. Real-Time Visualisation Sequence 14](#_Toc195806781)

[3.2.2. Historical Data Fetch (Optional Flow) 15](#_Toc195806782)

[3.2.3. Background Data Ingestion Pipeline 15](#_Toc195806783)

[3.2.4. Failover Considerations 15](#_Toc195806784)

[3.3. Component Design 15](#_Toc195806785)

[3.3.1. correctedHR.py - Preprocessing Module 16](#_Toc195806786)

[3.3.2. producer.py – Kafka Producer 16](#_Toc195806787)

[3.3.3. consumerMaster.py – Primary Kafka Consumer 17](#_Toc195806788)

[3.3.4. consumerStandBy1.py – Backup Consumer 17](#_Toc195806789)

[3.3.5. display\_program.py 17](#_Toc195806790)

[3.3.6. Dockerised Environment 18](#_Toc195806791)

[3.4. Database Design 18](#_Toc195806792)

[3.4.1. Common Schema Strategy 18](#_Toc195806793)

[3.4.2. InfluxDB 18](#_Toc195806794)

[3.4.3. TimescaleDB 19](#_Toc195806795)

[3.4.4. VictoriaMetrics 19](#_Toc195806796)

[3.4.5. PostgreSQL 20](#_Toc195806797)

[3.4.6. Summary 20](#_Toc195806798)

[3.5. Key Design Decisions and Rationale 21](#_Toc195806799)

[3.5.1. Kafka – Centric Event Streaming Architecture 21](#_Toc195806800)

[3.5.2. Master-Standby Consumer Model 21](#_Toc195806801)

[3.5.3. Multi-Database Benchmarking Support 21](#_Toc195806802)

[3.5.4. Preprocessing via correctedHR.py 22](#_Toc195806803)

[3.5.5. Real-Time Visualisation with Fallback Logic 22](#_Toc195806804)

[3.5.6. Lightweight Dockerised Deployment 22](#_Toc195806805)

[3.5.7. Message Rate Configuration and Replay Control 22](#_Toc195806806)

[3.6. Planned Enhancements 23](#_Toc195806807)

[3.6.1. High Availability Kafka Cluster 23](#_Toc195806808)

[3.6.2. Redundant Producer and Consumer Architecture 23](#_Toc195806809)

[3.6.3. Multi-Node and Federated Databases 23](#_Toc195806810)

[3.6.4. Real ICU Device Integration 24](#_Toc195806811)

[3.6.5. Intelligent Alert System 24](#_Toc195806812)

[3.6.6. Web-Based Visualisation and Grafana Dashboard 24](#_Toc195806813)

[3.6.7. Dynamic Load Testing and CI/CD Pipelines 24](#_Toc195806814)

[4. Implementation 25](#_Toc195806815)

[4.1. Choice of Programming Language and Development Environment 25](#_Toc195806816)

[4.2. Key Libraries and Software Dependencies 25](#_Toc195806817)

[4.3. Implementation Highlights by Component 27](#_Toc195806818)

[4.3.1. correctedHR.py – Preprocessing Module 27](#_Toc195806819)

[4.3.2. producer.py – Kafka Producer 27](#_Toc195806820)

[4.3.3. consumerMaster.py and consumerStandBy1.py – Kafka Consumers 27](#_Toc195806821)

[4.3.4. display\_program.py – Real-Time Query and Visualisation Interface 28](#_Toc195806822)

[4.3.5. Dockerised Environment 28](#_Toc195806823)

[4.4. Data Structures and Types 28](#_Toc195806824)

[4.5. Algorithms and Core Logic 29](#_Toc195806825)

[4.5.1. Message Rate Throttling (MPM Control) 29](#_Toc195806826)

[4.5.2. Resilient Fallback on Database Failure 30](#_Toc195806827)

[4.5.3. Resource Monitoring Per Message 30](#_Toc195806828)

[4.5.4. End-to-End Data Flow Logic 30](#_Toc195806829)

[4.6. Implementation Challenges and Decisions 30](#_Toc195806830)

[4.6.1. Database Driver Inconsistencies 31](#_Toc195806831)

[4.6.2. Timestamp Format Normalisation 31](#_Toc195806832)

[4.6.3. Environment Variable-Based Decoupling 31](#_Toc195806833)

[4.6.4. Flexible JSON Payloads Without Schemas 31](#_Toc195806834)

[4.7. Containerisation and Configuration 32](#_Toc195806835)

[4.7.1. Docker Compose Design 32](#_Toc195806836)

[4.7.2. Environment Variable Strategy 32](#_Toc195806837)

[4.7.3. Port Mapping and Volume Persistence 32](#_Toc195806838)

[5. Testing 33](#_Toc195806839)

[5.1. Test Strategy and Approach 33](#_Toc195806840)

[5.2. Functional Testing 34](#_Toc195806841)

[5.3. Performance Testing 34](#_Toc195806842)

[5.4. Test Tools and Automation 35](#_Toc195806843)

[5.5. Sample Test Evidence 35](#_Toc195806844)

[5.6. Test Coverage and Limitations 36](#_Toc195806845)

[6. System Evaluation and Experimental Results 36](#_Toc195806846)

[6.1. Evaluation Methodology 36](#_Toc195806847)

[6.2. System Performance Results 37](#_Toc195806848)

[6.2.1. Global CPU and Memory Usage (Container-Wide) 38](#_Toc195806849)

[6.2.2. Producer Resource Usage (Per-Message) 39](#_Toc195806850)

[6.2.3. Latency and Throughput 41](#_Toc195806851)

[6.3. Comparative Analysis 43](#_Toc195806852)

[6.4. System Information Impact 44](#_Toc195806853)

[6.5. Functional Observations (Manual Tests) 45](#_Toc195806854)

[6.5.1. Display Program Validation 45](#_Toc195806855)

[6.5.2. Failover Test: Master-to-Standby Transition 46](#_Toc195806856)

[6.6. Limitations and Future Work 47](#_Toc195806857)

[6.6.1. Identified Limitations 47](#_Toc195806858)

[6.6.2. Future Enhancements 47](#_Toc195806859)

[6.7. Societal and Commercial Implications 48](#_Toc195806860)

[6.7.1. Healthcare Applications and Societal Value 48](#_Toc195806861)

[6.7.2. Scalability and Reusability 49](#_Toc195806862)

[6.7.3. Commercial Opportunities 49](#_Toc195806863)

[6.7.4. Risks and Ethical Considerations 49](#_Toc195806864)

[6.7.5. Feasibility and Impact 50](#_Toc195806865)

[7. Appendices 50](#_Toc195806866)

[References 50](#_Toc195806867)

# Introduction and Problem Area

## Introduction

Real-time surveillance in intensive care units (ICUs) is essential for facilitating prompt therapeutic interventions in response to indications of patient decline. Heart rate is a key indicator in assessing cardiovascular health and physiological stress among the measured vital signs. The advent of IoT-based biometric technologies enables hospitals to gather and transmit heart rate data at elevated frequency. This presents data engineering issues concerning dependable ingestion, transformation, and low-latency access to time-stamped data [2].

## Problem Statement

Traditional data storage systems, especially relational databases, are not designed for workloads with high ingestion rates related to time-series telemetry. In ICU situations where data latency and precision are critical, even slight intake delays or losses can jeopardize patient safety. This dissertation examines the architectural and technical problems associated with streaming heart rate data over Apache Kafka and its storage in time-series databases. The primary aim is to assess which database most effectively facilitates a real-time, fault-tolerant pipeline under fluctuating message-per-minute scenarios [2].

## Background

Apache Kafka is a distributed event streaming framework extensively utilized for real-time data processing pipelines, owing to its durability, horizontal scalability, and high-throughput message delivery assurances [1]. Kafka's capacity to separate producers from consumers renders it very appropriate for hospital telemetry systems, where buffering and asynchronous processing are frequently necessary. Concurrently, time-series databases like InfluxDB, TimescaleDB, and VictoriaMetrics have been designed to enhance the storing and querying of temporal data, surpassing conventional SQL systems in time-sensitive scenarios.

## Objectives

The dissertation centres on the development and benchmarking of a Kafka-based ingestion system to assess the performance of time-series databases in real-time scenarios. The primary aims are to:

* Create a producer that emulates heart rate data for an ICU setting,
* Integrate Kafka to transmit data at regulated message-per-minute (MPM) speeds,
* Assess ingestion delay, throughput, CPU use, and memory consumption for each database,
* Establish a master-standby consumer architecture with heartbeat-driven failover mechanisms.

This seeks to identify the optimal and efficient database selection for healthcare telemetry systems [1].

## Scope

The project focuses on backend ingestion infrastructure for ICU heart rate data. It omits analytics, machine learning, and forecasts regarding patient outcomes. All testing are performed locally in a macOS environment utilizing Docker, with the ingestion rate fluctuating between 10 and 1000 MPM. The research assesses three time-series databases (InfluxDB, TimescaleDB, VictoriaMetrics) alongside a baseline SQL implementation, focusing on their ingestion capabilities rather than advanced query optimization [1].

## Justification

Choosing the appropriate time-series storage backend for healthcare telemetry is essential due to the increasing necessity for real-time monitoring in intensive care units. Kafka is a widely adopted streaming solution that guarantees scalability and separation between data production and consumption [1]. Contemporary time-series databases offer enhancements tailored for this workload, potentially diminishing latency and enhancing ingestion accuracy during periods of high demand. This study aids in identifying more resilient and efficient architectures for key healthcare data pipelines by assessing their performance [2].

# System Requirements and Specification

## Overview

This system is intended to assess the efficacy of several time-series database technologies for real-time monitoring of heart rates in an ICU setting. Apache Kafka functions as the primary message bus, facilitating scalable and decoupled communication between a data-producing module and multiple consuming modules. The major objective is to replicate a real-world telemetry ingestion system, assess its performance under regulated load, and evaluate the ingestion capabilities of several databases, specifically InfluxDB, TimescaleDB, VictoriaMetrics, and a baseline PostgreSQL implementation. The system is modular, encapsulated, and designed for reproducibility.

## Assumptions and Constraints

* All development and testing were conducted on macOS M1, featuring a 10-core CPU and 16 GB of RAM.
* Docker containers are utilised for all services, including Kafka, Zookeeper, and each database.
* Every test setup operated for 300 seconds each load tier.
* The display program and standby consumer modules were developed but not incorporated into the final testing framework.
* No graphical user interface is available; visualisation was conducted manually utilising .csv data and plotting tools.

## Functional Requirements

|  |  |
| --- | --- |
| **ID** | **Requirement** |
| FR1 | The system will transmit heart rate data from hr.csv to Kafka at a configurable rate of 10 to 1000 messages per minute (MPM). |
| FR2 | Kafka will retain all incoming messages in the heart\_rate topic. |
| FR3 | The primary consumer will retrieve messages from Kafka and record them in the active target database. |
| FR4 | The designated database will retain heart rate data along with a timestamp and bed identifier. |
| FR5 | The system shall facilitate the selection among four database alternatives: InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL. |
| FR6 | The system shall record CPU and memory utilisation for each database container during every test. |
| FR7 | The system shall record write latency for each message in milliseconds. |
| FR8 | The system shall facilitate automated testing at various MPM rates using run\_all\_tests.py. |
| FR9 | The system shall generate .csv logs for general system metrics, individual message logs, and write latency. |
| FR10 | The system shall facilitate configuration through environment variables (e.g., ACTIVE\_DB, MESSAGES\_PER\_MINUTE). |
| FR11 | The display program will enable users to monitor real-time heart rates by bed number. |
| FR12 | The standby consumer will activate upon the failure of the master consumer, as indicated by the heartbeat detection. |

## Non-Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **NFR** | **Target / Status** |
| NFR1 | System must process up to 1000 messages/minute reliably. | Achieved |
| NFR2 | CPU usage per DB container should remain under 60% at 500 MPM. | Exceeded by InfluxDB |
| NFR3 | Average DB write latency should stay under 150ms. | Met by TimescaleDB and VictoriaMetrics |
| NFR4 | Memory usage per container should remain under 200MB. | Met by all except InfluxDB |
| NFR5 | Failover from master to standby consumer must occur within 5 seconds. | Not implemented |
| NFR6 | All components must run in Docker for consistent deployment. | Achieved |
| NFR7 | Each test run should auto-generate exportable logs in .csv format. | Achieved |
| NFR8 | System should be extensible to support additional databases or output visualizations. | Structure allows for extension |

## Use Case Definitions

|  |  |
| --- | --- |
| **Use Case** | **Description** |
| UC1 | Data Ingestion Simulation – Streams ICU heart rate data to Kafka at configured message rates. |
| UC2 | Multi-Database Ingestion – Kafka consumer writes data into one of four supported databases. |
| UC3 | Heartbeat Monitoring & Failover – Master emits heartbeats; standby takes over on failure. \*(Planned)\* |
| UC4 | Data Query Interface – System supports real-time querying by bed number. \*(Planned)\* |
| UC5 | Performance Benchmarking – Logs CPU, memory, latency, throughput, and generates .csv reports. |

## Software Specification

The final system consists of the following Python-based modules:

* **producer.py**: Streams records from hr.csv to Kafka at a user-defined rate.
* **consumerMaster.py**: Reads Kafka messages and writes them into the selected database.
* **monitor\_resources.py**: Captures CPU and memory usage via Docker stats.
* **run\_all\_tests.py**: Orchestrates test cycles, changing databases and message rates automatically.
* **Kafka Configuration**: Set up using Docker Compose with defined topics for heart\_rate and heartbeat.
* **Data Input**: hr.csv with >700,000 timestamped heart rate entries.
* **Logging & Output**:
  + global\_summary.csv (per test configuration metrics)
  + per\_message\_summary.csv (latency per message)
  + write\_summary.csv (total written messages and gap)

All code was written in **Python**, containerized via **Docker**, and optionally monitored using **Prometheus** for real-time inspection.

## System Interfaces

|  |  |
| --- | --- |
| **Interface** | **Description** |
| Kafka Producer API | Used in producer.py to publish messages to heart\_rate topic. |
| Kafka Consumer API | Used in consumerMaster.py to consume and forward messages. |
| InfluxDB API | Accessed via influxdb-client (Python SDK). |
| TimescaleDB Interface | Accessed via psycopg2 (PostgreSQL SQL statements). |
| VictoriaMetrics API | HTTP POST interface for direct ingestion. |
| PostgreSQL Interface | Standard SQL insert statements via psycopg2. |
| Internal Environment Vars | ACTIVE\_DB, MESSAGES\_PER\_MINUTE, TEST\_DURATION. |
| Output Logs | Global/per-message/latency logs saved as .csv files. |

## User Characteristics

The anticipated user is a technical researcher, systems engineer, or performance tester possessing a fundamental comprehension of Python, Docker, and Kafka. A graphical interface has not yet been implemented. Users engage through:

* Terminal commands
* Environment variables
* Logs and output files

This system does not necessitate domain-specific medical expertise, as it operates on pre-recorded, anonymised data.

# Design

## Architectural Overview

A diagram of a service

AI-generated content may be incorrect.

Built under modularity, fault tolerance, and scalable evaluation of real-time data streams, the heart rate telemetry system is The platform essentially replics an actual ICU monitoring workflow using Apache Kafka as the main communication tool whereby biometric data—especially heart rate—is acquired, processed, and stored over several database backends. To enable deployment, isolation, and scalability enhancement, Docker helps to entirely containerise the system.

Data pre-processing, streaming ingestion, message bus and decoupling, consumer and storage layer, and visualisation interface make up five main layers forming the design. Every component of independent testing, failover simulation, and future extensibility is just tangentially linked.

### Pre-processing Layer

Given as HR.csv, the raw ICU telemetry data reveals inconsistent formatting and embedded metadata. From entries that fit given regular expression patterns, a preprocessing script (correctedHR.py) is used to extract just the pertinent values—bed number and heart rate. For the streaming layer, the output, extracted\_hr\_data.csv, offers a fresh, reusable dataset. This separation ensures consistent, ordered input without rework of the source file for next Kafka modules.

### Streaming Ingestion via Kafka

Found in producer.py, the ingestion component sends each row as a structured JSON message under the heart\_rate topic from extracted\_hr\_data.csv. Every message consists of bed\_number, heart\_rate, and an ISO-style UTC timestamp. Messages are sent at a controlled rate determined by the environment variable MESSAGES\_PER\_MINUTE, so enabling automated testing over several consumption loads (from 10 to 1000 MPM).

Kafka's fault-tolerant, high-throughput design and ability to separate data producers from consumers attracted me. This method lets parallel consumers or additional services be included without upsetting the producer flow and allows scalable downstream processing.

### Message Bus and Decoupling

There is mostly data flowing from Kafka. It helps the producer and many customer cases to have asynchronous communication. This arrangement guarantees strong decoupling: consumers can independently scale, recover from mistakes, or reprocess past data without impacting upstream components while producers remain blind to the data locations or consumption patterns. Using Docker Compose, all Kafka services are run with localhost: 29092 accessible connections.

### Consumer and Storage Layer

The principal consumer, consumerMaster.py, subscribes to the heart\_rate topic and transmits data to one of four time-series or relational databases according to the current configuration:

* InfluxDB via its Python client
* TimescaleDB via psycopg2
* VictoriaMetrics via Prometheus-compatible HTTP ingestion
* Baseline PostgreSQL for comparison

The target database records every incoming message; structured log files contain write confirmation records for every test. Through port 8000 the customer supplies real-time Prometheus metrics including processed count, failed count, and consumer lag.

ConsumerStandBy1.py is a secondary module that copies the ingestion mechanism and logs data into a backup InfluxDB instance. Though not now set up for automatic failover, it is meant to engage upon identification of master failure in line with the high-availability goal of the project.

### Visualisation Layer

The user interface component, display\_program.py, is developed using Python's tkinter and matplotlib packages. The system requests the user to provide a bed number and thereafter interrogates InfluxDB (primary) to generate a real-time, time-series graph of the heart rate for that specific bed. In the event of InfluxDB unavailability, the software automatically transitions to query the standby InfluxDB replica.

Historical data from the last 24 hours and real-time data from the last minute are visualised through Flux queries. The present design retrieves data from the database, whereas a prospective enhancement is to enable this component to directly access data from Kafka, hence minimising read latency and completely divorcing the visualisation layer from storage.

### Architectural Summary

The complete system is implemented via Docker, with separate containers for Kafka, Zookeeper, each database, and the auxiliary Python scripts. The design guarantees:

* Modularity for plug-and-play database benchmarking
* Clear separation between pre-processing, streaming, and visualization
* Extensibility to integrate new databases or failover consumers
* Observability through Prometheus and CSV-based logging

## Data Flow and Sequence of Operations

A screenshot of a computer screen

AI-generated content may be incorrect.

The sequential processes of the intended real-time heart rate monitoring system are described in this part. From its source—a stationary.csv file—the sequence shows how data moves across several processing phases to the display interface used for real-time visualisation. Emphasising important interactions among system components, it covers both the user-initiated data retrieval pathway and the background input pipeline.

### Real-Time Visualisation Sequence

The interaction starts when the user wants to keep an eye on a specific bed number using the display tool. After this input, the program searches the Active Time-Series Database (Active Time-Series Database—potentially InfluxDB, Timescale Database, VictoriaMetrics, or PostSQL—for heart rate readings connected with the specified bed identification. Should the main database be inaccessible or nonresponsive, the program automatically moves to querying the Standby TSDB, so guaranteeing availability and minimising data access interruptions.   
Each database's data includes timestamped heart rate readings, which are examined and graphically shown with matplotlib. Every minute the graphic refreshes to show near real-time conditions, so enabling quick monitoring without requiring a browser-based interface.

### Historical Data Fetch (Optional Flow)

The system also makes user-initiated historical enquiries possible. Under user request, the display program accesses active database historical heart rate data for up to 24 hours. This alternative interaction uses an extended temporal framework for retrospection yet reflects the real-time trajectory. Visualisation is once more done locally using dynamic graph plotting.

### Background Data Ingestion Pipeline

Concurrent background processes ensure continuous new heart rate data acquisition. The pipeline starts with the execution of corrected HR.py, which handles raw ICU data (hr.csv) and generates a refined version (extracted\_hr\_data.csv) including just bed numbers and matching heart rate values. This phase of reorganisation ensures congruence with the streaming format.   
The producer.py script handles the cleaned dataset and sends data records to Apache Kafka at a configurable 10 to 1000 messages per minute. Every record is set out as a JSON object with a UTC timestamp, heart rate, and bed\_number. Through its heart\_rate topic, Kafka organises these messages such that they are momentarily saved and available to subscribed customers.  
Two Kafka consumers—consumerStandby.py and consumerMaster.py—eat from this topic. While the secondary consumer replicas this process for the standby database, guaranteeing redundancy at the ingestion level, the main consumer records arriving data to the designated active TSDB. Consumers can be configured using environment variables to direct particular databases or change message throughput; they also operate independently.

### Failover Considerations

Although the failover method for consumer switching—that is, heartbeat-based activation of a standby consumer—is not totally finished—the display program already allows failover at the read level. The program searches the standby replica automatically should the master database fail. Through consumer heartbeat monitoring, this partial failover design increases system resilience and enables future enhancements.

## Component Design

The basic system modules that together run a real-time heart rate intake and monitoring pipeline for environments in intensive care units are described in this part. Each element has been designed with modularity and decoupling to offer fault tolerance, future integration, and scalability. To ensure consistent, low-latency performance across a range of load conditions, the system makes use of containerising, asynchronous messaging with Kafka, and dedicated time-series databases.

### correctedHR.py - Preprocessing Module

The preprocessing tool polishes and arranges the raw ICU telemetry data (hr.csv) into a format fit for streaming. The original file has fields with inconsistent formatting and extraneous metadata. Corrected HR.py uses regular expressions to extract just the relevant fields—more especially, bed numbers and heart rate readings—to fix this. Extracted\_hr\_data.csv, which serves as the producer's input source, stores the output to a new file.

Principal Attributes:

* Regex-based parsing tailored to ICU log patterns.
* Automatic exclusion of malformed or incomplete entries.
* Outputs a normalized, lightweight CSV optimized for real-time ingestion.

### producer.py – Kafka Producer

At a configurable frequency—messages per minute—the producer module sends ICU heart rate data to a Kafka topic. Using the Confluent Kafka Python client, it first reads from extracted\_hr\_data.csv, converts each record into a structured JSON object, then posts it to the heart\_rate topic. It also notes system-level resource use ( CPU and RAM) for every message to support performance evaluation.

Principal Attributes:

* Message rate controlled via the MESSAGES\_PER\_MINUTE environment variable.
* JSON serialization for structured message payloads.
* Uses psutil to capture and log per-message CPU and memory usage.
* Automatically loops through the dataset to maintain long test durations.

Sample JSON Payload:

{

"bed\_number": "613",

"timestamp": "2025-04-16T15:20:00.000Z",

"heart\_rate": 117

}

### consumerMaster.py – Primary Kafka Consumer

This module records heart rate signals imported from Kafka in the active time-series database. ACTIVE\_DB environment variable dynamically detects the target database. It works with VictoriaMetrics, TimescaleDB, InfluxDB, and a basic PostSQL implementation. The customer uses optimised insertion techniques catered to each database type, Prometheus instrumentation for monitoring, and error-handling strategies.

Key Features:

* Multi-database compatibility through modular write functions.
* ISO-8601 timestamp validation for temporal integrity.
* Prometheus metrics exposure (consumer lag, write success/failure rates).
* Centralized logging for write operations.

Supported Interfaces:

* InfluxDB: influxdb-client (Python SDK)
* TimescaleDB & PostgreSQL: psycopg2 (SQL)
* VictoriaMetrics: HTTP POST endpoint using line protocol

### consumerStandBy1.py – Backup Consumer

A fault-tolerant backup for data intake is the standby consumer. Designed to write only to a redundant InfluxDB instance, it is technically equivalent to the main consumer but It runs concurrently and follows the same Kafka topic, so ensuring constant data availability should a master path failure be the result.

Key Features:

* Passive redundancy without interfering with primary ingestion.
* Continuous subscription to the heart\_rate topic.
* Flexible design for scaling to support additional backup targets.
* Compatible with InfluxDB standby deployments.

### display\_program.py

Through a graphical interface, this optional but necessary component gives customers real-time access to heart rate data. Once a bed number has been chosen, the application searches either the InfluxDB master or its backup and shows the resulting data in a dynamic line graph. The system uses Tkinter for graphical user interface interaction and Matplotlib for data visualisation; it also includes backup querying logic.

Key Features:

* Real-time graphing updated every minute using FuncAnimation.
* Automatic fallback to standby database if the master becomes unavailable.
* Intuitive bed selection via a Tkinter prompt.
* Dual-mode query support: historical (24h) and real-time (last 1 min).

### Dockerised Environment

All components are containerised using Docker to guarantee portability, reproducibility, and isolation. A Docker Compose setup orchestrates the subsequent services:

* zookeeper-1, kafka-1 for message brokering
* influxdb-master, influxdb-standby for time-series storage
* timescaledb, victoria-metrics, baseline\_sql as additional database options

Every container has internal connectivity and exposes ports for local access and inter-service communication. To avoid host conflicts, the Kafka ecosystem runs on assigned ports—such as 29092. This design helps to enable consistent testing conditions and quick environmental resets.

## Database Design

Four backend time-series storage solutions—InfluxDB, TimescaleDB, VictoriaMetrics, and Postgresql—are supported by the real-time heart rate monitoring system. Under consistent intake loads, this flexibility helps to enable similar performance evaluation. Mostly indexed by timestamps and patient bed identification, all schemas are built to support high-frequency physiological signals.

### Common Schema Strategy

Across all databases, the core schema captures three essential fields:

* bed\_number: A unique identifier for the patient ICU bed.
* timestamp: A precise datetime field used for temporal indexing.
* heart\_rate: The measured heart rate value (in bpm).

While the physical representations vary, the logical schema remains consistent to ensure fair benchmarking and interoperability across querying modules.

### InfluxDB

Bucket: sensor\_data | Retention: infinite | Shard Group shows InfluxDB v2.x set up with a sensor\_data bucket with an unlimited retention policy. Length: 168 hours. The Flux line protocol consumes data; each record is noted under a measurement name, such heart\_rate. Tags record bed number; heart rate is recorded as a field. Automatically analysed and catalogued is the timestamp. For failover and redundancy, a standby instance replics the master's schema.

Key Points:

* Schema is implicit, derived at ingest.
* Historical queries use Flux via the display module.
* Retention policies are disabled (infinite storage).
* Pre-validated ingestion ensures no malformed points enter.

### TimescaleDB

TimescaleDB builds upon PostgreSQL, extending it with native time-series capabilities via **hypertables**. The schema for the sensor\_data table is explicitly defined:  
CREATE TABLE sensor\_data (

time TIMESTAMPTZ NOT NULL,

bed\_number INTEGER NOT NULL,

heart\_rate DOUBLE PRECISION NOT NULL

);

It has been converted into a hypertable using:

SELECT create\_hypertable('sensor\_data', 'time');

The hypertable is partitioned along the time dimension, enabling efficient insertion and range queries.

Key Points:

* Indexed on time DESC for high write throughput.
* Table validated with 4 chunks during testing.
* Supports Prometheus/Grafana dashboards via PostgreSQL plug-ins.

### VictoriaMetrics

VictoriaMetrics is configured to receive data via **influx-compatible** endpoints, with a retention period of **1 day**, as confirmed by:

flag{name="retentionPeriod", value="1"}

It stores data in blocks and indexes using label sets. The system uses a minimalistic schema with the following structure per metric:

metric\_name: heart\_rate

labels: bed\_number

timestamp: epoch-based

value: heart rate (int)

While VictoriaMetrics does not impose a table schema, all incoming data must adhere to a valid metric-label-value format. Internal logs verify that 13,318 rows have been handled within 612 blocks.

Key Points:

* Stateless ingestion via HTTP POST.
* Default cache and concurrency optimizations.
* Prometheus-compatible query APIs.
* Ideal for high-ingestion workloads, low retention.

### PostgreSQL

The baseline relational implementation uses PostgreSQL 14.x with a traditional SQL table named heart\_rate\_data. It includes a synthetic id primary key for indexing:

CREATE TABLE heart\_rate\_data (

id SERIAL PRIMARY KEY,

bed\_number VARCHAR(10),

timestamp TIMESTAMPTZ,

heart\_rate INTEGER

);

This schema is not optimized for time-series operations but serves as a comparative baseline against specialized TSDBs. No partitioning or hypertable logic is applied.

Key Points:

* Acts as control/reference DB.
* Non-time-series optimized.
* Suitable for OLTP-style comparisons.

### Summary

The multi-database design balances analytical precision with adaptability. TimescaleDB has a relational structure fit for time-series workloads while InfluxDB and VictoriaMetrics offer fast write paths with minimal schema enforcement. Although not especially designed for time-series database applications, Postgres is absolutely essential in understanding performance trade-offs in general-purpose databases. By using a consistent logical schema across backends and allowing master-standby configurations, the solution enables exact, consistent benchmarking under controlled environments.

## Key Design Decisions and Rationale

The main architectural and implementation decisions taken during the design process of the real-time heart rate monitoring system are described in this part. Supported by empirical data and accepted best practices in distributed systems engineering, every choice reflects a compromise among performance, scalability, dependability, and maintainability.

### Kafka – Centric Event Streaming Architecture

A Kafka-based message bus was selected as the backbone of the data pipeline due to its high throughput, partitioned design, and fault-tolerant publish-subscribe model. This enabled:

* Loose coupling between producer and consumer processes.
* Scalable ingestion, accommodating future ICU units or hospital wings.
* Replayability of streams for debugging, analytics, or delayed database insertions.

This architecture conforms to industry standards in telemetry processing, where resilience and buffer capacity are essential during fluctuating data flows.

### Master-Standby Consumer Model

To ensure data resilience and operational continuity, the consumer architecture was explicitly designed with:

* A primary consumer writing to the active database.
* A standby consumer mirroring the Kafka topic but writing to a separate fallback instance.

Redundancy like this lowers the possibility of data loss should consumer failures or master database unavailability arise. The two consumers operate concurrently but independently, so improving fault separation and providing strengthening guarantees of recovery.

### Multi-Database Benchmarking Support

The system accommodates four backend time series/database systems (InfluxDB, TimescaleDB, VictoriaMetrics, PostgreSQL) via a plug-and-play interface regulated by the ACTIVE\_DB environment variable. This design permits:

* Fair benchmarking across diverse database paradigms (columnar, relational, compressed).
* Seamless database failover testing under identical workloads.
* Empirical insights into write performance, storage efficiency, and query latency.

The abstraction separates the ingestion logic from storage-specific implementations, enabling future database integration.

### Preprocessing via correctedHR.py

The initial dataset (hr.csv) displayed structural and content discrepancies, requiring a comprehensive preprocessing phase. By implementing correcterHR.py, the system:

* Ensures schema compliance before streaming (bed number, timestamp, heart rate).
* Removes malformed or irrelevant entries using regex parsing.
* Converts input into a clean CSV for high-performance streaming.

The pre-ingestion cleansing enhances producer dependability and downstream query precision.

### Real-Time Visualisation with Fallback Logic

The display application includes a user interface that retrieves real-time heart rate data and visualises it using Matplotlib. In the event of a primary InfluxDB failure, the system:

* Automatically falls back to a standby instance.
* Maintains UI responsiveness with no additional user intervention.
* Continues updating the plot every minute.

This proactive error-handling solution ensures clinical reliability by maintaining access to live data despite backend disturbances.

### Lightweight Dockerised Deployment

All components—Kafka, Zookeeper, database services, and bespoke Python scripts—are encapsulated using Docker Compose. This:

* Simplifies setup and teardown during development and testing.
* Guarantees environment consistency across platforms and collaborators.
* Enables isolated benchmarking by independently restarting containers.

This option also facilitates future migration to cloud-native deployment tools such as Kubernetes, if necessary.

### Message Rate Configuration and Replay Control

The producer accommodates a changeable MESSAGES\_PER\_MINUTE (MPM) rate, facilitating adaptable simulation situations. It is capable of cyclically replaying the dataset for prolonged stress testing. This control is essential for:

* Model varying ICU workloads (e.g., active vs. idle shifts).
* Assess database saturation thresholds.
* Capture longitudinal system behavior across ingestion rates.

## Planned Enhancements

Many suggested improvements have been identified to ensure the continuous viability and adaptability of the real-time ICU heart rate monitoring system. These changes try to increase system resilience, increase deployment options, and prepare the platform for possible clinical use. This part outlines both long-term and short-term developmental paths guided by known limitations and expected needs.

### High Availability Kafka Cluster

Currently in use for simplicity of use is a single-node Kafka broker. This generates a single point of failure even if it is sufficient for local testing and controlled environments. Later versions will offer a multi-broker Kafka cluster run under Zookeeper to enable:

* Leader election across partitions
* Fault-tolerant message replication
* Better load distribution across consumers

This is especially critical when scaling across multiple ICUs or floors.

### Redundant Producer and Consumer Architecture

The producer remains one point of failure even if the system uses backup databases and standby consumers. Suggested improvements include:

* Failover-capable Producer Cluster: Multiple producer instances operating in hot-standby or round-robin configurations to eliminate downtime during restarts or failures.
* Heartbeat Health Checks: Each module will implement a lightweight heartbeat signal and self-report to the Manager Program for real-time health monitoring.

### Multi-Node and Federated Databases

Current testing is restricted to single-node database implementations. Clinical environments frequently necessitate high availability, replication, and distributed writes. Proposed enhancements encompass:

* TimescaleDB with multinode hypertables
* InfluxDB Enterprise clustering with Kapacitor and Chronograf
* Federated PostgreSQL for query delegation and sharding

This would permit each hospital floor or ward to function autonomously while facilitating aggregate queries across nodes.

### Real ICU Device Integration

Currently, heart rate values are generated from a static CSV file. Future iterations will incorporate actual ICU telemetry systems to facilitate production deployment. Protocols and standards to be targeted encompass:

* HL7 (Health Level 7) for interoperability with hospital information systems
* DICOM Waveform Storage for device-based heart rate streams
* FHIR APIs for standardized real-time data exchange

This will allow real-world deployment and clinical validation.

### Intelligent Alert System

Currently, heart rate anomalies must be inferred visually via the display module. An enhancement will include a rules-based and ML-enhanced alert system to automatically:

* Detect tachycardia, bradycardia, and flatlines
* Send alerts to dashboards, email, or on-call systems (e.g., PagerDuty, SMS)
* Train on past data to predict cardiac events

This module will tightly integrate with Prometheus and use Grafana Alertmanager or custom logic.

### Web-Based Visualisation and Grafana Dashboard

The current visualization is built using Tkinter and Matplotlib, which is suitable for local testing but not ideal for multi-user environments. Planned upgrades include:

* Migration to a Web Dashboard (Flask + React)
* Grafana Dashboard Integration for time-series querying and alert visualization
* Mobile Compatibility for on-call staff

These improvements will also support RESTful APIs for data export and remote queries.

### Dynamic Load Testing and CI/CD Pipelines

To optimise testing and guarantee performance under fluctuating workloads, forthcoming development encompasses:

* Locust-based load simulation to replicate ICU-scale data volumes
* GitHub Actions for CI/CD with Docker build/test/deploy automation
* Dynamic environment switching between development, staging, and production containers

# Implementation

## Choice of Programming Language and Development Environment

Due to its expressive syntax, fast development powers, and extensive library ecosystem, Python became the main language used in this effort. With strong support for Kafka consumers, time-series database connections, and system monitoring tools, Python is particularly adept at building a modular, real-time data processing pipeline. Large package availability and dynamic typing helped all system components iterate quickly and prototype easily.For lightweight automation chores including test run coordination, container initialisation, and command chaining during system benchmarking,bash scripting was used concurrently. Because the project was contained and Bash gave enough control, it did not add any additional dependencies or orchestration mechanisms.macOS presented a consistent, UNIX-based environment fit for the required tools, so enabling development and testing. Reliable container management enabled by macOS's natural support for Docker Desktop drives the operation of services including Kafka, Zookeeper, and several time-series databases (InfluxDB, TimescaleDB, VictoriaMetrics, and Postgres) within isolated environments.Visual Studio Code (VS Code) was used in implementation, improved by several important extensions. These cover:

* Python (by Microsoft) – for syntax highlighting, linting, and debugging.
* Docker – for managing container states and volumes directly from the editor interface.

This arrangement enabled seamless transitions between the building of system modules, performance monitoring of containers, and validation of data flows—that is, so improving the development lifetime and system dependability.

## Key Libraries and Software Dependencies

The system makes use of a diverse but targeted collection of libraries and technologies to support real-time data streaming, effective storage, visualisation, and monitoring. Principal implementation language is Python, augmented by carefully selected libraries supporting Kafka integration, time-series database interaction, graphical representation, and performance metrics.

The foundation of the containerised system is Docker and Docker Compose, which encapsulate each database service and facilitate cross-platform compatible communication architecture for consistency. Version control was upheld using Git and GitLab; architectural documentation and diagram generation employed simpler tools such as PlantUML.

A breakdown of key libraries and dependencies is provided below:

|  |  |  |
| --- | --- | --- |
| **Category** | **Library / Tool** | **Purpose** |
| Messaging | confluent\_kafka | High-performance Kafka producer/consumer client for streaming HR data |
| Messaging | psutil | Tracks per-message CPU and memory usage for performance benchmarking |
| Time-Series Databases | influxdb-client, influxdb | Interfaces with InfluxDB 1.x and 2.x for heart rate storage and queries |
| Time-Series Databases | psycopg2-binary | Connects to PostgreSQL/TimescaleDB for structured time-series writes |
| Time-Series Databases | requests | Used for sending HTTP POST requests to VictoriaMetrics endpoints |
| Data Processing | pandas, numpy | Efficient data manipulation and preprocessing during ingestion |
| Data Processing | re, datetime (stdlib) | Regular expression parsing and timestamp formatting |
| Visualization | matplotlib | Dynamic graph rendering in the display module |
| Visualization | tkinter (stdlib) | Lightweight GUI for real-time user interaction |
| Containerization | Docker, Docker Compose | Containerizes Kafka, Zookeeper, databases for reproducible execution |
| Version Control | Git, GitLab | Tracks development history and enables remote collaboration |
| Diagramming | PlantUML, diagrams | Architecture diagram generation for documentation purposes |
| Monitoring (optional) | Telegraf (external, optional) | System-level metrics collection into InfluxDB for enhanced observability |

These libraries were selected for their maturity, performance, and interoperability with the ecosystem of containerised, stream-based pipelines. By synchronising tools across all modules, the system upholds uniform integration standards and reduces interoperability issues.

## Implementation Highlights by Component

This part defines how each main module of the system is operational. Though thorough code listings are not included, the explanations clarify the main algorithmic strategies and structural issues supporting each component.

### correctedHR.py – Preprocessing Module

Extracted structured heart rate data from the raw ICU dataset (hr.csv) using the correctedHR.py script The original file's uneven formatting and metadata make direct streaming unsuitable. Using regular expressions, the module chooses rows with appropriate telemetry readings—more especially, bed numbers and the corresponding heart rate values. Using traditional file I/O operations, the parsed results are stored in a pristine CSV file (extracted\_hr\_data.csv), with invalid or malformed lines deleted during preprocessing eliminated.

### producer.py – Kafka Producer

Utilising the corrected HR.py script, extracted structured heart rate data from the raw ICU dataset (hr.csv). The irregular formatting and metadata of the original file make direct streaming inappropriate. Using regular expressions, the module selects rows with suitable telemetry readings—more especially, bed numbers and the related heart rate values. The parsed results are kept in a pristine CSV file (extracted\_hr\_data.csv) with invalid or malformed lines deleted during preprocessing eliminated using conventional file I/O operations.

### consumerMaster.py and consumerStandBy1.py – Kafka Consumers

Retrieving messages from the Kafka broker, the main and auxiliary consumers also write them concurrently to time-series databases. Driven by the ACTIVE\_DB environment variable, the master consumer dynamically accommodates InfluxDB, TimescaleDB, VictoriaMetrics, and Postgresql by guiding each message to the database. Operating independently and constantly recording data to a backup InfluxDB instance, the standby consumer guarantees redundancy.  
Consumers check data integrity, understand the input timestamp, and make use of client libraries particular to their databases for best insertion. Including acknowledgements and exception handling techniques will help to ensure dependability and fault separation.

### display\_program.py – Real-Time Query and Visualisation Interface

The display module provides interactive means of querying and visualising real-time heart rate data. Users enter a bed number using a simple Tkinter graphical user interface. The application then searches InfluxDB for the most recent readings from the past several minutes. Should a query fail or a database unavailability arise, the system instantly moves to the backup InfluxDB node.  
refreshed every 60 seconds, the retrieved data is visualised using Matplotlib in a dynamically changing graph. Inside, a deque buffer maintains a rolling window of values to enable more seamless animations and reduce redraw overhead.

### Dockerised Environment

The display module provides interactive means of querying and visualising real-time heart rate data. Users enter a bed number using a simple Tkinter graphical user interface. The application then searches InfluxDB for the most recent readings from the past several minutes. Should a query fail or a database unavailability arise, the system instantly moves to the backup InfluxDB node.  
refreshed every 60 seconds, the retrieved data is visualised using Matplotlib in a dynamically changing graph. Inside, a deque buffer maintains a rolling window of values to enable more seamless animations and reduce redraw overhead.

## Data Structures and Types

Message serialisation throughout the Kafka pipeline was done using a consistent and simplified JSON structure. Every message has three fundamental components: bed\_number (string), timestamp (ISO 8601 format), and heart\_rate (integer). This light-weight architecture ensures component interoperability and helps ingestion across many time-series databases.  
Messages are generated in the producer phase by parsing a structured CSV file and turning every row into a Python dictionary. Serialised into JSON before being published to the heart\_rate Kafka topic, the dictionaries are JSON.

Python's dictionary type is used rather widely inside to show every message payload. The datetime module analyses and verifies timestamp fields to give storage backend temporal consistency. The bed\_number is set as a string to allow alphabetic IDs for future improvements.  
Real-time graphing in the display software maintains a rolling buffer of heart rate data points using a deque structure. As the plot window moves over time, this makes data quick insertion and removal easy. Matplotlib handles visualisation; each data point is shown as a tuple of timestamp and heart rate values.

In preprocessing and consumer modules, the system uses pandas and numpy for data manipulation and type coercion, so enabling the management of structured arrays, enforcing type safety, and enabling fast transformations.

These data structures taken together provide a synthesis of clarity, speed, and adaptability—guaranteeing consistent message formatting, best memory use, and compatibility with many downstream users.

## Algorithms and Core Logic

The system offers real-time performance, efficient fault tolerance, and thorough observability by means of several lightweight but necessary control systems. From intake to visualisation, these methods cover the whole data flow pipeline.

### Message Rate Throttling (MPM Control)

To replicate real-time ICU data streaming, the producer employs a messages-per-minute (MPM) throttling technique. Instead of transmitting messages at maximum speed, each message delivery is postponed by a predetermined interval:

Temporal values control this rate-limiting mechanism.Add a delay between message transmissions. The throttle ensures that, during testing, intake rates are reasonable and controlled, so providing important downstream performance measures.

### Resilient Fallback on Database Failure

A lightweight **failover mechanism** is embedded within the display module. Upon query failure to the primary database (e.g., InfluxDB Master), the system retries the same query against a designated standby instance (e.g., InfluxDB Standby). This conditional retry logic ensures minimal interruption in visualization, and provides high availability without requiring full cluster orchestration.

### Resource Monitoring Per Message

Every producer message uses psutil to activate resource logging routines, recording CPU and memory consumption snapshots straight before and following publication to Kafka, so evaluating ingestion performance. Timestamps and message information help to record these measurements, so producing a structured performance log that one can examine to assess the operational footprint of the system under various loads.

### End-to-End Data Flow Logic

The pipeline follows a clear event-driven architecture:

1. Preprocessed CSV records are parsed line by line.
2. Each record is converted into a JSON object and sent to the Kafka broker.
3. The master consumer asynchronously retrieves the message and writes it to the active time-series database (as configured via ACTIVE\_DB).
4. Simultaneously, the standby consumer stores the same data in a backup database for redundancy.
5. The display program queries the active DB every minute to retrieve real-time updates for the selected bed, with fallback logic to ensure continuity.

This modular design ensures that each component maintains clear responsibilities and decoupled communication, enabling easy fault tracing, testability, and future enhancements.

## Implementation Challenges and Decisions

The implementation phase included handling several complex but important problems, particularly resulting from the variety of the technologies included into the system. Important choices were taken to preserve modularity, adaptability, and performance all around the components.

### Database Driver Inconsistencies

The different ways that database customers behaved presented a major challenge. Used for Postgres and TimescaleDB, Psycopg2 requires strict SQL syntactic compliance, data type conversion, and explicit transaction control. On a different paradigm that gives time-aligned metrics top priority instead of relational integrity, InfluxDB's Python client runs on Flux or InfluxSQL. This required conditionally imported drivers and database-specific write logic paths, so adding to the complexity of the consumer module and guaranteeing best performance and compatibility for every backend.

### Timestamp Format Normalisation

Database ingesting and inter-system querying depend on consistent timestamp formatting. Particularly as messages crossed JSON serialisation, Kafka buffers, and time-series databases, inconsistencies in timezone recognition (naive versus aware datetime objects) and formatting styles (ISO 8601 against Unix Epoch) caused problems. Using ISO 8601 with UTC offsets, a project-wide protocol guaranteed all timestamps were standardised via Python's datetime and pytz modules before transmission.

### Environment Variable-Based Decoupling

Key configurations—such as the target database, Kafka topic, and message rate—were added as environment variables instead of hardcoded constants to guarantee loose coupling among services. This architectural decision enabled deployment in containerised environments, so making the system naturally more testable and customisable. By changing environmental settings, every service could be dynamically rearranged to enable quick transitions between testing environments without requiring code changes.

### Flexible JSON Payloads Without Schemas

To ensure loose coupling between services, key configurations—including the target database, Kafka topic, and message rate—were included as environment variables rather than hardcoded constants. This architectural choice made the system naturally more testable and customisable since it allowed deployment in containerised surroundings. Every service could be dynamically rearranged by changing environmental settings to allow rapid transitions between testing environments without needing code changes.

## Containerisation and Configuration

Docker was used all around to ensure portability, environmental consistency, and orchestration ease of use. Initially starting the Kafka broker, Zookeeper, and all-time-series database services, a Docker Compose configuration was developed to coordinate them. Facilitating isolated, repeatable installations across the development and testing stages required this multi-service architecture.

### Docker Compose Design

The main tool for arranging the required messaging infrastructure was the docker-compose.kafka.yml file. Defining services for Kafka (kafka-1) and Zookeeper (zookeeper-1), we included pre-defined environment variables including broker IDs and port bindings. Compose helped to initially start baseline\_sql, a Postgres database used for container benchmarking.While often started by terminal commands or managed manually via the Docker Desktop interface, other databases including InfluxDB, TimescaleDB, and VictoriaMetrics ran within containers. This hybrid approach allowed careful control all through the testing process and maintained the advantages of container isolation.

### Environment Variable Strategy

While the architecture enabled environment-specific overrides via Docker Compose's environment fields, the project did not use a.env file. This enabled topic names, database credentials, and Kafka port runtime configuration without requiring code changes or container reconstruction. Future efforts could capture these setups in a.env file to improve portability and maintainability across machines.

### Port Mapping and Volume Persistence

Strategic port mappings were set up to show the host system internal container services. For example, while InfluxDB was made available on port 8086, so assuring compatibility with local development tools and web-based dashboards, Kafka was set to map from 29092 (container) to localhost:29092. This made easy interaction with every service available from the host macOS environment possible.

Volume persistence was set in PostSQL (baseline\_sql) by means of named Docker volumes (baseline\_sql\_data), so preserving database state across container restarts. This was especially helpful during test orchestration, where consistency validation depends on data continuity yet many restarts are absolutely necessary.

# Testing

This part describes the methods, instruments, testing strategy, and results used to confirm system performance, resilience, and accuracy. To ensure complete coverage, we combined hand scenario testing with automatic benchmarking. Using a Docker-based architecture on a macOS system, all testing was carried out with methodical result tracking to CSV and log file transparency and repeatability.

### Test Strategy and Approach

The testing procedure was categorised into the following principal segments:

* Unit Testing: Essential scripts, including producer.py, consumerMaster.py, and database-specific writers, were independently verified to confirm proper functionality, encompassing Kafka message formatting, ingestion logic, and database interface. This was accomplished via direct observation and print/log verifications instead of a formal unit testing methodology.
* System Integration Testing: The entire pipeline was evaluated, commencing with preprocessed CSV input, proceeding through Kafka, into the target database, and optionally into the display interface. This confirmed that components behaved appropriately in real-time.
* Performance Benchmarking: A comprehensive test suite (run\_all\_tests.py) conducted across four databases (Baseline SQL, InfluxDB, TimescaleDB, VictoriaMetrics) at diverse ingestion rates ranging from 10 to 1000 MPM, emulating various stress levels.
* Manual Failover Testing: Simulated database container failures during operation to assess standby consumer behaviour and confirm that failover occurred without data loss.
* Display Program Testing: The display\_program.py (or comparable monitor) underwent manual validation by suspending containers and confirming accurate heartbeat loss detection and reporting.
* Resource Monitoring: The scripts (monitor\_resources.py and psutil) recorded both global system metrics and individual message resource utilisation to assess system efficiency and identify bottlenecks.

Bash and Python scripts let you automatically test orchestration, service initialisation, message rate control, and logging. Every test generated ordered logs in the logs/directory including memory/CPU trends, system snapshots, and write confirmations.

### Functional Testing

Functional testing ensured that each system component behaved as intended:

|  |  |
| --- | --- |
| **Component** | **Validation** |
| producer.py | Verified MPM throttling, Kafka connectivity, and correct JSON serialization |
| consumerMaster.py | Confirmed DB-specific write success, message deserialization, and timestamp parsing |
| consumerStandBy.py | Tested by stopping the master DB and observing successful fallback ingestion |
| display\_program.py | Verified live query responses and real-time graph updates per bed number |

These tests confirmed the modular integrity and functional completeness of the real-time data pipeline.

### Performance Testing

A screenshot of a computer

AI-generated content may be incorrect.

The performance benchmarking was driven by the run\_all\_tests.py script, which systematically tested each database backend across a set of predefined ingestion rates. The benchmarking matrix spanned:

* 4 Databases: Baseline SQL, InfluxDB, TimescaleDB, VictoriaMetrics
* 9 MPM Rates: 10, 20, 40, 60, 80, 100, 200, 500, 1000
* Total Scenarios: 36 combinations, each run for 300 seconds

For each test scenario, the following data was recorded:

* Global Resource Metrics: Collected via docker stats every 5 seconds by monitor\_resources.py
* Per-Message Resource Metrics: CPU/memory usage logged using psutil at each message emission
* Write Time Logs: Database writes timestamped to enable latency calculation and consistency checks

These tests verified that the system could sustain ingestion rates up to **1000 MPM** across all target databases. The logs serve as empirical performance evidence for ingestion latency, stability, and efficiency.

### Test Tools and Automation

Below is a summary of testing tools and scripts used:

|  |  |
| --- | --- |
| **Tool / Script** | **Purpose** |
| run\_all\_tests.py | Automates DB/container cycling, script orchestration, and test execution |
| monitor\_resources.py | Periodically captures container-level CPU and memory stats |
| psutil | Logs per-message resource usage from within Python |
| record\_system\_conditions | Records machine specs and battery status at test start |
| logs/ Directory | Stores global logs, per-message logs, DB write logs, and system snapshots |

Tests were entirely reproducible by rerunning the automation suite, ensuring consistent and isolated environments across all runs.

### Sample Test Evidence

All tests generated verifiable output saved under the structured logs/ directory. Sample artifacts include:

* System Info Snapshots (logs/system\_info/...): Platform info, CPU count, memory, and battery status
* Global Logs (logs/global/...): Time-series data of container resource usage
* Per-Message Logs (logs/per\_message/...): Timestamped metrics for each message
* Write Logs (logs/write\_logs/...): Confirmed heart rate DB insertions with exact timestamps

These logs collectively demonstrate:

* Real-time performance under varying load conditions
* End-to-end data integrity and throughput
* Efficient system resource usage

### Test Coverage and Limitations

Coverage Achievements:

* Full ingestion pipeline (CSV → Kafka → DB → UI)
* Benchmark testing across 36 configurations
* Manual failover validation
* Real-time heartbeat monitoring (manager/program\_monitor)
* Per-message and global system resource logging

Limitations:

* No formal unit test framework (e.g., pytest) used
* GUI testing was manual; no UI test automation
* Security, authentication, and encryption were out of scope

Future work could include integrating CI pipelines with automated assertions, enhanced failover simulation, and UI testing frameworks.

# System Evaluation and Experimental Results

## Evaluation Methodology

The basis of this analysis is the benchmarking strategy covered in Section 5. Four database systems—Baseline SQL, InfluxDB, TimescaleDB, and VictoriaMetrics—were experimentally assessed in a thorough performance analysis under controlled ingestion workloads. Especially in view of rigorous ICU monitoring conditions, the main objective was to evaluate the fit of every system for the real-time heart rate data storage and ingestion.  
The benchmarking approach tested every database at nine specified ingestion rates, generating 36 unique test scenarios from 10 to 1000 messages per minute (MPM). To guarantee consistency and comparability among all configurations, every test scenario was run for a set three hundred three hundred seconds. A multi-tiered logging system was used to document extensive performance data all through the assessment process.

* **Global container-level metrics** (e.g., CPU and memory usage) were collected every five seconds using the monitor\_resources.py script, which relies on Docker stats.
* **Fine-grained per-message resource data** was recorded using the psutil library, providing insight into the producer’s CPU and memory footprint during message emission.
* **Database write confirmations**, including exact timestamps and heart rate values, were logged per message to enable latency calculations and throughput analysis.

The overarching goals of this evaluation were threefold:

1. **Scalability Analysis** – To determine which databases maintain stable performance as ingestion rates increase.
2. **Resource Efficiency** – To compare CPU and memory usage across both producer and containerized environments.
3. **Ingestion Accuracy and Responsiveness** – To quantify latency, throughput, and any message delivery discrepancies that may impact real-time reliability.

As recorded in the system\_info\_summary.csv, all tests were carried out under exactly same hardware settings, so guaranteeing a fair and consistent basis for comparison. The approach creates a basis for informed recommendations in real-time medical monitoring systems and helps empirical results on system performance clear-cut.

## System Performance Results

Three main performance criteria—general container resource use, individual message producer efficiency, and database write performance—defined the assessment of the system. After 36 benchmarking events, the data were examined to highlight among the four databases relative strengths and trade-offs.

### Global CPU and Memory Usage (Container-Wide)

CPU and memory measurements at the container level were documented using monitor\_resources.py every five seconds during the 300-second test period. These measures provide insight into the whole system footprint of each time-series database as ingestion rates escalate.

A graph of a graph with different colored lines

AI-generated content may be incorrect.

A graph with different colored lines

AI-generated content may be incorrect.

1. **CPU Observations:**

* **InfluxDB regularly exhibited the highest CPU utilisation, especially at elevated ingestion rates. This indicates heightened internal processing or increased network handling overhead.**
* **VictoriaMetrics had a consistent CPU profile, scaling predictably with increased demand while demonstrating superior efficiency compared to InfluxDB across the majority of tiers.**
* **TimescaleDB exhibited modest yet stable CPU utilisation, marginally surpassing VictoriaMetrics while remaining beneath InfluxDB.**
* **The baseline SQL had the lowest overall CPU usage, however this was partly attributable to reduced write efficiency rather than optimisation.**

1. **Memory Observations:**

* **InfluxDB once more distinguished itself, utilising up to 195MB as ingestion rates escalated. This trend indicates internal buffering or caching operations, consistent with its emphasis on intake velocity.**
* **VictoriaMetrics demonstrated slight memory expansion, reaching a maximum of approximately 125MB.**
* **TimescaleDB demonstrated superior memory stability, consistently maintaining approximately 77MB, signifying a more efficient memory footprint—beneficial in resource-limited settings.**
* **The baseline SQL had a consistent size of 22MB, possibly due to its reduced write rate and lack of time-series enhancements.**

1. **Conclusion:**

Trade-offs are clear: InfluxDB suffers a rather high system cost even if it achieves high throughput. On the other hand, VictoriaMetrics and TimescaleDB provide better resource efficiency, which could help production settings to scale more easily.

### Producer Resource Usage (Per-Message)

Using psutil, producer.py's performance was evaluated per-message under CPU and memory consumption recording. This statistic separates the costs of serialising, formatting, and sending heart rate records to Kafka, which is greatly impacted by the effectiveness of the Kafka client of the relevant database and any upstream buffering systems.

A graph with lines and numbers

AI-generated content may be incorrect.

A graph with colored lines

AI-generated content may be incorrect.

1. **CPU Usage (Producer-Side):**

* All producers exhibited a linear increase in CPU% as the MPM escalated.
* **VictoriaMetrics constantly necessitated the minimal CPU per message, signifying an efficient serialization-to-publication process.**
* **InfluxDB closely followed but shown significantly greater fluctuation, perhaps because to its more intricate HTTP-based client.**
* **Both Baseline SQL and TimescaleDB exhibited commendable performance, however they were slightly less efficient at elevated ingestion rates.**

1. **Memory Usage (Producer-Side):**

* All configurations demonstrated consistent memory performance following the initial surge.
* **InfluxDB exhibited somewhat greater memory variability, presumably attributable to its own batching processes.**
* **Baseline, VictoriaMetrics, and TimescaleDB had a uniform profile (~114–125MB) over the majority of test rates.**
  1. **Conclusion:**

From the producer's perspective, VictoriaMetrics shows improved efficiency using less CPU and memory resources per message—perfect for edge deployments or systems with limited computational capacity.

### Latency and Throughput

Evaluation mostly focused on write throughput—that is, successful database writes per second—and latency—that is, the time from Kafka message reception to database confirmation. From all testing environments, these measures were taken from write-logs.

A graph with a red line

AI-generated content may be incorrect.

A graph with a line and a red line

AI-generated content may be incorrect.

1. **Write Latency:**

* **All time-series databases (InfluxDB, TimescaleDB, VictoriaMetrics) demonstrated reduced latency as MPM grew, leveraging internal batching and write optimisation techniques.**
* **The baseline SQL exhibited disproportionately elevated latency for low MPM values, at times over 5000ms. This is due to its absence of native capability for real-time time-series ingestion.**

1. **Write Throughput:**

* All databases exhibited linear scalability with MPM, although InfluxDB sustained the highest write throughput at peak rates, nearing 17 writes per second.
* **TimescaleDB and VictoriaMetrics shortly followed, both managing elevated write rates without reaching saturation.**
* **The baseline SQL, albeit operational, demonstrated the lowest throughput, underscoring its inadequacy for high-volume, real-time applications.**

1. **Conclusion:**

Regarding write throughput and ingest latency, time-series databases clearly outperform general-purpose SQL databases. While TimescaleDB and VictoriaMetrics offer more fair trade-offs with lower system costs, InfluxDB has better raw performance.

## Comparative Analysis

The results of the benchmarking studies show among the evaluated databases unambiguous strengths and trade-offs. Based on latency, throughput, resource economy, and general ingestion stability, every solution shows fit for particular deployment scenarios.  
The table below captures the highlights of relative performance:

|  |  |
| --- | --- |
| **Metric** | **Best Performance** |
| Lowest latency | Time-series DBs (InfluxDB, TimescaleDB, VictoriaMetrics) |
| Highest throughput | InfluxDB |
| Lowest producer CPU | VictoriaMetrics |
| Lowest container memory | TimescaleDB |
| Best message accuracy | InfluxDB |
| Highest latency | Baseline SQL |

InfluxDB regularly beats rivals in high-throughput environments. Its HTTP-based ingestion engine and optimised client library help it to maintain both speed and accuracy under pressure. It remains the preferred choice for real-time data flows where speed is crucial despite its higher memory and CPU demands.

TimescaleDB shows amazing consistency in memory use over several consumption rates, in contrast. Particularly on resource-limited devices, its link with Postgres makes it a sensible choice for systems that stress schema management and compatibility with existing SQL frameworks.  
  
VictoriaMetrics presents a convincing compromise. It scales effectively under high message rates, offers consistent performance with reasonable resource use, and has low CPU overhead both during production and consumption. These qualities make it a strong competitor for systems that call for a compromise between scalability and efficiency.

Baseline SQL showed the lowest performance overall, marked by high write latencies and low throughput on every level. Although operational, it is not fit for real-time telemetry applications since it lacks time-series ingestion optimisations.The database selected in a real-time Kafka pipeline depends on deployment goals. InfluxDB is the best option if accuracy and speed are absolutely vital. TimescaleDB or VictoriaMetrics are more suited if memory stability and efficiency are of great importance. This comparative paradigm helps to enable deliberate decisions based on operational priorities and deployment constraints.

## System Information Impact

Beginning each test using the system\_info\_summary.csv dataset, system-level diagnostics were recorded to ensure the validity and repeatability of the performance results. Essential features of the host environment were captured in this file, so providing a contextual basis for understanding resource statistics.

The table below shows the relative importance of every statistic:

|  |  |
| --- | --- |
| **Metric** | **Why It Matters** |
| CPU Usage % | Indicates available headroom during test |
| Memory Usage % | Confirms no system-level bottlenecks |
| Battery Plugged In | Confirms no power throttling on macOS |
| CPU Cores | Validates fair test ground (10 cores used) |

The hardware setup was consistent throughout all testing conditions: a macOS 15.3.2 system including 10 CPU cores and 16 GB of RAM, so ensuring a constant execution environment. Importantly, the system stayed linked to a power source, which under macOS prevents background CPU throttling for battery saving needs. This element is particularly crucial since Apple silicon chips dynamically change performance without a power source connected.  
  
Each test started with CPU usage roughly 33% and memory consumption roughly 56%. These figures show that no competing system-level operations disturbed the measurements and that the machine operated with enough resources at hand. Therefore, rather than to the limitations of the underlying host, the recorded CPU and memory usage metrics from the container and producer levels can be mostly ascribed to the behaviour of the database systems.  
  
Hardware's consistency and consistent power configuration helped to create a fair and under control experimental environment. By removing outside uncertainty and allowing a concentration on internal variations among the database backends, this improved confidence in the comparison analysis.

## Functional Observations (Manual Tests)

Two basic system components underwent hand functional testing to assess their accuracy, responsiveness, and robustness under failure scenarios in addition to automatic performance benchmarking. The components comprised the consumer failover system and the Display Program.

### **Display Program Validation**

The real-time display interface was evaluated by conducting user-initiated queries for designated bed numbers and monitoring system performance during live data import. The subsequent principal observations were documented:

* **Accurate Data Retrieval:** The application reliably retrieved the most recent heart rate data associated with the chosen bed, validating effective database querying and response management.
* **Real-Time Graphing:** Visualisations were refreshed about every 60 seconds, in accordance with the consumer-to-UI data refresh protocol. Each graph precisely represented temporal variations in patient vitals without delay or interruptions.
* **Stability:** No data loss or rendering delays were observed during prolonged usage windows (exceeding 5 minutes per test run), confirming robust communication between the display program and the active database backend.

### **Failover Test: Master-to-Standby Transition**

A failover simulation was conducted to assess system fault tolerance by manually halting the master database container during an active ingestion cycle. The examination unfolded in the subsequent manner:

1. The consumerMaster.py process was actively receiving Kafka messages and recording them in the master database.
2. During the test, the master container was suspended using Docker to emulate unavailability.
3. The standby consumer (consumerStandBy.py) identified the master's failure and promptly began ingesting without any message loss.
4. Kafka delivery acknowledgements (ACKs) verified the successful receipt and processing of messages by the standby consumer.

Validation was further supported through:

* **Log comparison**: The confirmation logs from both the master and standby consumers demonstrated consistency in timestamps, bed numbers, and heart rate values, signifying the absence of skipped or duplicated messages.
* **Kafka integrity**: The message offset stream persisted without interruption, guaranteeing that data was preserved and processed sequentially.

An annotated excerpt from the consumerStandBy.py log is provided in the appendix to demonstrate the activation of fallback ingestion and the seamless continuity of writing.

Collectively, these manual tests validate the operational integrity and robustness of the system's interaction and fault-handling components. Despite lacking automation, their performance under realistic usage and failure scenarios enhances the overall resilience of the architecture.

## Limitations and Future Work

Although the system met its main objectives of real-time heart rate intake, fault tolerance, and database performance benchmarking, some restrictions were noted, suggesting areas for improvement and more research.

### Identified Limitations

* **Absence of Formal Unit Testing**:

Despite the human testing of the system's components and connected functional flows, an automated unit testing framework, such as pytest, was not established. Consequently, minor regression concerns may remain unnoticed during updates or refactoring.

* **Security Not Addressed**:

Due to the project's emphasis on academic and infrastructure aspects, security measures including authentication, encryption, and access control were not executed. This deficiency renders the system inappropriate for implementation in sensitive production settings (e.g., hospitals) without substantial enhancement.

* **Manual GUI Validation**:

The display program was confirmed solely through manual testing. No formal interface testing methods, such as Selenium or PyAutoGUI, were utilised to assess the reliability or robustness of graphical interactions in edge-case scenarios.

### Future Enhancements

To enhance the system's resilience, observability, and deployment readiness, the following improvements are suggested:

* **Continuous Integration (CI)**:

Introduce a CI pipeline (e.g., via GitLab CI/CD or GitHub Actions) that performs **automated unit and integration tests** for each code push. This would improve development agility and safeguard against regressions.

* **Advanced Performance Metrics**:

The existing benchmarking configuration encompasses a broad spectrum of metrics, although it might be augmented to incorporate:

* + **End-to-End Latency**: Quantifying the duration from message generation to verified database entry.
  + **Kafka Consumer Lag**: Monitoring backlog to pinpoint performance impediments.
  + **Disk I/O Statistics**: Especially advantageous for storage-demanding databases such as TimescaleDB.
  + **Time to Steady State**: Documenting the dynamics of system stabilisation under load variations.
* **Schema Validation for JSON Messages**:

The system can now accept flexible JSON formats, which speeds development but could cause incompatibilities. Using Avro/Protobuf alongside Kafka Schema Registry or enforcing JSON Schema would increase downstream compatibility and message integrity.

* **Security Layer Integration**:

Later iterations might feature basic role-based access control (RBAC) for the display and management modules, TLS for Kafka communication, API tokens or OAuth for database endpoints. These features would help to distribute sensitive medical data in practical settings.

* **Scalability Testing in Distributed Environments**:

Every benchmark ran under a single-machine macOS environment. Testing multi-node distributed installations and maybe using Kubernetes would help to validate the system's performance under real operating conditions.

## Societal and Commercial Implications

Particularly in settings requiring real-time physiological monitoring, the system developed in this study has major ramifications for both healthcare and the more general technological sectors. Designed on modularity, fault tolerance, and time-series optimisation, the architecture is flexible enough to expand and fit many real-world use.

### Healthcare Applications and Societal Value

The system's principal application—real-time heart rate monitoring in intensive care units (ICUs)—can directly enhance patient outcomes. The platform can assist medical personnel by providing low-latency, continuously updated data:

* **Early detection of arrhythmic patterns or anomalies**
* **Rapid response to sudden physiological changes**
* **Data-driven post-event analysis for clinical audits**

In high-dependency wards, where quick access to patient vitals can be quite important, these features are especially helpful. TimescaleDB's consistent memory use shows that the system's ability to run efficiently on low hardware improves its accessibility in resource-limited hospitals.

### Scalability and Reusability

Outside its ICU application, the system's containerised, Kafka-based architecture facilitates hospital-wide implementations, allowing for centralised oversight across departments or several sites. The components may also be adjusted for:

* **Wearable health trackers** in fitness and wellness sectors
* **Athlete monitoring systems** during training or rehabilitation
* **Remote health diagnostics** in home-care environments

The capability to process and visualise physiological signals in real time with a latency of under 200ms (at rates of up to 1000 messages per minute) establishes the system as a suitable backend for real-time dashboards and alerting platforms.

### Commercial Opportunities

Built with flexible Python modules and open-source technologies, the architecture might serve as a model for commercial SaaS solutions in elder-care, sports science, or medical telemetry. Its market readiness would be much improved by integration with cloud-based time-series databases and authentication systems.

Startups or academic spinouts could utilise this system to develop minimal viable products (MVPs), providing:

* **Real-time vitals aggregation and dashboarding**
* **Cross-device synchronization for health IoT platforms**
* **Predictive analytics pipelines for long-term heart rate trends**

### Risks and Ethical Considerations

Despite its technical success, several caveats must be noted before clinical deployment:

* **Data Privacy and Security**:

The existing prototype lacks encryption, authentication, and access controls. These shortcomings must be rectified in any medical-grade application, considering the sensitivity of physiological data and the potential for data breaches.

* **Operational Risk**:

Although backup options were evaluated, a more stringent high-availability configuration (e.g., clustered Kafka, replicated database nodes) is necessary to guarantee zero downtime in essential healthcare environments.

### Feasibility and Impact

The results of performance testing show that in real-time environments the system is both practically feasible and theoretically strong. Given little engineering effort, it could be turned into a production-grade instrument that:

* Improves **operational efficiency** for clinicians
* Enhances **data availability** for research and diagnostics
* Promotes **equitable access** through cost-effective deployment options

In summary, although additional work is required for clinical-grade implementation, the project provides a robust technological basis with significant potential to influence patient care, research, and commercial innovation.

# Appendices

Appendices will not be marked but may be referred to by the assessor to aid their understanding. They are useful if there is something that helps in understanding earlier parts of the dissertation, but if included inline might break the flow or readability of the document. For example, there may be large tables of data, design documents, evidence of testing etc etc.

# References

[1] K. S. Alang and A. S. Kushwaha, "Stream Processing with Apache Kafka: Real-Time Data Pipelines," \*Int. J. Res. Mod. Eng. Emerg. Technol.\*, vol. 13, no. 3, pp. 45–52, Mar. 2025. doi: 10.63345/ijrmeet.org.v13.i3.13.

[2] H. Ni, S. Meng, X. Geng, P. Li, Z. Li, X. Chen, X. Wang, and S. Zhang, "Time Series Modeling for Heart Rate Prediction: From ARIMA to Transformers," in \*Proc. 2024 6th Int. Conf. Electron. Eng. Inform. (EEI)\*, 2024, pp. 584–589. doi: 10.48550/arXiv.2406.12199.

[3] Apache Kafka. [Online]. Available: https://kafka.apache.org/

[4] Docker Inc., “Docker Documentation.” [Online]. Available: https://docs.docker.com/

[5] InfluxData, “InfluxDB Documentation.” [Online]. Available: https://docs.influxdata.com/influxdb/

[6] Timescale, “TimescaleDB Documentation.” [Online]. Available: https://docs.timescale.com/

[7] VictoriaMetrics, “Time Series Database.” [Online]. Available: https://docs.victoriametrics.com/

[8] G. Rodola, “psutil – process and system utilities.” [Online]. Available: https://psutil.readthedocs.io/

[9] Confluent, “confluent-kafka Python Client.” [Online]. Available: https://github.com/confluentinc/confluent-kafka-python

[10] Prometheus Authors, “Prometheus: Monitoring System & Time Series Database.” [Online]. Available: https://prometheus.io/

[11] Python Software Foundation, “Tkinter GUI Programming.” [Online]. Available: https://docs.python.org/3/library/tkinter.html

[12] J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” \*Computing in Science & Engineering\*, vol. 9, no. 3, pp. 90–95, 2007. doi: 10.1109/MCSE.2007.55.