RE-ARCHITECTING 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

28th April 2025

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.
6. 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* 28/04/2025

Acknowledgements

I would like to express my sincere gratitude to my supervisor, **Dr. Charles Gillan**, for his invaluable guidance, encouragement, and continuous support throughout the development of this project and the preparation of this dissertation.

I am also grateful to my family and friends for their support and encouragement during my studies.

Abstract

This work presents the development of a real-time heart rate monitoring system for ICU conditions using Apache Kafka and numerous time series databases (TSDBs). The system generates synthetic heart rate data, transports it across a Kafka channel, and stores it among several TSDB servers using a master-standby consumer architecture with automatic heartbeat-based failover. Designed to provide patient heart rates in real-time, historical data retrieval, and system status monitoring, this Django online dashboard is designed to give resilience first attention, allowing standby consumers to immediately promote themselves in the case of master failure, therefore ensuring continuous data flow and availability. Extensive testing covering system integration, performance benchmarking across several TSDBs, failover and recovery operations, and user interface capability. By means of consistent real-time monitoring with seamless failover capabilities, the system reveals a scalable and fault-tolerant architecture appropriate for major healthcare needs.

Table of Contents

Table of Figures and Tables

1. Introduction and Problem Area

1.1. 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 an elevated frequency. This presents data engineering issues concerning dependable ingestion, transformation, and low-latency access to time-stamped data [1]. This project builds upon prior work by Dr. Charles Gillan and his team, who developed an initial ICU telemetry system [2], and extends it through enhanced real-time ingestion, failover resilience, and comparative database benchmarking.

1.2. Problem Context

The specific requirements of ICU telemetry, where data accuracy and promptness are essential, make conventional relational databases insufficient for high-frequency streaming input. Time-series databases (TSDBs) as InfluxDB, TimescaleDB, and VictoriaMetrics, specifically designed for temporal data, provide solutions by enhancing write throughput, compression, and retrieval speed [4], [5]. Apache Kafka, a distributed event streaming platform, offers the decoupled, high-throughput infrastructure essential for dependable ingestion pipelines [3]. Nonetheless, architectural obstacles persist in achieving real-time performance, failover resilience, and minimal latency amid variable message-per-minute (MPM) loads. This study seeks to analyse and evaluate various TSDB alternatives within a fault-tolerant Kafka pipeline to determine the best appropriate solution for future ICU systems.

1.3. 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. As part of the baseline comparison, PostgreSQL was selected based on prior work by Gillan et al. [6], where PostgreSQL was adopted for ICU telemetry systems due to its reliability, widespread adoption, and proven consistency for structured medical data. 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 [3].

1.4. 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 is performed locally in a macOS environment utilising 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 optimisation [3].

2. System Requirements and Specifications

2.1. Overview

Multiple time-series database (TSDB) technologies are tested in this system for real-time ICU heart rate monitoring. Apache Kafka is the main message bus that lets data-producing and consuming components communicate decoupled in scale. The system includes fault-tolerant capabilities, including heartbeat emission and monitoring, automated promotion of backup consumers, and fallback querying across several database instances; it also simulates a real-world telemetry intake pipeline. Under regulated data loads, the major objectives are to test the ingestion performance of different TSDBs—InfluxDB, TimescaleDB, VictoriaMetrics, and a baseline Postgres setup to show a resilient system architecture that can maintain real-time monitoring during consumer failures. On a Django dashboard are live visuals, history searches, and system status updates. Perfect failover and recovery allow the modular, repeatable solution to replicate mission-critical telemetry systems.

2.2. Assumptions and Constraints

- All development and testing were conducted on a macOS M1 device equipped with a 10-core CPU and 16 GB of RAM.

- Docker containers were utilised for all services, including Kafka, Zookeeper, and each database instance.
- Each test setup operated for 300 seconds (5 minutes) at each configured message load tier.
- The system includes a Django-based web dashboard for real-time data visualisation and system status monitoring.
- Heartbeat mechanisms were implemented for consumer liveness detection and automatic standby promotion.
- Standby consumers actively monitored the master heartbeat and performed automatic promotion based on failure detection.
- Fallback querying logic was incorporated in the Django dashboard to ensure data availability across multiple TSDB instances.
- Resource utilisation metrics (CPU, memory) were collected during system operation to assess efficiency.
- The final system is capable of automatic failover, live monitoring, historical querying, and live performance analysis without manual intervention.

2.3. 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 write heart rate data into the active time-series database.
FR4	The designated database will retain heart rate data with associated timestamps and bed identifiers.
FR5	The system shall support multiple database backends: InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL.
FR6	The system shall record CPU and memory utilisation for each database container during tests.
FR7	The system shall measure and record write latency for each ingested message.

FR8	The system shall support automated performance benchmarking across multiple load tiers.
FR9	The system shall generate .csv logs for resource metrics, individual message logs, and write latencies.
FR10	The system shall support configuration through environment variables (e.g., ACTIVE_DB, MESSAGES_PER_MINUTE).
FR11	The display program shall enable users to monitor real-time heart rate data by bed number through a Django web dashboard, it also shall support fallback querying across standby databases if the primary database is unavailable.
FR12	Standby consumers shall monitor master heartbeats and automatically promote themselves if the master consumer fails.
FR13	Standby consumers, upon promotion, shall write heart rate data to all available time-series database instances (master and standbys).
FR14	The system shall ensure that after master recovery, promoted standby consumers voluntarily demote themselves and return to passive monitoring.

Table 1. Functional Requirements

2.4. Non-Functional Requirements

ID	NFR	Target / Status
NFR1	System must process up to 1000 messages per minute reliably.	Achieved
NFR2	CPU usage per database container should remain under 60% at 500 MPM.	Exceeded by InfluxDB; Met by others
NFR3	Average database write latency should stay under 150ms.	Met by all Time-Series Databases
NFR4	Memory usage per container should remain under 200MB.	Achieved
NFR5	Failover from master to standby consumer must occur within 10 seconds.	Achieved
NFR6	Promoted standby consumers must voluntarily demote upon master recovery.	Achieved

NFR7	All system components must run in Docker for consistent, reproducible deployment.	Achieved
NFR8	Each test run must auto-generate exportable .csv logs for performance analysis.	Achieved
NFR9	System must enable fallback querying across standby databases for high availability.	Achieved
NFR10	System should be extensible to support additional databases, output visualizations, and enhanced monitoring features.	Structure allows for extension

Table 2. Non-Functional Requirements

2.5. Use Case Definitions

Use Case	Description
UC1	Data Ingestion Simulation – Streams ICU heart rate data to Kafka at configurable message rates from a CSV source
UC2	Multi-Database Ingestion – Active consumer writes ingested data simultaneously to multiple time-series databases (InfluxDB, TimescaleDB, VictoriaMetrics).
UC3	Heartbeat Monitoring & Failover – Master consumer emits heartbeats; standby consumers monitor and automatically promote themselves upon master failure.
UC4	Real-Time Data Query Interface – Django-based dashboard allows users to query and visualize real-time and historical heart rate data by bed number, with fallback querying across standby databases if needed.
UC5	Performance Benchmarking and Logging – System automatically records CPU usage, memory usage, message latencies, throughput statistics, and exports detailed .csv performance logs.

Table 3. Use Case Definitions

2.6. Software Specification

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

- *main.py*: Master launcher script that initialises the whole system stack, including Kafka services, producers, consumers, heartbeat emitters, and the Django dashboard.

- *producer.py*: Streams heart rate records from hr.csv into Kafka at a configurable message rate.
- *consumerMaster.py, consumerStandby1.py, consumerStandby2.py*: Consumers that write incoming Kafka messages into multiple active time-series databases. Standbys are capable of automatic promotion upon master failure.
- *heartbeat_emitter.py*: Sends heartbeat messages from active consumers for system liveness detection.
- *monitor_resources.py*: Captures containerised CPU and memory usage using Docker statistics.
- *run_all_tests.py*: Orchestrates automated test cycles across different databases and message rates.
- *Kafka and Zookeeper*: Provisioned via Docker Compose, managing heart_rate and heartbeat topics.
- *Data Input*: hr.csv, containing over 700,000 timestamped heart rate entries.
- *Web Dashboard (Django)*: Provides live visualisation and system monitoring with fallback querying.

Logging and Output:

- *global_summary.csv*: Aggregated system metrics per test configuration.
- *per_message_summary.csv*: Detailed latency records per message.
- *write_summary.csv*: Write success rates and observed ingestion gaps.

All system modules were implemented in Python, fully containerised with Docker, and optionally monitored using Prometheus and Django dashboards for real-time system observation.

2.7. System Interfaces

Interface	Description
Kafka Producer API	Used in producer.py to publish messages to the heart_rate topic.
Kafka Consumer API	Used in consumerMaster.py, consumerStandby1.py, and consumerStandby2.py to consume heart rate and heartbeat messages.
InfluxDB API	Accessed via influxdb-client (Python SDK) for heart rate data ingestion and querying.

TimescaleDB Interface	Accessed via psycopg2 library using PostgreSQL SQL statements for data insertion.
VictoriaMetrics API	Accessed via HTTP POST requests for direct metric ingestion.
PostgreSQL Interface	Accessed using psycopg2 standard SQL operations for baseline testing.
Django Web Interface	Real-time ICU monitor and system manager dashboard built with Django framework, accessible via browser.
Internal Environment Variables	Variables like ACTIVE_DB, MESSAGES_PER_MINUTE, IS_TEST_MODE, PROM_PORT, CONSUMER_ROLE, and others control system behavior dynamically.
Output Logs	global_summary.csv, per_message_summary.csv, and write_summary.csv exported for performance analysis.

Table 4. System Interfaces

2.8. User Characteristics

The intended users are technical researchers, system engineers, or performance testers who possess foundational knowledge in:

- Python scripting and environment management.
- Docker containerisation and orchestration.
- Kafka messaging system and topic configuration.
- Web application interaction (Django-based dashboard).

Users primarily interact with the system through:

- Terminal commands (launching main.py, Docker management, etc.).
- Environment variable configuration for dynamic setup adjustments.
- Monitoring real-time system status via the Django web dashboard.
- Analysing system behaviour through .csv log files.

No domain-specific medical expertise is required, as the system operates entirely on pre-recorded, anonymised ICU heart rate data.

3. Design

3.1. Architectural Overview

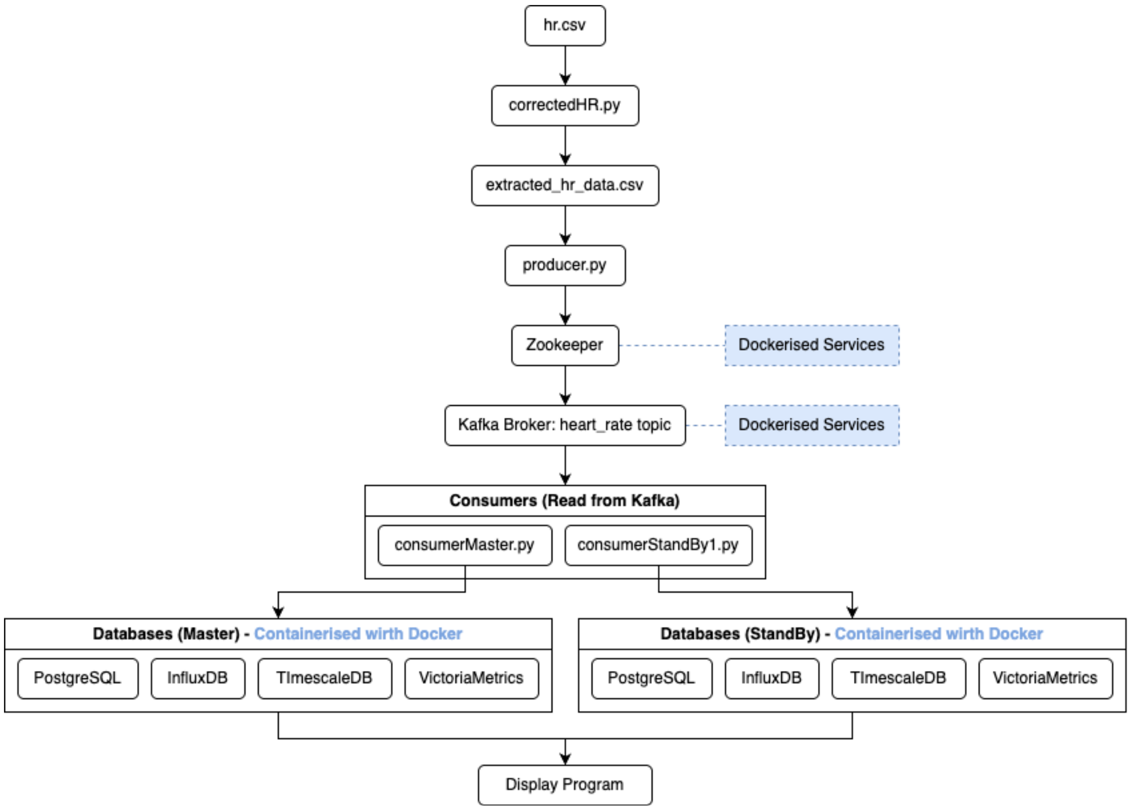


Figure 1. Data Flow Diagram

The system architecture emulates a real-world ICU telemetry pipeline, highlighting modularity, fault tolerance, and scalability. Apache Kafka functions as the primary message channel, separating producers from consumers. Docker Compose facilitates the orchestration of service deployment, guaranteeing reproducibility. The design is structured into five layers:

- Preprocessing Layer
- Streaming Ingestion Layer
- Message Bus and Decoupling Layer
- Consumer and Storage Layer
- Visualisation Layer

Each layer is independently testable, fault-tolerant, and extensible for future improvements like additional databases or cloud deployments.

3.2. Data Flow and Sequence of Operations

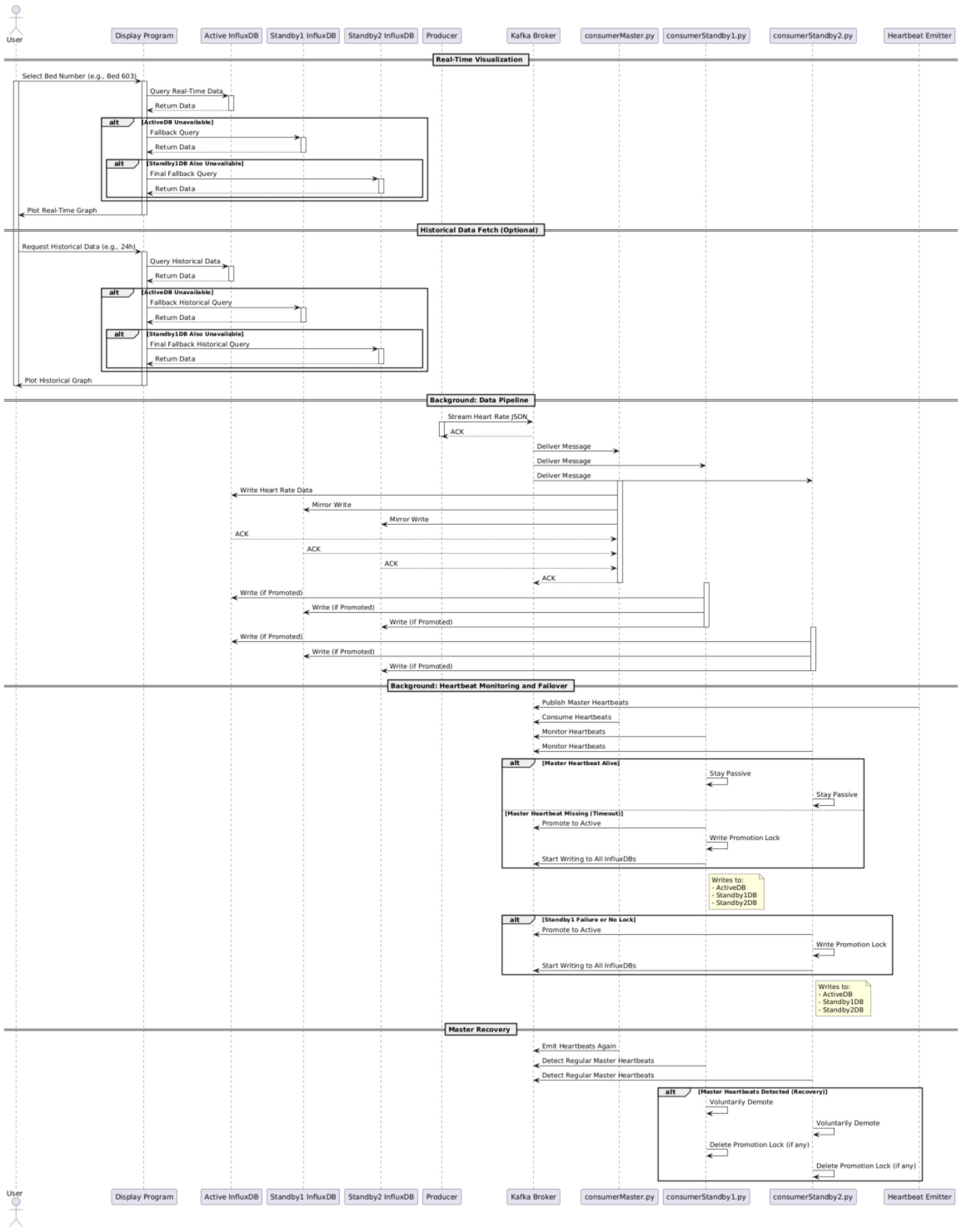


Figure 2. System Sequence Diagram

This section covers the entire operation of a real-time heart rate telemetry system. The system supports heartbeat-based failover, background data intake, and real-time and historical data searches. The operational flow begins with pre-recorded CSV data prior to preprocessing, streaming, and the live web-based dashboard for visualisation and monitoring.

3.2.1. Pre-processing Layer

The unprocessed ICU telemetry dataset, supplied as hr.csv, exhibited inconsistent formatting and incorporated information. A preprocessing script, correctedHR.py, was created to extract pertinent fields (bed number and heart rate), resulting in a refined extracted_hr_data.csv file. This clear delineation guarantees that upstream streaming modules consistently receive well-structured, predictable data without reprocessing the original source.

3.2.2. Streaming Ingestion via Kafka

The producer.py module transmits structured JSON messages to the Kafka heart_rate topic at adjustable rates between 10 and 1000 messages per minute (MPM). Every message contains a bed_number, heart_rate, and a timestamp consistent with ISO 8601. Kafka offers resilient, high-capacity, and asynchronous message transmission, facilitating the separation of data creation from consumption layers.

3.2.3. Message Bus and Decoupling

Kafka brokers consistently cache incoming heart rate data, ensuring availability despite customer interruptions. Apache ZooKeeper facilitates broker discovery and manages leadership. Docker Compose facilitates the containerised deployment of Kafka and ZooKeeper services, guaranteeing consistent endpoint accessibility and reproducibility of local orchestration.

3.2.4. Consumer and Storage Layer

The ingestion backbone comprises a master-standby consumer architecture to ensure fault tolerance and high availability:

- consumerMaster.py: Actively consumes messages from Kafka and writes ingested heart rate data into all configured database backends.
- consumerStandby1.py and consumerStandby2.py: Operate in passive standby mode, continuously monitoring the master's heartbeat signals on a dedicated Kafka topic. Upon detection of a heartbeat timeout, a standby consumer autonomously promotes itself to active ingestion mode, preserving system continuity.

Heartbeat emission and monitoring are conducted entirely within the Kafka infrastructure, eliminating external health checks. Prometheus metrics endpoints expose ingestion throughput, latency, resource utilisation, and failover event counts, enabling real-time system observability. This autonomous failover mechanism ensures seamless ingestion without human intervention, maintaining consistent telemetry availability under failure conditions.

3.2.5. Visualisation Layer

A Django-based dashboard serves as the front-end monitoring interface, offering both real-time and historical heart rate visualisations:

- Real-Time Graphs: Data is retrieved from the active database and refreshed every five seconds, presenting continuous heart rate trends per selected ICU bed.
- Automatic Fallback Querying: If the primary database is unavailable, the system transparently retries querying standby databases in a hierarchical order, ensuring data accessibility.
- Historical Queries: Users may retrieve aggregated telemetry trends over specified intervals (1h, 6h, 12h, 24h) for retrospective analysis.

Time-series visualisation is dynamically rendered via Chart.js, providing responsive, multi-client, and low-latency access suitable for critical care environments.

3.3. Component Design

This part describes the fundamental modules that enable real-time heart rate ingestion, processing, and monitoring in an ICU-like setting. Designed with modularity, fault tolerance, and scalability in mind, every element supports high availability and smooth failover. The system uses containerising with Docker, asynchronous messaging via Kafka, and several time-series database backends to provide consistent, low-latency performance under varied load situations.

3.3.1. correctedHR.py - Preprocessing Module

Raw ICU telemetry data (hr.csv) is converted in the preprocessing module into a format fit for real-time intake. The original data includes irregular formatting and associated metadata that could impede streaming operations. correctedHR.py uses regular expression matching to extract just the necessary fields—especially the heart rate values and bed number. Malformed or incomplete entries are automatically excluded. Extracted_hr_data.csv stores the cleaned output, creating a lightweight, normalised dataset directly available for the streaming producer.

3.3.2. producer.py – Kafka Producer

The producer module sends ICU heart rate data to a Kafka topic at a configurable frequency—messages per minute. Using the Confluent Kafka Python client, it first reads from extracted_hr_data.csv, converts each record into a structured JSON object, and 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 is controlled via the MESSAGES_PER_MINUTE environment variable.
- JSON serialisation 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.

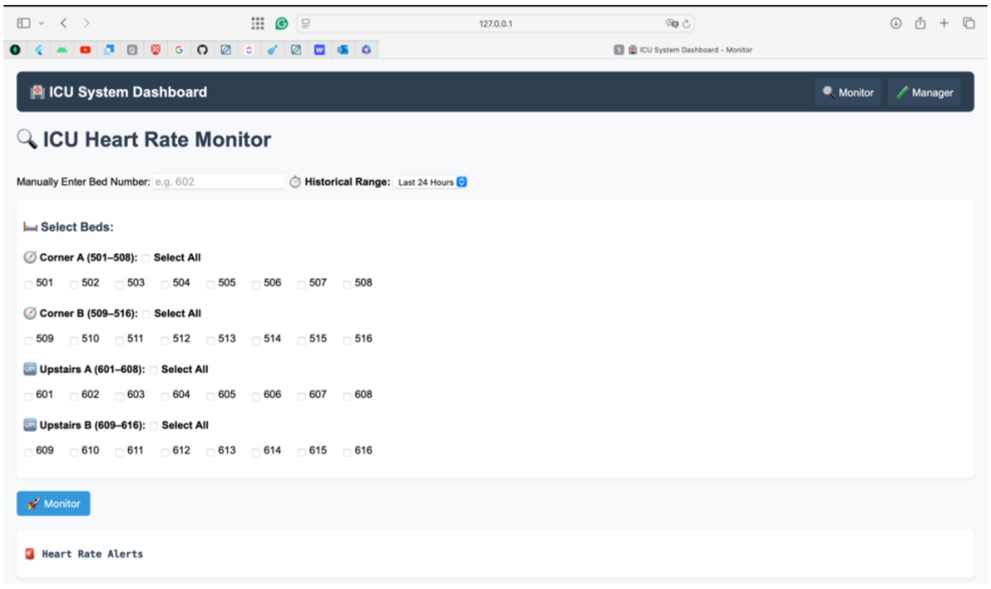
3.3.3. consumerMaster.py – Primary Kafka Consumer

The master consumer writes arriving messages to all configured time-series databases: Influxdb, Timescaledb, VictoriaMetrics, and Postgresql, following the heart_rate topic. Dynamic selection of the target databases is accomplished using the ACTIVE_DB environment variable; write routines are modularised for fit with each storage backend. Prometheus metrics—which track important factors including customer lag, processed message counts, and error rates—are made public. Comprehensive logging records write operation results for evaluation use; ISO-8601 timestamp validation guarantees temporal accuracy.

3.3.4. consumerStandBy1.py and consumerStandby2.py – Standby Consumers

The standby consumer modules (consumerStandby1.py and consumerStandby2.py) passively monitor Kafka heartbeats and keep them inactive until a master failure is found. When a standby consumer detects the lack of master heartbeats beyond a specified threshold, it starts writing heart rate data to all database instances, promotes itself to active status, and takes over consumption chores. Post-promotion, heartbeat emitters keep running; if the original master consumer recovers, voluntary demotion happens naturally. The standby design guarantees excellent system availability and fault tolerance that is free from human involvement.

3.3.5. Django Web Dashboard – Display and Visualisation Program


ICU System Dashboard

Monitor | Manager

ICU Heart Rate Monitor

Manually Enter Bed Number: e.g. 602 Historical Range: Last 24 Hours

Select Beds:

Corner A (501–508): Select All

501 502 503 504 505 506 507 508

Corner B (509–516): Select All

509 510 511 512 513 514 515 516

Upstairs A (601–608): Select All

601 602 603 604 605 606 607 608

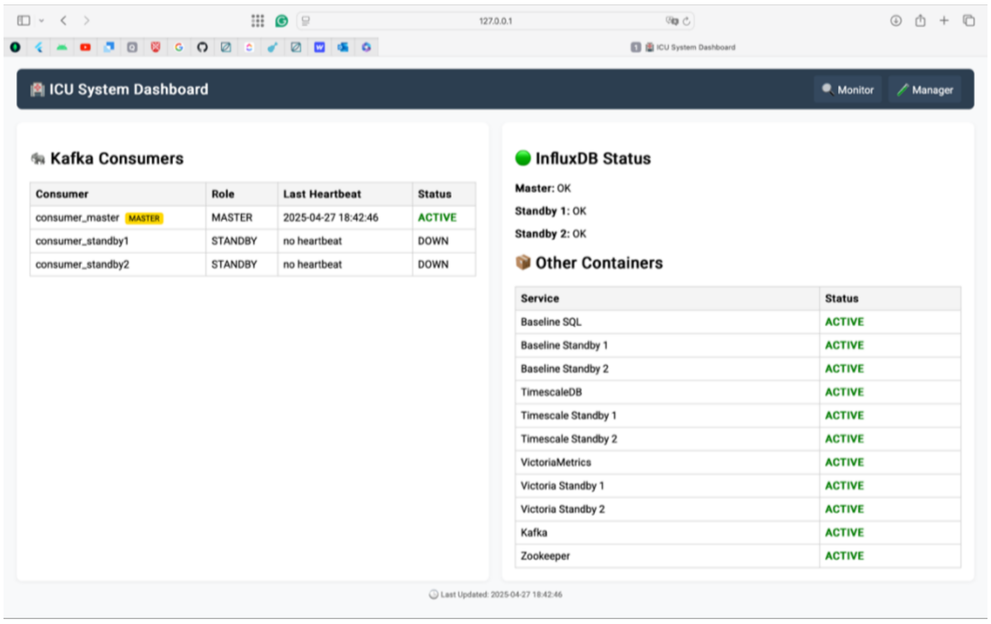
Upstairs B (609–616): Select All

609 610 611 612 613 614 615 616

Monitor

Heart Rate Alerts

Figure 3. Heart Rate Monitor UI


ICU System Dashboard

Monitor | Manager

Kafka Consumers

Consumer	Role	Last Heartbeat	Status
consumer_master MASTER	MASTER	2025-04-27 18:42:46	ACTIVE
consumer_standby1	STANDBY	no heartbeat	DOWN
consumer_standby2	STANDBY	no heartbeat	DOWN

InfluxDB Status

Master: OK

Standby 1: OK

Standby 2: OK

Other Containers

Service	Status
Baseline SQL	ACTIVE
Baseline Standby 1	ACTIVE
Baseline Standby 2	ACTIVE
TimescaleDB	ACTIVE
Timescale Standby 1	ACTIVE
Timescale Standby 2	ACTIVE
VictoriaMetrics	ACTIVE
Victoria Standby 1	ACTIVE
Victoria Standby 2	ACTIVE
Kafka	ACTIVE
Zookeeper	ACTIVE

Last Updated: 2025-04-27 18:42:46

Figure 4. Manager Dashboard UI

One Django web application is the real-time monitoring interface. Users retrieve both real-time and historical heart rate data by entering bed numbers on a live dashboard. First, the Django server searches the active database; if unavailable, it immediately turns back to the standby databases. Dynamic plotting of heart rate time series within the browser uses Chart.js, which refreshes data every five seconds to preserve an almost real-time view.

Supporting total system observability, a secondary Manager Dashboard shows the operational state of consumers, databases, and containerised services.

Monitor Dashboard (index.html):

- Select one or multiple ICU beds manually or via zone grouping.
- View heart rate time-series graphs, auto-refreshing every five seconds.
- Trigger historical data queries (1h, 2h, 6h, 12h, 24h).
- Detect critical or elevated heart rates with blinking, colour-coded alerts.
- Automatically fall back to standby databases if the active database is unavailable.

Manager Dashboard (manager.html):

- Display real-time heartbeat status and roles (MASTER/STANDBY) of Kafka consumers.
- Monitor container health across all services with colour-coded statuses.
- Auto-refresh system health every five seconds.

3.3.6. Dockerised Environment

The entire system is orchestrated within a Dockerised environment to guarantee portability, reproducibility, and environmental isolation. A Docker Compose file manages the deployment of all core services, including Zookeeper, Kafka brokers, time-series databases (InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL), and auxiliary Python modules. Dedicated ports are assigned to each service, such as 29092 for Kafka and 8086–8088 for the InfluxDB replicas. Containers expose internal and external interfaces to facilitate seamless communication while maintaining accessibility to local development. This containerised architecture enables rapid deployment, consistent test conditions, and effortless resets across testing cycles.

3.4. Database Design

Real-time heart rate telemetry supports InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL backend databases. The diversity allows a complete examination of database ingestion performance under consistent demand. All databases use a uniform schema logic to index time-based physiological signals by timestamps and bed identifiers. The multi-database design enables fair and comparable benchmarking with failover redundancy.

3.4.1. 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.

3.4.2. InfluxDB

InfluxDB is configured with a bucket named sensor_data, using an infinite retention policy to avoid premature data expiration. Data ingestion occurs via the Influx line protocol, where each record is treated under a measurement name (heart_rate), with bed_number stored as a tag and heart_rate as a field. Timestamps are automatically parsed and indexed. Failover is enabled through a replicated standby InfluxDB instance.

Key attributes:

- Schema is implicit and automatically derived at ingestion.
- Flux queries are used for historical data retrieval.
- Infinite retention policy configured (no automatic deletion).
- Pre-ingestion validation ensures malformed records are excluded.

3.4.3. 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 was 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 was validated with 4 chunks during testing.
- Supports Prometheus/Grafana dashboards via PostgreSQL plug-ins.

3.4.4. VictoriaMetrics

VictoriaMetrics is optimised for high-throughput, short-retention time-series workloads. It ingests data through an InfluxDB-compatible HTTP POST interface. Each heart rate entry is modelled as:

- metric_name: heart_rate
- labels: key-value pair with bed_number
- timestamp: epoch-based timestamp
- value: heart rate reading

Although VictoriaMetrics operates without explicit schema declarations, data must conform to metric-label-value conventions. During system validation, ingestion logs consistently handled over 13,000 records within a minimal memory and storage footprint.

Key attributes:

- Stateless ingestion is optimised for horizontal scaling.
- Prometheus query language (PromQL) supported for analysis.
- Best suited for high ingestion rates and low-latency retrievals.

3.4.5. 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 optimised for time-series operations but is a comparative baseline against specialised TSDBS. No partitioning or hypertable logic is applied.

Key Points:

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

3.5. Key Design Decisions and Rationale

The system design adhered to known scalability, fault tolerance, and performance principles, according to best practices in real-time telemetry engineering. Every option embodies a deliberate optimisation to enhance ingestion resilience and operational efficiency.

3.5.1. Kafka – Centric Event Streaming Architecture

Apache Kafka was chosen as the primary message bus because of its partitioned, durable, and high-throughput publish-subscribe architecture. This design facilitates asynchronous decoupling between the producer and numerous consumers, promotes scalable ingestion, and permits replayable message streams to enhance robustness against temporary errors.

3.5.2. Master-Standby Consumer Model with Heartbeat-Driven Failover

A master-standby consumer architecture was established to ensure uninterrupted ingesting during failure scenarios. Heartbeat signals, relayed over Kafka topics, allow standby consumers to independently identify master failures and elevate themselves without external assistance, ensuring system availability.

3.5.3. Multi-Database Benchmarking Support

The system supports InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL using a modular ingestion interface governed by environment settings. This abstraction facilitates empirical assessment of ingestion performance, resource efficiency, and scalability across diverse database backends under uniform workload conditions.

3.5.4. Preprocessing via correctedHR.py

To guarantee ingestion reliability, the correctedHR.py module standardises raw telemetry data into a structured CSV file (extracted_hr_data.csv) that includes verified bed identifiers, heart rates, and ISO 8601 timestamps. Pre-ingestion purification diminishes parsing overhead and eradicates schema inconsistency while streaming.

3.5.5. Real-Time Visualisation with Fallback Logic

The Django-based dashboard retrieves real-time and historical telemetry from the active database. In cases of database outage, automatic fallback mechanisms seamlessly redirect queries to standby instances, maintaining monitoring continuity and system observability without disrupting users.

3.5.6. Lightweight Dockerised Deployment

All system components are encased using Docker Compose, guaranteeing reproducible deployments, uniform setup across environments, and isolated benchmarking. This framework facilitates swift orchestration and potential transition to distributed orchestration platforms like Kubernetes.

3.5.7. Message Rate Configuration and Replay Control

The producer facilitates adjustable ingestion rates (10–1000 MPM) using environment settings, emulating diverse ICU workloads. Cyclic dataset replay prolongs test duration, enabling saturation testing and longitudinal performance evaluation under controlled, reproducible settings.

4. Implementation

4.1. 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, enabling development and testing. Reliable container management enabled by maOS'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 the implementation and was improved by several essential 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 building system modules, monitoring container performance, and validating data flows, improving the development lifetime and system dependability.

4.2. Key Libraries and Software Dependencies

The system employs a focused stack of libraries optimised for real-time telemetry:

Category	Library / Tool	Purpose
Messaging	confluent_kafka	High-performance Kafka producer/consumer client for heart rate data streaming
Resource Monitoring	psutil	Tracks per-message CPU and memory usage for benchmarking
Time-Series Databases	influxdb-client, influxdb	Interface with InfluxDB for heart rate ingestion and queries
Time-Series Databases	psycopg2-binary	PostgreSQL and TimescaleDB client for structured writes
Time-Series Databases	requests	HTTP client for sending ingestion data to VictoriaMetrics
Data Processing	pandas, numpy	Preprocessing and manipulation of large heart rate datasets
Data Processing	re, datetime (stdlib)	Regular expression parsing and timestamp normalization
Web Framework	Django	Web server for real-time dashboard and RESTful API endpoints
Frontend Visualization	Chart.js, Luxon.js	JavaScript libraries for responsive heart rate graphs and timestamp formatting
Heartbeat Monitoring	Kafka heartbeat topic + custom heartbeat emitters/listeners	Internal heartbeat communication and failover coordination

Containerization	Docker, Docker Compose	Encapsulation and orchestration of Kafka, databases, Django, and auxiliary modules
Version Control	Git, GitLab	Source code management and collaboration
Diagramming	PlantUML, diagrams	Automated generation of sequence and architecture diagrams
Monitoring (built-in)	Prometheus metrics exposure	System and consumer metrics natively exposed over HTTP ports

Table 5. Core Libraries and Dependencies

The choice of these libraries prioritises stability, performance, and compatibility with containerised, distributed, and real-time telemetry systems. Uniform standards across modules minimise integration challenges and maximise system observability.

4.3. 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.

4.3.1. 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, 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.

4.3.2. 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, 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, using conventional file I/O operations.

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

Consumer modules use the Kafka broker's heart_rate topic to commit heart rate messages to time-series databases. Consumer Master.py actively writes messages to all available databases, including TimescaleDB, VictoriaMetrics, Postgresql, and InfluxDB (master, standby 1, and standby 2).

The dynamic database targets of the ACTIVE_DB environment variable ensure flexible benchmarking and multi-database integration. Customers check ISO8601 timestamps for optimal write operations, employ client libraries customised for their databases, and validate message structure.

ConsumerStandBy1.py and consumerStandBy2.py passively track pulse signals in typical circumstances using Kafka's heartbeat topic. Following a timeout, standby consumers take on master responsibility, promote themselves, and write to all databases without operator interaction. With Prometheus metrics endpoints, all clients may monitor throughput, resource usage, error rates, and delays.

4.3.4. Django Web Dashboard – Real-Time Query and Visualisation Interface

The display module consists of two primary views:

- Monitor Page (index.html):

  Allows users to select ICU beds manually or by zones, retrieve real-time heart rate graphs refreshed every five seconds, and request historical data (1h, 6h, 12h, 24h). If the active database is unavailable, the fallback logic automatically queries the standby databases.
- Manager Page (manager.html):

  Displays the operational health of Kafka consumers (master and standbys), database nodes, and other services with real-time status updates and visual role indicators.

Visualisation is achieved client-side with Chart.js, enabling dynamic, responsive, and multi-patient monitoring inside the browser without manual refresh. Colour-coded alerts highlight elevated or critical heart rates immediately, enhancing clinical usability. This web-based approach enhances multi-user access, remote operability, and low-latency monitoring suitable for intensive care contexts.

4.3.5. Dockerised Environment

The entire system is orchestrated through Docker Compose, ensuring isolated, repeatable deployment across environments. The following services are containerised:

- Messaging Stack:
  - Zookeeper
  - Kafka Broker (with heart_rate and heartbeat topics configured)
- Time-Series Databases:
  - InfluxDB (master, standby1, standby2 instances)
  - TimescaleDB
  - VictoriaMetrics
  - PostgreSQL (control database)
- Application Layer:
  - Django Web Server
  - producer.py (streaming heart rate messages)
  - consumerMaster.py, consumerStandby1.py, consumerStandby2.py
  - heartbeat_emitter.py (embedded in master consumer)

Each container is networked internally and exposed through mapped ports (e.g., Kafka 29092, Django 8000). Prometheus metrics are natively exposed on service endpoints for real-time monitoring.

This design guarantees rapid setup/teardown, controlled benchmarking, and high modularity, forming the basis for future cloud-native deployments (e.g., Kubernetes).

4.4. Data Structures and Types

Message serialisation throughout the Kafka pipeline employs a consistent and lightweight JSON structure. Each message includes three fields:

- bed_number (string): Identifies the ICU bed.
- timestamp (ISO 8601 format): Captures the UTC event time.
- heart_rate (integer): The recorded heart rate in beats per minute (bpm).

This design ensures cross-component interoperability and efficient ingestion into various time-series databases, including InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL. During ingestion, Python dictionaries represent each message payload internally. Prior to publishing to the Kafka heart_rate topic, the dictionaries are serialised into JSON format. The datetime module ensures timestamp standardisation, while bed_number is typed as a string to maintain flexibility for future extensions.

On the visualisation side, the Django web dashboard fetches heart rate data via API endpoints and renders it using Chart.js. Each time-series dataset consists of {timestamp, value} pairs associated with selected beds, updated every five seconds to provide near-real-time monitoring in the browser.

Preprocessing scripts (correctedHR.py) utilise pandas and numpy for initial dataset cleaning, structure enforcement, and format normalisation. However, real-time ingestion flows favour lightweight dictionaries and JSON operations for speed and minimal overhead.
This carefully selected set of data structures, pandas for static preprocessing, lightweight JSON for live operations, and Chart.js datasets for browser rendering, provides optimal clarity, performance, and flexibility across the pipeline.

4.5. Algorithms and Core Logic

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

4.5.1. Message Rate Throttling (MPM Control)

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

$$interval = \frac{60}{\mathrm{MPM}} seconds$$

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

4.5.2. Resilient Fallback on Database Failure

The system implements a resilient fallback mechanism at the visualisation layer. Upon failed queries to the active database, the system sequentially retries with:

- Standby 1 database
- Standby 2 database (if necessary)

This fallback sequence ensures uninterrupted data visualisation across failures of InfluxDB, TimescaleDB, VictoriaMetrics, or PostgreSQL nodes, maintaining high availability without user intervention.

4.5.3. Heartbeat Monitoring and Autonomous Failover

At the consumer layer, robust heartbeat monitoring secures fault-tolerant ingestion. Key mechanisms include:

- heartbeat_emitter.py sends periodic heartbeat messages from the master consumer.
- Standby consumers monitor heartbeats via Kafka's heartbeat topic.
- Upon missing heartbeats for a configured timeout period, standby consumers:
    - Promote themselves to master.
    - Write a promotion lock to prevent split-brain scenarios.
    - Begin ingestion into all target databases.

When the original master recovers and resumes heartbeat emission, standby consumers detect the recovery and voluntarily demote themselves, restoring normal system balance.

4.5.4. Resource Monitoring Per Message

To facilitate performance analysis, the producer logs per-message resource metrics using the psutil library:

- CPU utilisation
- Memory consumption

Snapshots are taken immediately before and after each Kafka publish operation. This fine-grained resource telemetry supports benchmarking database ingestion efficiency under varying load conditions.

4.5.5. End-to-End Data Flow Logic

The complete pipeline orchestrates seamless real-time data processing:

(i) Raw hr.csv is preprocessed to extract cleaned heart rate records.

(ii) Producer reads cleaned data and streams structured JSON messages to Kafka.

(iii) Kafka brokers temporarily store and distribute messages asynchronously.

(iv) Master consumer ingests messages into multiple databases; standbys monitor heartbeats.

(v) Django dashboard queries time-series databases, renders live and historical heart rate graphs, and handles automatic fallback if needed.

This event-driven architecture, supported by modular service decomposition, ensures low latency, fault resilience, and high extensibility throughout the system.

4.6. Implementation Challenges and Decisions

During the system's integration phase, numerous complex engineering problems had to be solved, particularly those caused by the technologies' heterogeneity and the requirement to provide fault-tolerant real-time operations. Important choices were made to maintain each component's performance, flexibility, and modularity.

4.6.1. Database Driver and Write API Inconsistencies

Supporting four different database backends introduced complexities in the consumer modules.

- PostgreSQL/TimescaleDB (via psycopg2) required strict adherence to SQL syntax, explicit transactions, and careful type casting.
- InfluxDB (via influxdb-client) demanded Flux line protocol compliance.
- VictoriaMetrics ingestion used raw HTTP POST requests in Influx-compatible formats.

To accommodate these differences, database-specific write functions were modularised within the consumers, and conditional imports were employed to maintain flexibility without bloating inactive pathways.

4.6.2. Timestamp Normalisation Across the Pipeline

Database ingesting and inter-system querying depend on consistent timestamp formatting. 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.

4.6.3. Heartbeat Monitoring and Failover Coordination

The implementation of a reliable heartbeat-driven failover mechanism presented several challenges. Consumers were required to detect missing heartbeats rapidly while minimising the risk of false positives that could trigger unnecessary promotions. A promotion lock strategy was introduced to avoid conflicts where multiple standby consumers might attempt to promote simultaneously (split-brain scenarios). Furthermore, voluntary demotion protocols were embedded, allowing standbys to step down if the original master consumer resumed operation gracefully. Achieving robust failover required careful calibration of heartbeat intervals, timeouts, and state transitions, ensuring the system could maintain ingestion continuity under failure conditions without human intervention.

4.6.4. Flexible Environment-Driven Configuration

To enhance modularity and deployment flexibility, key system parameters, such as active database selection (ACTIVE_DB), message production rates (MESSAGES_PER_MINUTE), test duration (TEST_DURATION), and Kafka broker addresses, were externalised as environment variables. This approach decoupled configuration from the source code, enabling rapid testing across varying environments, smoother container orchestration under Docker Compose, and simplified resets without requiring rebuilds. Environment-driven configuration became particularly valuable during performance benchmarking and fault injection experiments, where fast parameter tuning was critical for efficient system iteration and analysis.

4.6.5. Django Web Dashboard Development

Transitioning from a Tkinter-based interface to a Django-powered web dashboard introduced several design and engineering considerations. Real-time updates needed to balance freshness and network efficiency, leading to the adoption of a five-second refresh cycle. Supporting multi-bed queries demanded careful backend API aggregation to minimise client-side performance degradation. Additionally, fallback querying logic had to be implemented within the browser layer, allowing transparent retries against standby databases without exposing backend failures to users. The integration of Chart.js further required accurate timestamp parsing and sorting to maintain graph continuity. Despite these complexities, adopting a Django-based architecture significantly improved system accessibility, scalability, and operational suitability for real-world ICU scenarios.

5. Testing

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

5.1. Test Strategy and Approach

The testing procedure was categorised into the following principal segments:

Test Category	Description
Unit Testing	Critical scripts (producer.py, consumerMaster.py, consumerStandby1.py, consumerStandby2.py) were tested via pytest with mock and monkeypatch fixtures, verifying Kafka interaction, database writes, and heartbeat mechanisms.
Integration Testing	The system was run end-to-end via main.py, confirming real-time data flow from CSV input through Kafka to consumers, databases, and Django UI.
System Testing	Monitored all components operating concurrently under realistic loads, validating stability, heartbeat integrity, and system responsiveness.
Performance Benchmarking	Automated tests (run_all_tests.py) evaluated ingestion performance across four database systems at nine different message rates (10–1000 MPM).
Failover and Recovery Testing	Simulated master consumer failures to verify standby consumer promotion mechanisms based on heartbeat monitoring.
User Interface Testing	Manually validated the Django dashboard's real-time graph updates and system monitoring displays.

Table 6. Test Strategy

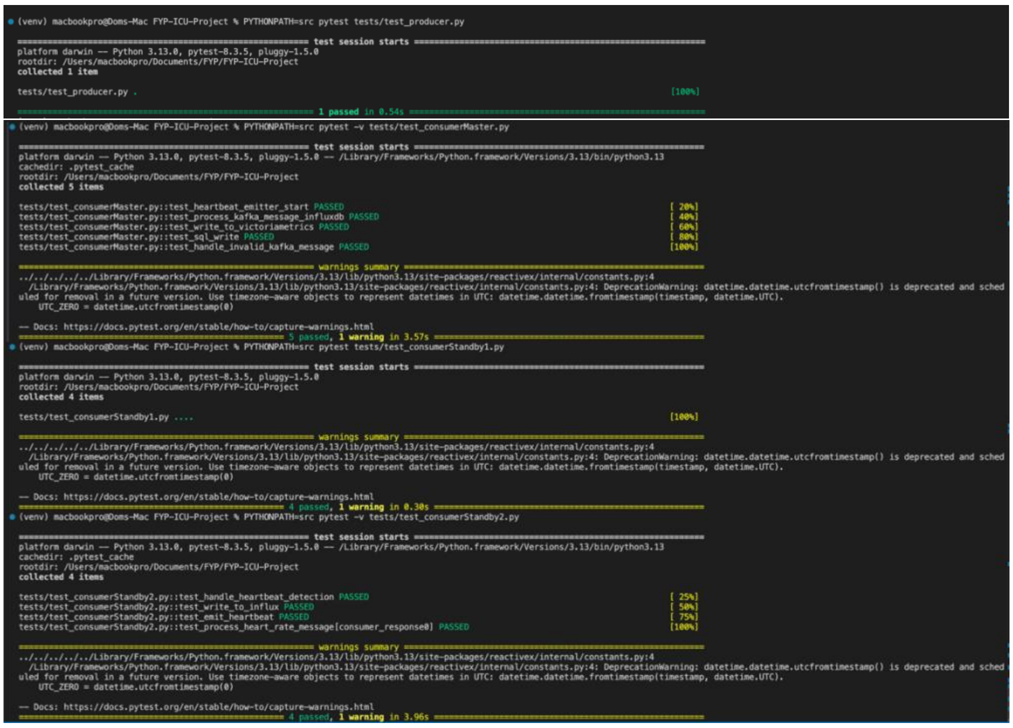
5.2. Functional Testing

Functional testing ensured that each system component behaved as intended:

Component	Validation Performed

producer.py	Verified message throttling, Kafka connectivity, and correct JSON serialization.
consumerMaster.py	Confirmed successful database writes across all target DBs and correct message parsing.
consumerStandby1.py	Validated heartbeat detection and automatic promotion to master upon failure.
consumerStandby2.py	Confirmed double-failover logic when both master and standby1 were unavailable.

Table 7. Functional Testing



```
● (venv) macbookpro@Doms-Mac FYP-ICU-Project % PYTHONPATH=src pytest tests/test_producer.py

============================================================ test session starts ============================================================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0
rootdir: /Users/macbookpro/Documents/FYP/FYP-ICU-Project
collected 1 item

tests/test_producer.py .                                                                                                              [100%]

============================================================= 1 passed in 0.54s =============================================================
● (venv) macbookpro@Doms-Mac FYP-ICU-Project % PYTHONPATH=src pytest -v tests/test_consumerMaster.py

============================================================ test session starts ============================================================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0 -- /Library/Frameworks/Python.framework/Versions/3.13/bin/python3.13
cachedir: .pytest_cache
rootdir: /Users/macbookpro/Documents/FYP/FYP-ICU-Project
collected 5 items

tests/test_consumerMaster.py::test_heartbeat_emitter_start PASSED                                                                     [ 20%]
tests/test_consumerMaster.py::test_process_kafka_message_influxdb PASSED                                                              [ 40%]
tests/test_consumerMaster.py::test_write_to_victoriametrics PASSED                                                                    [ 60%]
tests/test_consumerMaster.py::test_sql_write PASSED                                                                                   [ 80%]
tests/test_consumerMaster.py::test_handle_invalid_kafka_message PASSED                                                                [100%]

============================================================= warnings summary ==============================================================
../../../../../Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4
  /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4: DeprecationWarning: datetime.datetime.utcfromtimestamp() is deprecated and sched
uled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.fromtimestamp(timestamp, datetime.UTC).
    UTC_ZERO = datetime.utcfromtimestamp(0)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================================================= 5 passed, 1 warning in 3.57s ========================================================
● (venv) macbookpro@Doms-Mac FYP-ICU-Project % PYTHONPATH=src pytest tests/test_consumerStandby1.py

============================================================ test session starts ============================================================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0
rootdir: /Users/macbookpro/Documents/FYP/FYP-ICU-Project
collected 4 items

tests/test_consumerStandby1.py ....                                                                                                   [100%]

============================================================= warnings summary ==============================================================
../../../../../Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4
  /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4: DeprecationWarning: datetime.datetime.utcfromtimestamp() is deprecated and sched
uled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.fromtimestamp(timestamp, datetime.UTC).
    UTC_ZERO = datetime.utcfromtimestamp(0)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================================================= 4 passed, 1 warning in 0.30s ========================================================
● (venv) macbookpro@Doms-Mac FYP-ICU-Project % PYTHONPATH=src pytest -v tests/test_consumerStandby2.py

============================================================ test session starts ============================================================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0 -- /Library/Frameworks/Python.framework/Versions/3.13/bin/python3.13
cachedir: .pytest_cache
rootdir: /Users/macbookpro/Documents/FYP/FYP-ICU-Project
collected 4 items

tests/test_consumerStandby2.py::test_handle_heartbeat_detection PASSED                                                                [ 25%]
tests/test_consumerStandby2.py::test_write_to_influx PASSED                                                                           [ 50%]
tests/test_consumerStandby2.py::test_emit_heartbeat PASSED                                                                            [ 75%]
tests/test_consumerStandby2.py::test_process_heart_rate_message[consumer_response0] PASSED                                            [100%]

============================================================= warnings summary ==============================================================
../../../../../Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4
  /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages/reactivex/internal/constants.py:4: DeprecationWarning: datetime.datetime.utcfromtimestamp() is deprecated and sched
uled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.fromtimestamp(timestamp, datetime.UTC).
    UTC_ZERO = datetime.utcfromtimestamp(0)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================================================= 4 passed, 1 warning in 3.96s ========================================================
```

Figure 5. Unit Testing Results

5.3. Performance Testing

Performance benchmarking validated ingestion stability under various stress scenarios:

Dimension	Details
Databases Tested	Baseline SQL, InfluxDB, TimescaleDB, VictoriaMetrics
Ingestion Rates	10, 20, 40, 60, 80, 100, 200, 500, 1000 messages per minute
Duration	300 seconds per scenario
Metrics Collected	CPU usage, memory usage, write latency per message

Table 8. Performance Testing Dimensions

The tests were run automatically with the following flow:

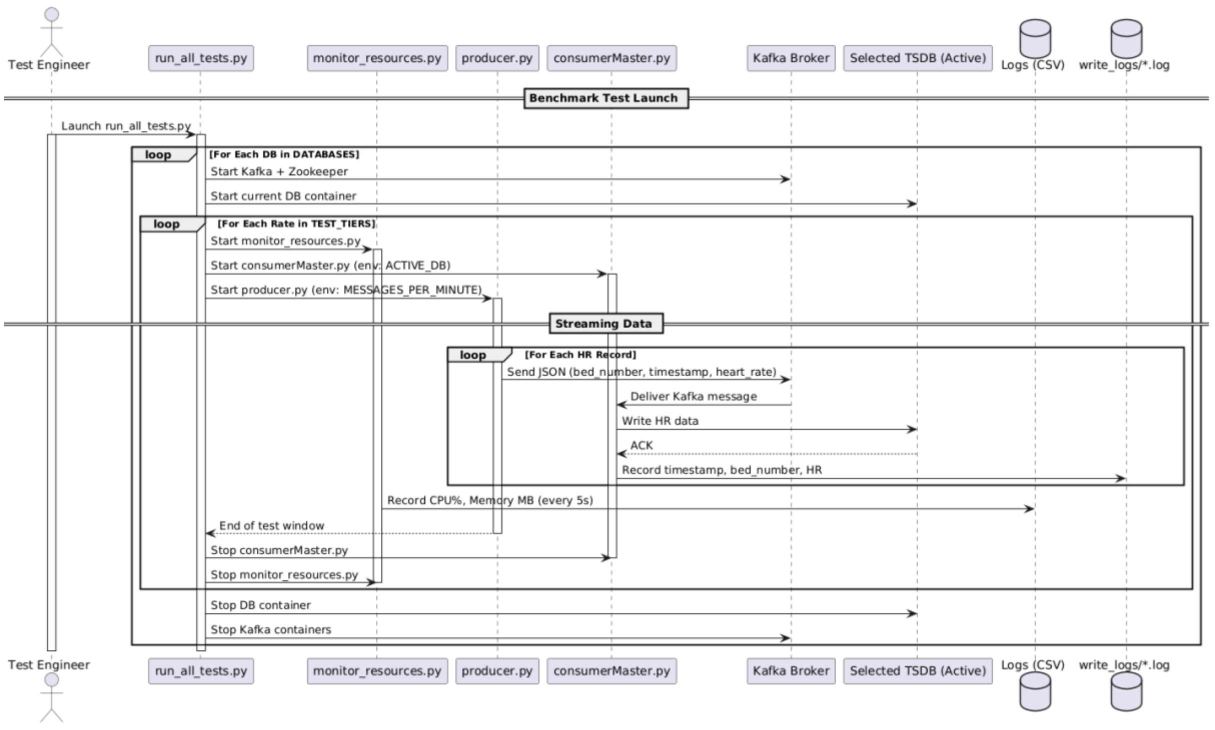

Test Engineer
run_all_tests.py
monitor_resources.py
producer.py
consumerMaster.py
Kafka Broker
Selected TSDB (Active)
Logs (CSV)
write_logs/*.log
Benchmark Test Launch
Launch run_all_tests.py
loop [For Each DB in DATABASES]
Start Kafka + Zookeeper
Start current DB container
loop [For Each Rate in TEST_TIERS]
Start monitor_resources.py
Start consumerMaster.py (env: ACTIVE_DB)
Start producer.py (env: MESSAGES_PER_MINUTE)
Streaming Data
loop [For Each HR Record]
Send JSON (bed_number, timestamp, heart_rate)
Deliver Kafka message
Write HR data
ACK
Record timestamp, bed_number, HR
Record CPU%, Memory MB (every 5s)
End of test window
Stop consumerMaster.py
Stop monitor_resources.py
Stop DB container
Stop Kafka containers

Figure 6. Performance Test Sequence Diagram

5.4. Integration and System Testing

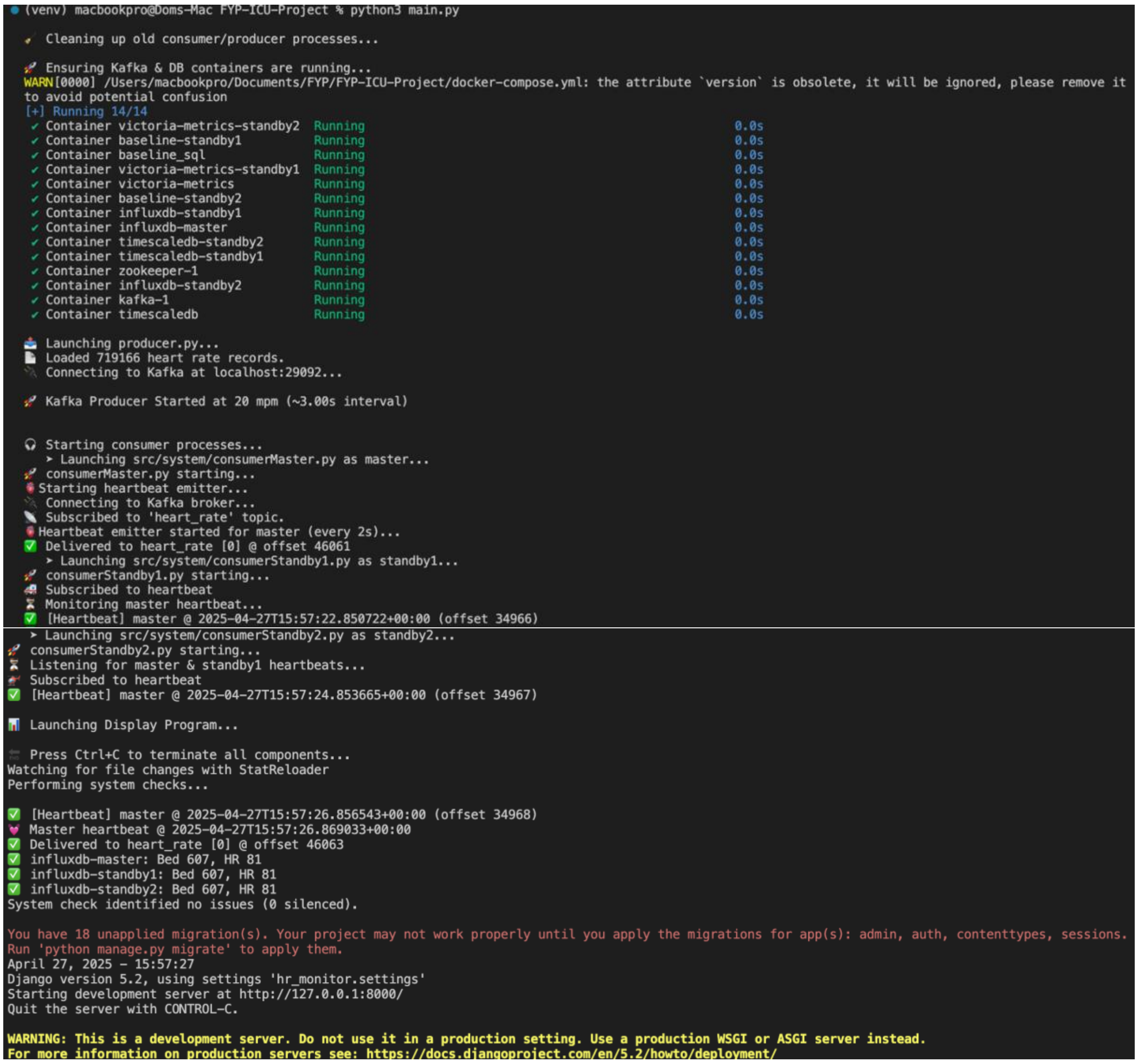
The overall system was validated by running the full pipeline via main.py, which launched:

- Kafka and Zookeeper services
- Databases (InfluxDB, TimescaleDB, VictoriaMetrics, Baseline SQL)
- Producer, Master Consumer, Standby Consumers
- Django Web Application

Test Observations:

- Continuous data production and ingestion at configured MPM rates.
- Reliable Kafka delivery to consumer groups.
- Heartbeats are emitted every 2 seconds; consumers monitor heartbeats actively.
- Standby consumers maintained readiness for promotion.
- Django dashboard was launched successfully.

Sample Log Evidence:



```
● (venv) macbookpro@Doms-Mac FYP-ICU-Project % python3 main.py

  🧹 Cleaning up old consumer/producer processes...

  🚀 Ensuring Kafka & DB containers are running...
  WARN[0000] /Users/macbookpro/Documents/FYP/FYP-ICU-Project/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it
  to avoid potential confusion
  [+] Running 14/14
   ✔ Container victoria-metrics-standby2  Running                                                          0.0s
   ✔ Container baseline-standby1          Running                                                          0.0s
   ✔ Container baseline_sql               Running                                                          0.0s
   ✔ Container victoria-metrics-standby1  Running                                                          0.0s
   ✔ Container victoria-metrics           Running                                                          0.0s
   ✔ Container baseline-standby2          Running                                                          0.0s
   ✔ Container influxdb-standby1          Running                                                          0.0s
   ✔ Container influxdb-master            Running                                                          0.0s
   ✔ Container timescaledb-standby2       Running                                                          0.0s
   ✔ Container timescaledb-standby1       Running                                                          0.0s
   ✔ Container zookeeper-1                Running                                                          0.0s
   ✔ Container influxdb-standby2          Running                                                          0.0s
   ✔ Container kafka-1                    Running                                                          0.0s
   ✔ Container timescaledb                Running                                                          0.0s

  📤 Launching producer.py...
  📄 Loaded 719166 heart rate records.
  🔌 Connecting to Kafka at localhost:29092...

  🚀 Kafka Producer Started at 20 mpm (~3.00s interval)


  🎧 Starting consumer processes...
    ➤ Launching src/system/consumerMaster.py as master...
  🚀 consumerMaster.py starting...
  🫀Starting heartbeat emitter...
  🔌 Connecting to Kafka broker...
  📡 Subscribed to 'heart_rate' topic.
  🫀Heartbeat emitter started for master (every 2s)...
  ✅ Delivered to heart_rate [0] @ offset 46061
    ➤ Launching src/system/consumerStandby1.py as standby1...
  🚀 consumerStandby1.py starting...
  🚑 Subscribed to heartbeat
  ⏳ Monitoring master heartbeat...
  ✅ [Heartbeat] master @ 2025-04-27T15:57:22.850722+00:00 (offset 34966)
   ➤ Launching src/system/consumerStandby2.py as standby2...
🚀 consumerStandby2.py starting...
⏳ Listening for master & standby1 heartbeats...
🚁 Subscribed to heartbeat
✅ [Heartbeat] master @ 2025-04-27T15:57:24.853665+00:00 (offset 34967)

📊 Launching Display Program...

🔚 Press Ctrl+C to terminate all components...
Watching for file changes with StatReloader
Performing system checks...

✅ [Heartbeat] master @ 2025-04-27T15:57:26.856543+00:00 (offset 34968)
💓 Master heartbeat @ 2025-04-27T15:57:26.869033+00:00
✅ Delivered to heart_rate [0] @ offset 46063
✅ influxdb-master: Bed 607, HR 81
✅ influxdb-standby1: Bed 607, HR 81
✅ influxdb-standby2: Bed 607, HR 81
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 27, 2025 - 15:57:27
Django version 5.2, using settings 'hr_monitor.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
```

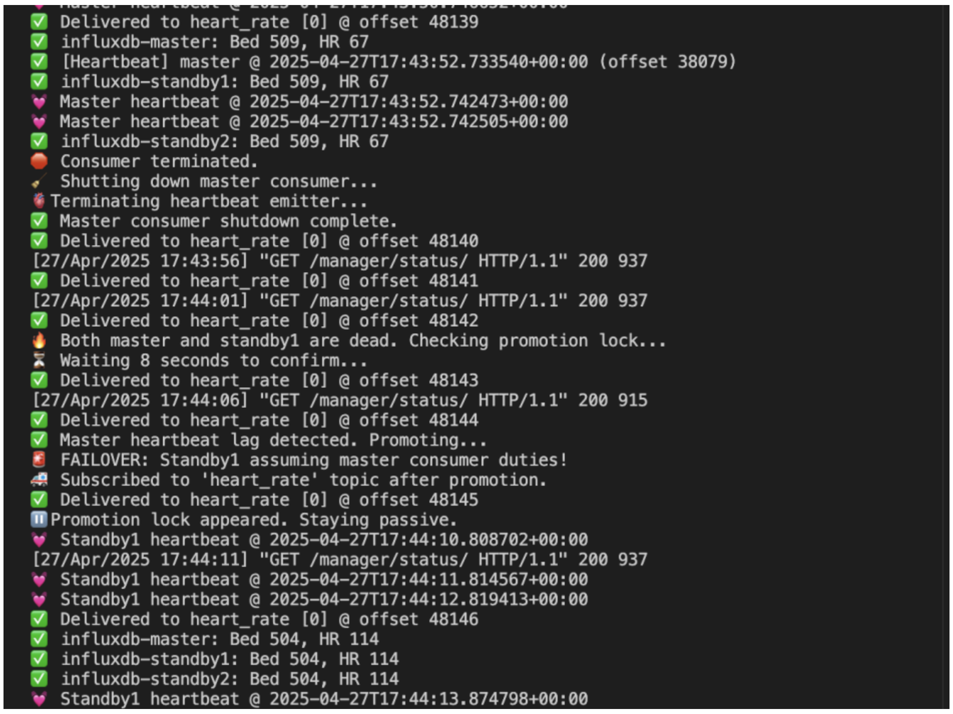
Figure 7. Integration and System Testing Logs

5.5. Failover and Recovery Testing

Failover mechanisms were manually tested by:

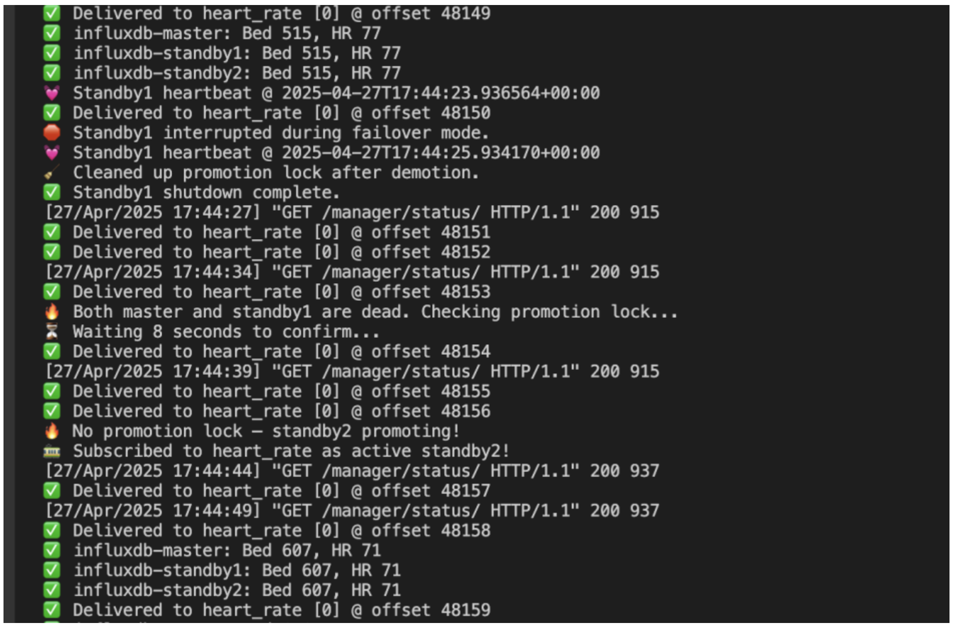
- Simulating the termination of the master consumer process
- Observing standby1 detecting heartbeat loss and promoting itself
- Confirming that standby1 began producing heartbeats and ingesting data
- Similarly, verifying standby2 promotion if both master and standby1 failed

The test shows that the failover mechanism worked properly:



```
✅ Delivered to heart_rate [0] @ offset 48139
✅ influxdb-master: Bed 509, HR 67
✅ [Heartbeat] master @ 2025-04-27T17:43:52.733540+00:00 (offset 38079)
✅ influxdb-standby1: Bed 509, HR 67
💓 Master heartbeat @ 2025-04-27T17:43:52.742473+00:00
💓 Master heartbeat @ 2025-04-27T17:43:52.742505+00:00
✅ influxdb-standby2: Bed 509, HR 67
🔴 Consumer terminated.
🧹 Shutting down master consumer...
🫀Terminating heartbeat emitter...
✅ Master consumer shutdown complete.
✅ Delivered to heart_rate [0] @ offset 48140
[27/Apr/2025 17:43:56] "GET /manager/status/ HTTP/1.1" 200 937
✅ Delivered to heart_rate [0] @ offset 48141
[27/Apr/2025 17:44:01] "GET /manager/status/ HTTP/1.1" 200 937
✅ Delivered to heart_rate [0] @ offset 48142
🔥 Both master and standby1 are dead. Checking promotion lock...
⏳ Waiting 8 seconds to confirm...
✅ Delivered to heart_rate [0] @ offset 48143
[27/Apr/2025 17:44:06] "GET /manager/status/ HTTP/1.1" 200 915
✅ Delivered to heart_rate [0] @ offset 48144
✅ Master heartbeat lag detected. Promoting...
🚨 FAILOVER: Standby1 assuming master consumer duties!
🚑 Subscribed to 'heart_rate' topic after promotion.
✅ Delivered to heart_rate [0] @ offset 48145
⏸️Promotion lock appeared. Staying passive.
💓 Standby1 heartbeat @ 2025-04-27T17:44:10.808702+00:00
[27/Apr/2025 17:44:11] "GET /manager/status/ HTTP/1.1" 200 937
💓 Standby1 heartbeat @ 2025-04-27T17:44:11.814567+00:00
💓 Standby1 heartbeat @ 2025-04-27T17:44:12.819413+00:00
✅ Delivered to heart_rate [0] @ offset 48146
✅ influxdb-master: Bed 504, HR 114
✅ influxdb-standby1: Bed 504, HR 114
✅ influxdb-standby2: Bed 504, HR 114
💓 Standby1 heartbeat @ 2025-04-27T17:44:13.874798+00:00
```

Figure 8. Failover Test Result (Master Dies)



```
✅ Delivered to heart_rate [0] @ offset 48149
✅ influxdb-master: Bed 515, HR 77
✅ influxdb-standby1: Bed 515, HR 77
✅ influxdb-standby2: Bed 515, HR 77
💓 Standby1 heartbeat @ 2025-04-27T17:44:23.936564+00:00
✅ Delivered to heart_rate [0] @ offset 48150
🔴 Standby1 interrupted during failover mode.
💓 Standby1 heartbeat @ 2025-04-27T17:44:25.934170+00:00
🧹 Cleaned up promotion lock after demotion.
✅ Standby1 shutdown complete.
[27/Apr/2025 17:44:27] "GET /manager/status/ HTTP/1.1" 200 915
✅ Delivered to heart_rate [0] @ offset 48151
✅ Delivered to heart_rate [0] @ offset 48152
[27/Apr/2025 17:44:34] "GET /manager/status/ HTTP/1.1" 200 915
✅ Delivered to heart_rate [0] @ offset 48153
🔥 Both master and standby1 are dead. Checking promotion lock...
⏳ Waiting 8 seconds to confirm...
✅ Delivered to heart_rate [0] @ offset 48154
[27/Apr/2025 17:44:39] "GET /manager/status/ HTTP/1.1" 200 915
✅ Delivered to heart_rate [0] @ offset 48155
✅ Delivered to heart_rate [0] @ offset 48156
🔥 No promotion lock — standby2 promoting!
🎟 Subscribed to heart_rate as active standby2!
[27/Apr/2025 17:44:44] "GET /manager/status/ HTTP/1.1" 200 937
✅ Delivered to heart_rate [0] @ offset 48157
[27/Apr/2025 17:44:49] "GET /manager/status/ HTTP/1.1" 200 937
✅ Delivered to heart_rate [0] @ offset 48158
✅ influxdb-master: Bed 607, HR 71
✅ influxdb-standby1: Bed 607, HR 71
✅ influxdb-standby2: Bed 607, HR 71
✅ Delivered to heart_rate [0] @ offset 48159
```

Figure 9. Failover Test Result (Standby1 Dies)

5.6. Test Tools and Automation

Below is a summary of the testing tools and scripts used:

Tool / Script	Purpose
pytest	Unit testing with mock-based validation of Kafka/database behavior

run_all_tests.py	Automation of stress testing across ingestion rates and database types
monitor_resources.py	Periodic capture of container CPU and memory usage
psutil	Logging per-message CPU and memory resource usage
record_system_conditions.py	Capturing host system specifications before test runs

Table 9. Test Tools and Scripts

5.7. Test Coverage and Limitations

Achievements	Limitations
Full ingestion pipeline validation (CSV → Kafka → Consumers → DB → UI)	No GUI automation for dashboard testing
Real-time heartbeat monitoring and successful failover	Security and encryption testing were out of scope
Performance stability validated up to 1000 MPM	No CI/CD pipelines integrated into testing
Detailed per-message and global resource tracking	-

Table 10. Test Coverage and Limitations

6. System Evaluation and Experimental Results

6.1. Evaluation Methodology

The basis of this analysis is the benchmarking strategy covered in Section 5. Four database systems—Baseline SQL, InfluxDB, TimescaleDB, and VictoriaMetrics—wer 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 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 300 seconds. A multi-tiered logging system was used to document extensive performance data throughout 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:

(i) Scalability Analysis – To determine which databases maintain stable performance as ingestion rates increase.

(ii) Resource Efficiency – To compare CPU and memory usage across both producer and containerised environments.

(iii) 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 the same hardware settings, guaranteeing a fair and consistent basis for comparison. The approach creates a basis for informed recommendations in real-time medical monitoring systems and helps clarify empirical results on system performance.

6.2. 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 the relative strengths and trade-offs among the four databases.

6.2.1. Global CPU and Memory Usage (Container-Wide)

CPU and memory measurements were collected at the container level every five seconds during each 300-second benchmarking session using monitor_resources.py. These metrics provide insight into the total resource footprint of each time-series database as ingestion rates increase. The following figures summarise the trend lines:

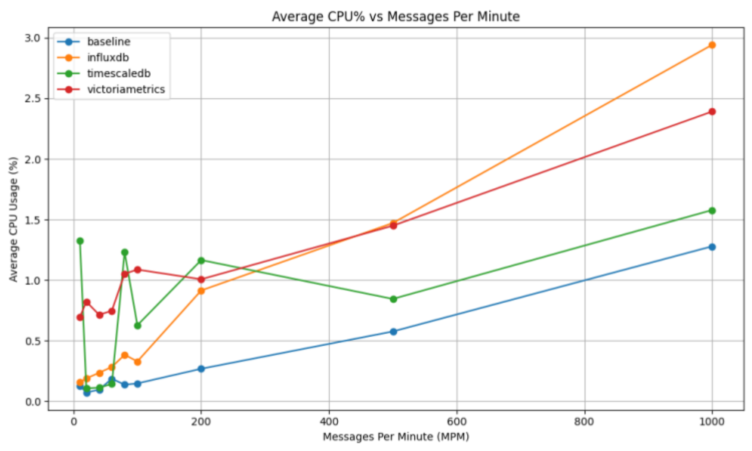


Figure 10. Average CPU% vs Messages Per Minute Chart

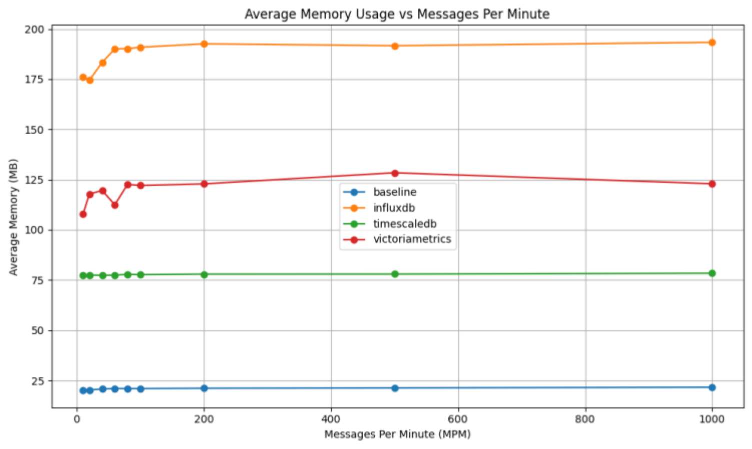


Figure 11. Average Memory Usage vs Messages Per Minute Chart

(i) To concisely capture the performance observations:

Database	CPU Usage Trend	Max Memory Usage	Notes
InfluxDB	Highest, escalates at high MPM	~195 MB	High internal processing overhead; fastest ingestion
VictoriaMetrics	Moderate, consistent scaling	~125 MB	Balanced efficiency with predictable scaling
TimescaleDB	Modest and stable	~77 MB	Highly memory-efficient; suitable for resource-constrained environments
Baseline SQL	Lowest, due to slow write rates	~22 MB	Not optimised for time-series ingestion

Table 11. Global CPU and Memory Usage Observation

(ii) Conclusion:

While InfluxDB achieves the highest ingestion rates, it does so at the cost of elevated CPU and memory consumption. VictoriaMetrics and TimescaleDB demonstrate better resource efficiency, making them preferable for scalable or resource-sensitive deployments.

6.2.2. Producer Resource Usage (Per-Message)

The resource performance of producer.py was evaluated per message using psutil, isolating the costs of serialising, formatting, and publishing heart rate records to Kafka. This assessment highlights the impact of upstream buffering and client efficiency across different database targets. The following figures summarise producer-side resource trends:

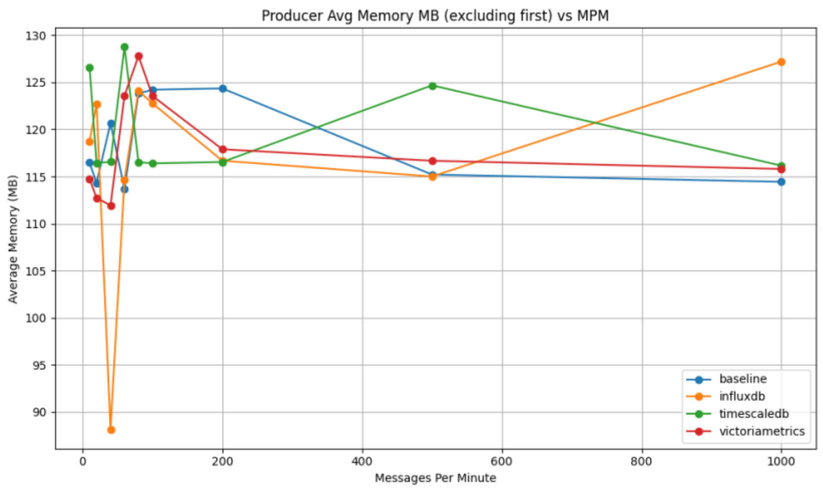


Figure 12. Producer Average Memory Usage vs Messages Per Minute Chart

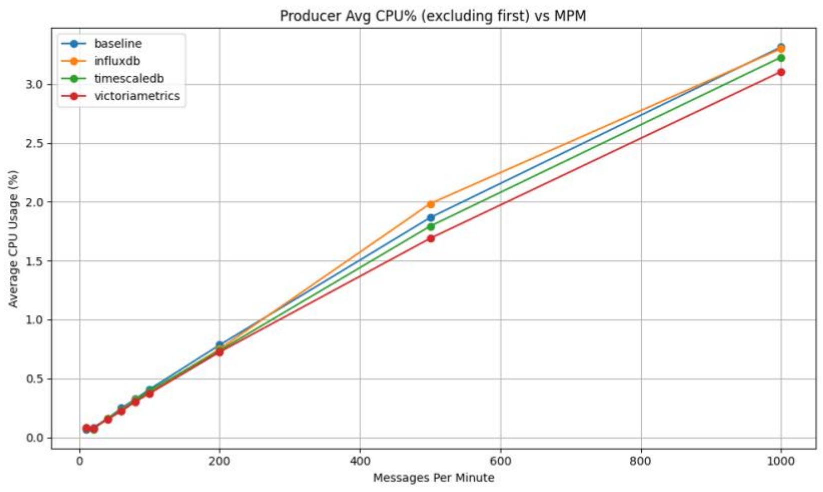


Figure 13. Producer Average CPU% vs Messages Per Minute Chart

(i) Summary of producer resource efficiency:

Database	CPU Usage Trend	Memory Usage Trend	Notes
VictoriaMetrics	Lowest and most stable CPU%	Stable (~114–120 MB)	Most efficient for CPU; ideal for edge devices
InfluxDB	Moderate CPU%, fluctuating	Slight memory variability (~125 MB)	Fluctuations likely due to HTTP batching overhead
TimescaleDB	Moderate CPU%, stable	Stable (~115 MB)	Efficient; slightly higher CPU at high MPM
Baseline SQL	Stable but marginally higher CPU at high loads	Stable (~114 MB)	Acceptable for non-time-series optimized systems

Table 12. Producer Resource Usage Observation

(ii) Conclusion:

VictoriaMetrics consistently demonstrated the lowest CPU and memory footprint per message, making it the optimal choice for deployments where computational efficiency is critical. InfluxDB offered competitive performance but exhibited greater variability under stress, while TimescaleDB remained a strong performer in resource-constrained environments.

6.2.3. Latency and Throughput

The evaluation primarily focused on two critical performance metrics:

- Write Throughput — the number of successful database writes per second.
- Write Latency — the time interval from Kafka message reception to database confirmation.

Measurements were collected across all test scenarios using detailed per-message write logs. The following figures and findings summarise the outcomes:

.

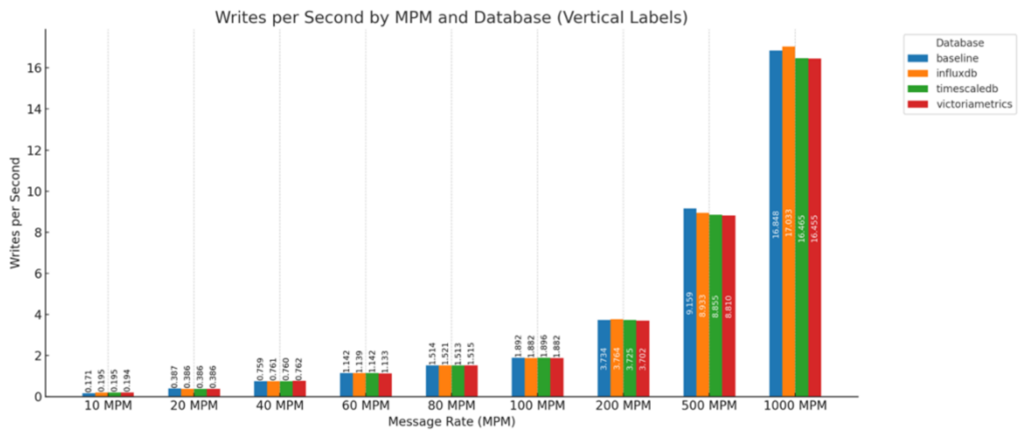


Figure 14. Write Throughput vs Messages Per Minute Chart

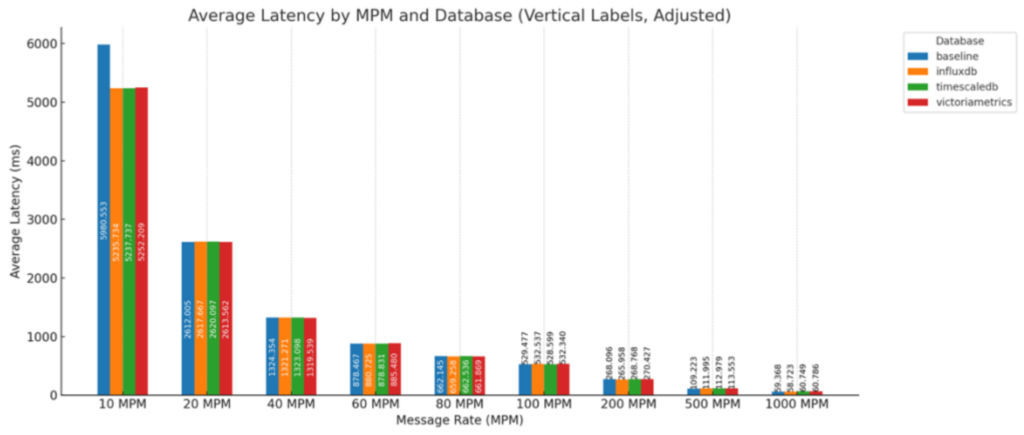


Figure 15. Average Latency vs Messages Per Minute Chart

Database	Top Writes_per_sec	Best Avg_Latency_ms
Baseline	16.8 writes/sec	59.4 ms
InfluxDB	17.0 writes/sec	58.7 ms
TimescaleDB	16.5 writes/sec	60.7 ms
VictoriaMetrics	16.5 writes/sec	60.8 ms

Table 13. Latency and Throughput Observation

(i) Observation:

InfluxDB achieved the highest throughput and lowest average latency, confirming its superior suitability for high-frequency ingestion. TimescaleDB and VictoriaMetrics demonstrated similar performance, scaling efficiently without saturation. Baseline SQL exhibited competitive throughput but unstable latency at low message rates, highlighting its limitations for real-time time-series workloads.

(ii) Conclusion:

Time-series optimised systems consistently outperform relational baselines for real-time telemetry among the evaluated databases. InfluxDB offers peak performance, while TimescaleDB and VictoriaMetrics balance ingestion stability with lower resource costs.

6.2.4. Message Gap Analysis

Database	**10 MPM**	**20 MPM**	**40 MPM**	**60 MPM**	**80 MPM**	**100 MPM**	**200 MPM**	**500 MPM**	**1000 MPM**
Baseline (SQL)	0	0	1	2	3	7	21	109	610
InfluxDB	0	0	1	2	4	7	19	156	514
TimescaleDB	0	0	1	2	4	6	26	177	676
VictoriaMetrics	0	0	1	2	5	8	31	187	683

Table 14. Message Gap

As the ingestion rate (messages per minute, MPM) rose, all database systems had more missing messages ("gaps"), showing ingestion dependability degradation under increasing loads. At lower rates (10–40 MPM) with few gaps (0–1 missed messages), all databases were nearly reliable. However, as demand increased to 500 and 1000 MPM, all systems' gaps grew. InfluxDB had the fewest lost messages at practically all speeds, including 514 at 1000 MPM, compared to TimescaleDB (676) and VictoriaMetrics (683). The basic SQL implementation has 610 gaps at 1000 MPM, worse than time-series-optimized databases. This strengthens InfluxDB's high-frequency data streaming ingestion robustness.

6.3. Comparative Analysis

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

Table 15. Comparative Summary of Database Performance

InfluxDB consistently achieved the highest throughput and message accuracy, making it ideal for environments where ingestion speed and reliability are critical, despite its higher resource footprint. TimescaleDB offered superior memory efficiency across varying loads, making it well-suited for resource-constrained systems requiring SQL compatibility. VictoriaMetrics balanced scalability and CPU efficiency, positioning it as an attractive option for high-ingestion systems seeking operational stability with moderate resource usage. Baseline SQL, while operational, exhibited the highest latency and lowest throughput, reaffirming its unsuitability for real-time telemetry ingestion. Database selection in real-time Kafka pipelines should align with deployment priorities: InfluxDB for maximum speed, TimescaleDB for memory-optimised SQL environments, and VictoriaMetrics for scalable, resource-efficient ingestion.

6.4. System Information Impact

System-level diagnostics were recorded at the beginning of each test using the system_info_summary.csv dataset to validate the consistency and reliability of performance results. Key metrics and their relevance are summarised below:

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

Table 16. System Diagnostics for Test Validity

All benchmarks were conducted on a consistent macOS 15.3.2 environment (10-core CPU, 16 GB RAM) with the system plugged into a power source to prevent background performance throttling, a typical behaviour on Apple Silicon systems. Initial system metrics showed CPU utilisation at approximately 33% and memory usage around 56%, ensuring no external processes skewed the results. Thus, observed resource usage patterns during tests are attributable solely to database behaviours rather than host limitations. This consistent hardware and power configuration strengthened the experimental validity, allowing the benchmarking to focus entirely on internal system variations across database backends.

6.5. Functional Observations (Manual Tests)

In addition to automated benchmarking, critical components were manually validated to ensure real-world operational robustness. Testing focused on the Display Program for real-time data retrieval and the failover system for consumer resilience under failure scenarios.

6.5.1. Display Program Validation

User-driven queries were conducted on the Django dashboard by selecting specific bed numbers during active data ingestion. The following outcomes were observed:

- Accurate Data Retrieval: Heart rate data for selected beds was consistently retrieved, confirming effective query execution and database connectivity.
- Real-Time Graphing: Visualisations refreshed every five seconds without lag, faithfully representing live variations in heart rate trends.
- System Stability: No data loss, graphical glitches, or system interruptions were noted during prolonged usage sessions exceeding five minutes.

These results validate the UI's ability to maintain reliable, near-real-time performance under continuous ingestion loads.

6.5.2. Failover Test System Validation

Failover behaviour was assessed by deliberately simulating master consumer failure:

1. Controlled Failure: The consumerMaster.py process was forcefully stopped during live Kafka ingestion by suspending its associated Docker container.
2. Automatic Promotion: The standby consumer (consumerStandBy1.py) detected the heartbeat loss and promoted itself to active ingestion mode within the configured threshold.
3. Seamless Continuity: Kafka acknowledgements (ACKS) confirm uninterrupted message delivery. No data gaps, duplicates, or timestamp anomalies were observed across log records.

Further validation through log comparison showed consistent sequencing of bed number and heart rate values between the master and promoted standby consumers, ensuring transactional integrity across failover events.

6.5.3. Summary

Manual tests confirmed the system’s ability to maintain visualisation integrity, data continuity, and automatic failover under adverse conditions. Despite the absence of automated GUI testing, the tested components demonstrated production-grade resilience in realistic scenarios.

6.6. Limitations and Future Work

Although the system achieved its real-time ingestion, fault tolerance, and database benchmarking objectives, several limitations were identified, guiding directions for future enhancement.

6.6.1. Identified Limitations

- Partial GUI Automation:
  While system components were unit and integration tested, the Django dashboard underwent only manual validation. No formal GUI automation (e.g., Selenium, Playwright) was employed, leaving edge-case interface robustness partially unverified.
- Security Measures Omitted:
  Security mechanisms—such as authentication, authorisation, and encryption—were beyond the project's academic scope, making the system unsuitable for production healthcare environments without further hardening.

6.6.2. Future Enhancements

Future work should focus on expanding system resilience, observability, and scalability:

- Continuous Integration (CI): Introduce a CI/CD pipeline to automate unit, integration, and system tests, improving development robustness.
- Extended Performance Metrics: For deeper operational insight, capture additional metrics, such as end-to-end ingestion latency, Kafka consumer lag, disk I/O statistics, and system time-to-stabilisation.
- Schema Enforcement: Implement JSON schema validation or transition to Avro/Protobuf with a Kafka Schema Registry to enforce message integrity.
- Security Hardening: Integrate role-based access control (RBAC), TLS encryption, and API authentication to support safe deployment in sensitive environments.

- Scalability Testing: Conduct distributed system evaluations using Kubernetes or multi-node Docker setups to validate system behaviour under actual production-like loads.

6.7. Societal and Commercial Implications

The heart rate telemetry system developed in this study has significant societal and technological implications, particularly in environments requiring real-time physiological monitoring. Designed with modularity, fault tolerance, and time-series optimisation, the architecture offers flexibility for deployment in diverse real-world scenarios.

6.7.1. Healthcare Applications and Societal Value

In healthcare, the system's primary application is real-time heart rate monitoring in intensive care units (ICUs), where it can enhance patient outcomes by enabling early detection of arrhythmic patterns, facilitating rapid responses to sudden physiological changes, and supporting data-driven clinical audits. The platform's low-latency architecture can be lifesaving in high-dependency wards, where timely access to accurate telemetry is crucial. Furthermore, TimescaleDB's stable memory usage demonstrates the system's ability to function efficiently even on lower-end hardware, improving accessibility for resource-limited hospitals.

6.7.2. Scalability and Reusability

Beyond ICU applications, the system's containerised, Kafka-based design enables hospital-wide scalability, supporting centralised monitoring across multiple departments or facilities. Its modular components are also adaptable for wearable health trackers in fitness and wellness sectors, athlete monitoring during training and rehabilitation, and remote health diagnostics in home-care environments. Given its ability to process physiological signals at sub-200ms latency and handle ingestion rates up to 1000 messages per minute, the system is well-suited as a backend for real-time dashboards and clinical alerting platforms.

6.7.3. Commercial Opportunities

Built with open-source technologies and flexible Python modules, the system offers a strong foundation for commercial SaaS products targeting elder care, sports science, and medical telemetry. Integration with cloud-hosted time-series databases and enterprise authentication layers would significantly enhance its market readiness. Startups and academic spinouts could leverage the architecture to develop minimum viable products (MVPS) focusing on real-time vitals aggregation, health IoT synchronisation, or long-term predictive analytics pipelines.

6.7.4. Risks and Ethical Considerations

Despite the technical strengths, critical risks must be addressed before clinical deployment. The current prototype lacks security measures such as encryption, authentication, and access control, posing significant risks to patient data privacy. Moreover, while failover mechanisms were evaluated, a more robust high-availability configuration, such as clustered Kafka brokers and fully replicated database nodes, is essential to ensure uninterrupted operation in healthcare-critical environments.

6.7.5. Feasibility and Impact

Performance testing demonstrates that the system is both theoretically robust and practically feasible in real-time environments. With modest engineering investment, it could be transformed into a production-grade tool capable of improving clinician efficiency, enhancing research data availability, and expanding access through cost-effective deployment strategies. Although further work is needed to meet clinical-grade standards, the project lays a strong technological foundation with substantial potential to positively impact healthcare delivery, biomedical research, and commercial innovation.

7. Conclusion

This dissertation presented a methodology for evaluating the ingestion performance of several time-series and relational databases within a real-time ICU heart rate monitoring system. The system was developed over six months using Apache Kafka for controlled telemetry streaming, a master-standby failover architecture, real-time visualisation dashboards, and containerised deployment. InfluxDB, TimescaleDB, VictoriaMetrics, and PostgreSQL databases were assessed for ingestion throughput, latency, and system resource usage under varying load conditions.

Benchmarking results indicated that InfluxDB consistently achieved the highest ingestion rates and the lowest write latency, although with higher memory consumption. TimescaleDB demonstrated exceptional memory efficiency and SQL compatibility, while VictoriaMetrics showed excellent ingestion scalability and low resource usage. These findings are consistent with previous benchmarking studies [4], [5], which identified VictoriaMetrics and TimescaleDB as strong candidates for large-scale telemetry ingestion and SQL-based analytics, respectively.

PostgreSQL, previously used in the supervisor’s ICU telemetry system [6], exhibited adequate performance at lower ingestion rates but showed increased latency and resource demands under higher loads, highlighting limitations for real-time critical care environments.

Given the real-time requirements of ICU telemetry systems, where data latency directly impacts clinical decision-making, prioritising ingestion speed and responsiveness is essential. InfluxDB, despite its higher memory footprint, demonstrated superior ingestion performance, resilience under concurrent load, and minimal latency. Based on the experimental results, InfluxDB is recommended as the optimal database solution for future ICU telemetry architectures requiring high-frequency, low-latency data processing.

Future work should extend the evaluation methodology to cover long-term query performance, storage efficiency, and the integration of clinical alerting systems. The architecture and benchmarking approach developed are also applicable to broader IoT telemetry scenarios where real-time ingestion performance is critical.

References

[1] 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.

[2] R. Hagan, C. J. Gillan and M. Shyamsundar, "Applying AI to manage acute and chronic clinical conditions," in *Technologies and Applications for Big Data Value*, E. Curry, S. Auer, A. J. Berre, A. Metzger, M. S. Perez, and S. Zillner, Eds. Cham: Springer, 2022, pp. 203-223, doi: 10.1007/978-3-030-78307-5_10.

[3] 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.

[4] L. K. Visperas and Y. Chodpathumwan, "Time-Series Database Benchmarking Framework for Power Measurement Data," in *2021 Research, Invention, and Innovation Congress (RI2C)*, IEEE, 2021, pp. 25-30, doi: 10.1109/RI2C51727.2021.9559822.

[5] I. Astrova, A. Koschel, S. T. Alsleben, J. T. Bellok, N. Meyer, and S. Meyer, "How to Select Time Series Databases for an Insurance Company," in *2023 14th International Conference on Information, Intelligence, Systems & Applications (IISA)*, IEEE, 2023, pp. 1-6, doi: 10.1109/IISA59645.2023.10345871.

[6] C. J. Gillan, A. Novakovic, A. H. Marshall, M. Shyamsundar and D. S. Nikolopoulos, "Expediting assessments of database performance for streams of respiratory parameters," *Computers in Biology and Medicine*, vol. 100, pp. 186-195, Sept. 2018, doi: 10.1016/j.compbiomed.2018.06.002.

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

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

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

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

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

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

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

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

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

[16] 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.

Appendices

Full experimental results, including additional graphs, raw output data, performance benchmarks, and complete logs, are available online.

They are hosted on:

- **GitHub (public)**: https://github.com/DOMINIC471/Final-Year-Project
- **GitLab (private)**: https://gitlab.eeecs.qub.ac.uk/40352799/Final-Year-Project

Here are some of the csv files examples:

DB	Rate	Avg_CPU%	Max_CPU%	Avg_Mem_MB	Max_Mem_MB
baseline	10	0.129	0.630	20.241	20.680
baseline	20	0.073	0.360	20.294	20.710
baseline	40	0.096	0.550	20.738	21.480
baseline	60	0.187	1.410	21.008	22.350
baseline	80	0.138	0.410	20.947	21.550
baseline	100	0.148	0.410	20.968	21.600
baseline	200	0.269	0.740	21.117	21.680
baseline	500	0.576	1.610	21.290	22.610
baseline	1000	1.279	4.830	21.636	22.400
influxdb	10	0.159	0.860	176.122	180.100
influxdb	20	0.189	2.190	174.447	174.700
influxdb	40	0.236	0.890	183.352	190.600
influxdb	60	0.284	2.900	190.013	190.400
influxdb	80	0.385	6.470	190.210	190.600
influxdb	100	0.329	0.690	190.860	191.900
influxdb	200	0.915	6.770	192.600	194.700
influxdb	500	1.471	8.760	191.645	194.900
influxdb	1000	2.941	11.170	193.298	197.100
timescaledb	10	1.329	3.650	77.347	78.340
timescaledb	20	0.107	1.830	77.458	78.210
timescaledb	40	0.113	0.530	77.381	77.800
timescaledb	60	0.143	0.530	77.431	77.880

timescaledb	80	1.231	2.290	77.838	78.430
timescaledb	100	0.628	1.740	77.689	78.320
timescaledb	200	1.166	5.630	77.948	78.550
timescaledb	500	0.845	1.720	77.968	78.520
timescaledb	1000	1.578	6.800	78.392	79.000
victoriametrics	10	0.696	1.420	107.786	130.600
victoriametrics	20	0.823	2.600	117.846	135.400
victoriametrics	40	0.715	2.290	119.538	145.800
victoriametrics	60	0.747	1.810	112.472	133.800
victoriametrics	80	1.053	5.050	122.606	147.300
victoriametrics	100	1.087	5.920	122.049	143.500
victoriametrics	200	1.007	6.770	122.850	140.000
victoriametrics	500	1.449	3.560	128.381	149.900
victoriametrics	1000	2.391	19.560	122.919	145.100

Table 15. Global Resource Usage Summary Across Databases and Message Rates

DB	Rate	First_CPU%	First_Mem_MB	Avg_CPU%	Max_CPU%	Avg_Mem_MB	Max_Mem_MB	Messages_Logged	Expected_Messages	Gap
baseline	10	93.8	116.280	0.069	0.100	116.468	116.470	50	50	0
baseline	20	89.7	113.950	0.084	0.100	114.319	114.390	100	100	0
baseline	40	91.5	120.280	0.158	0.300	120.670	120.770	199	200	1
baseline	60	97.2	113.250	0.248	0.400	113.668	113.700	298	300	2
baseline	80	94.6	123.640	0.326	0.500	123.810	123.830	397	400	3
baseline	100	95.4	123.860	0.408	0.700	124.224	124.270	493	500	7

baseline	200	95.6	123.890	0.785	1.400	124.359	124.380	979	1000	21
baseline	500	97.4	114.950	1.866	3.100	115.209	115.250	2391	2500	109
baseline	1000	97	113.950	3.314	5.300	114.438	114.450	4390	5000	610
influxdb	10	90	118.360	0.084	0.100	118.729	118.750	50	50	0
influxdb	20	97.5	122.360	0.074	0.100	122.725	122.800	100	100	0
influxdb	40	93.8	116.840	0.157	0.300	88.140	117.090	199	200	1
influxdb	60	90.1	114.340	0.232	0.400	114.675	114.700	298	300	2
influxdb	80	98.8	123.660	0.319	0.500	124.102	124.120	396	400	4
influxdb	100	100	122.420	0.383	0.600	122.771	122.810	493	500	7
influxdb	200	99.2	116.220	0.750	1.300	116.680	116.700	981	1000	19
influxdb	500	99.1	114.550	1.985	3.000	115.001	115.030	2344	2500	156

influxdb	1000	99.7	126.810	3.301	5.000	127.215	127.230	4486	5000	514
timescaledb	10	99.1	126.220	0.082	0.100	126.588	126.620	50	50	0
timescaledb	20	100	115.980	0.070	0.100	116.423	116.480	100	100	0
timescaledb	40	98.2	116.250	0.162	0.200	116.565	116.620	199	200	1
timescaledb	60	90.6	128.470	0.227	0.400	128.772	128.810	298	300	2
timescaledb	80	95.2	116.410	0.316	0.600	116.535	116.550	396	400	4
timescaledb	100	96.6	116.390	0.395	0.700	116.395	116.670	494	500	6
timescaledb	200	96.8	116.080	0.738	1.300	116.543	116.560	974	1000	26
timescaledb	500	99.9	124.390	1.793	2.800	124.683	124.700	2323	2500	177
timescaledb	1000	99.1	115.670	3.228	5.000	116.154	116.170	4324	5000	676
victoriametrics	10	98.1	114.640	0.084	0.100	114.750	114.750	50	50	0

victoriametrics	20	100	112.480	0.076	0.100	112.732	112.770	100	100	0
victoriametrics	40	90	118.170	0.152	0.300	111.924	118.420	199	200	1
victoriametrics	60	96.6	123.120	0.222	0.400	123.585	123.640	298	300	2
victoriametrics	80	98.8	127.450	0.304	0.500	127.798	127.830	395	400	5
victoriametrics	100	100	123.120	0.372	0.600	123.558	123.610	492	500	8
victoriametrics	200	99	117.520	0.725	1.300	117.894	117.940	969	1000	31
victoriametrics	500	98.2	116.200	1.690	2.900	116.675	116.690	2313	2500	187
victoriametrics	1000	93.7	115.340	3.105	5.600	115.802	115.810	4317	5000	683

Table 16. Per-Message Resource Usage and Message Gap Summary Across Databases and Message Rates

DB	**Rate**	**Messages_Written**	**Duration_s**	**Writes_per_sec**	**Avg_Latency_ms**
baseline	10	50	293.047	0.171	5980.550
baseline	20	100	258.588	0.387	2612.010

baseline	40	199	262.222	0.759	1324.350
baseline	60	298	260.905	1.142	878.467
baseline	80	397	262.209	1.514	662.145
baseline	100	493	260.503	1.892	529.477
baseline	200	979	262.198	3.734	268.096
baseline	500	2391	261.043	9.159	109.223
baseline	1000	4390	260.568	16.848	59.368
influxdb	10	50	256.551	0.195	5235.730
influxdb	20	100	259.149	0.386	2617.670
influxdb	40	199	261.612	0.761	1321.270
influxdb	60	298	261.575	1.139	880.725
influxdb	80	396	260.407	1.521	659.258
influxdb	100	493	262.008	1.882	532.537
influxdb	200	981	260.639	3.764	265.958
influxdb	500	2344	262.405	8.933	111.995
influxdb	1000	4486	263.372	17.033	58.723
timescaledb	10	50	256.649	0.195	5237.740
timescaledb	20	100	259.390	0.386	2620.100
timescaledb	40	199	261.973	0.760	1323.100
timescaledb	60	298	261.013	1.142	878.831
timescaledb	80	396	261.702	1.513	662.536
timescaledb	100	494	260.599	1.896	528.599
timescaledb	200	974	261.511	3.725	268.768
timescaledb	500	2323	262.338	8.855	112.979
timescaledb	1000	4324	262.619	16.465	60.749
victoriametrics	10	50	257.358	0.194	5252.210

victoriametrics	20	100	258.743	0.386	2613.560
victoriametrics	40	199	261.269	0.762	1319.540
victoriametrics	60	298	262.988	1.133	885.480
victoriametrics	80	395	260.776	1.515	661.869
victoriametrics	100	492	261.379	1.882	532.340
victoriametrics	200	969	261.773	3.702	270.427
victoriametrics	500	2313	262.535	8.810	113.553
victoriametrics	1000	4317	262.351	16.455	60.786

Table 17. Write Throughput and Latency Summary Across Databases and Message Rates