

DataFlow

A Project Report

Submitted in partial fulfilment of the
Requirement for award of the degree of

BACHELOR OF SCIENCE (COMPUTER SCIENCE)

Submitted By

MOHOMMAD SHAHID SHARIF SHAIKH

SEAT NUMBER: CS23054

Under the esteemed guidance of

Mrs. Mayra Lachhani

HOD & Assistant Professor



DEPARTMENT OF SCIENCE IN COMPUTER SCIENCE

S.S.T. COLLEGE OF ARTS & COMMERCE

(Affiliated to University of Mumbai)

ULHASNAGAR, 421004

MAHARASHTRA

2025-2026

S.S.T. COLLEGE OF ARTS & COMMERCE

(Affiliated to University of Mumbai)

ULHASNAGAR-MAHARASHTRA-421004

DEPARTMENT OF SCIENCE IN COMPUTER SCIENCE



CERTIFICATE

This is certify that the project entitled, “ **DataFlow** ”, is bonafide work of **MOHOMMAD SHAHID SHARIF SHAIKH** bearing Seat No. **CS23054** submitted in partial fulfilment of the requirements for the award of degree of BACHELOR OF SCIENCE in COMPUTER SCIENCE from University of Mumbai.

Internal Guide

Coordinator

External Examiner

Date:

College Seal

PROFORMA FOR THE APPROVAL PROJECT PROPOSAL

(Note: All entries of the proforma of approval should be filled up with appropriate and complete information. Incomplete proforma of approval in any respect will be summarily rejected.)

PNR No.**Roll no: CS23054**

1. **Name of the Student :** MOHOMMAD SHAHID SHARIF SHAIKH
2. **Title of the Project:** DataFlow
3. **Name of the Guide:** Mrs. Mayra Lachhani
4. **Teaching experience of the Guide** _____
5. **Is this your first submission?** Yes ☐ No ☐

Signature of the Student**Signature of the Guide****Date:****Date:****Signature of the coordinator****Date:**

Abstract

This project presents a **scalable, fault-tolerant, and intelligent real-time data pipeline system** designed to efficiently manage continuous IoT data streams. The system employs **Redis** as a high-speed in-memory buffer for temporary data handling and **Supabase** as the persistent storage layer for structured, reliable, and queryable data management. By decoupling buffering and storage, the pipeline ensures smooth data ingestion, consistent throughput, and reliable storage even under fluctuating loads.

To enhance scalability and fault tolerance, the system supports **multiple Redis instances**, which can be **added or removed dynamically** through an integrated management dashboard. A **round-robin load-balancing mechanism** distributes data evenly across available Redis instances, automatically bypassing any instance that becomes unreachable. This self-healing design minimizes downtime and ensures continuous data flow without manual intervention.

Each component of the system is modular and interactive. The **Bridge Server** includes a dedicated dashboard that displays connected Redis instances, their health status, and active load metrics, while allowing administrators to modify configurations in real-time. The **Data Validation Server** focuses on data verification and transformation before it is loaded into the database. Instead of a traditional dashboard, it features an **AI-powered agent** capable of interpreting user queries, analyzing data stored in Supabase, and generating meaningful summaries and insights. This enables users to interact naturally with the system and obtain contextual information rather than raw values.

The overall architecture combines **distributed in-memory processing, asynchronous FastAPI microservices, and intelligent AI analytics**, creating a data pipeline that is **scalable, resilient, and user-friendly**. This approach makes it ideal for **real-time IoT and analytics applications**, where continuous availability, efficient data handling, and actionable insights are critical for informed decision-making.

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our project guide, **Mrs. Mayra Lachhani**, for their invaluable guidance, encouragement, and support throughout the development of this project. Their expertise and constructive feedback were instrumental in helping us achieve our objectives.

We are also grateful to **Mrs. Mayra Lachhani** and the faculty members of the DEPARTMENT OF SCIENCE IN COMPUTER SCIENCE

S.S.T. COLLEGE OF ARTS & COMMERCE for providing us with the resources and knowledge required to successfully complete this work.

Our heartfelt thanks go to our friends and classmates for their constant motivation and collaboration, which made this journey more productive and enjoyable. Finally, we would like to thank our families for their continuous encouragement and support..

DECLARATION

I, **MOHOMMAD SHAHID SHARIF SHAIKH** , Roll No. CS23054, student of DEPARTMENT OF SCIENCE IN COMPUTER SCIENCE, **S.S.T. COLLEGE OF ARTS & COMMERCE**, hereby declare that the project titled “**DataFlow**” is my original work carried out under the guidance of **Mrs. Mayra Lachhani**.

This project report has not been submitted to any other university or institution for the award of any degree, diploma, or certification. The work presented here is a result of my own efforts, and all sources of information used in the project have been duly acknowledged.

I take full responsibility for the authenticity of the content of this report.

Name and Signature of the Students

**MOHOMMAD SHAHID SHARIF
SHAIKH**

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION		11
	1.1	Background	11
	1.2	Problem Statement	12
	1.3	Objectives	13
	1.4	Scope of the Project	14
	1.5	Key Features	15
	1.6	System Architecture Overview	15
	1.7	Advantages	16
CHAPTER 2	SURVEY OF TECHNOLOGIES		18
	2.1	Introduction	18
	2.2	Survey of Existing Systems	18
	2.3	Survey of Technologies Used	19
	2.4	Comparison and Justification of Chosen Technologies	22
	2.5	Summary	23
CHAPTER 3	Methodology / System Design		24

	3.1	System Architecture Overview	25
	3.2	Component-wise Design	26
	3.3	Data Flow and Interaction	27
	3.4	Summary	29
CHAPTER 4		Implementation / System Development	29

	4.1	Introduction	29
	4.2	Development Environment	29
	4.3	Implementation of Each Layer	30
	4.4	Integration and Workflow	41
	4.5	Challenges and Solutions	42
CHAPTER 5		Testing and Evaluation	43
	5.1	Introduction	43
	5.2	Testing Environment	43
	5.3	Functional Testing	43
	5.4	Performance Testing	44

	5.5	Error Handling and Fault Tolerance	44
	5.6	Test Results	44
	5.7	Summary	44
CHAPTER 6		Conclusion and Future Work	45
	6.1	Conclusion	45
	6.2	Limitations	45
	6.3	Future Work	46
CHAPTER 7		References & Bibliography	48

LIST OF FIGURES

2.1	Survey of Technologies Comparison	24
3.1	High-Level System Architecture of the Proposed Real-Time IoT Data Pipeline	25
3.2	High-level data flow of the IoT pipeline, showing data movement from IoT devices to dashboard visualization.	27
4.1	Simulated Iot Device Data Output	31
4.2	Mqtt calls Output (When we get an Mqtt Msg)	32
4.3	Storing in redis instances decided by round robin Output	33
4.4	Output of Dynamically storing Iot data in Supabase	39
4.5	Iot Data History Table	40
4.6	Data Validation Model Schema Table	41

CHAPTER 1

INTRODUCTION

In today's **digital landscape**, organizations and individuals are generating **data at an unprecedented rate**. Efficiently **managing, storing, and analyzing continuous data streams** has become a significant challenge.

This project, titled **DataFlow**, presents an **intelligent real-time data pipeline system** designed to handle **continuous data streams** efficiently while ensuring **reliability, fault tolerance, and actionable insights** for end users.

The system integrates **Redis** as a **high-speed in-memory buffer** to manage incoming data, ensuring **smooth and uninterrupted ingestion** even under heavy load. **Supabase** serves as the **persistent storage layer**, enabling **structured, reliable, and secure data management**.

The **Bridge Server** distributes incoming data across multiple Redis instances using a **round-robin load-balancing algorithm**, automatically bypassing failed nodes for **fault-tolerant operation**. Administrators can **add or remove Redis instances dynamically** via an **interactive dashboard**, enabling seamless scaling.

The **Data Validation Server** processes and validates the buffered data before storing it in Supabase. It allows **dynamically defining data models** for flexible validation of different data formats, ensuring that only **clean, structured, and reliable data** is persisted.

On top of this pipeline, an **AI agent** interacts with users to **analyze and summarize stored data**, providing **concise and actionable insights** instead of raw information. This intelligent layer simplifies user interaction and **enhances decision-making**.

By combining **asynchronous FastAPI microservices, high-speed buffering, reliable cloud storage, dynamic data validation, and AI-driven analytics**, the **DataFlow** system demonstrates a **scalable, resilient, and intelligent approach** to managing **real-time data pipelines**.

1.1 Background

With the rapid growth of **digital applications**, data is being generated continuously from various sources, such as **sensors, web applications, and IoT devices**. Efficiently **handling and processing this data** is critical for **system performance, reliability, and timely decision-making**. Traditional methods of data management, such as **batch processing** or **manual analysis**, are often slow, error-prone, and incapable of meeting the demands of **real-time applications**.

Modern **data pipelines** address these challenges by providing structured mechanisms for **data ingestion, buffering, processing, and storage**. By integrating **high-speed in-memory**

storage like Redis with **cloud-compatible databases such as Supabase**, the system can efficiently handle **high-throughput data streams**.

The **Bridge Server** ensures **even distribution of data across multiple Redis instances** using a **round-robin load-balancing approach**, while automatically bypassing any failed nodes for **fault-tolerant operation**. Each Redis instance can be **added or removed dynamically** via an **interactive dashboard**, allowing administrators to scale the system seamlessly.

The **Data Validation Server** processes and validates the buffered data before storing it in the database. It allows for **dynamically defining data models**, making the system adaptable to **changing data formats and validation rules**. This ensures that only **clean, structured, and reliable data** is persisted in Supabase.

On top of this pipeline, an **AI agent** interacts with users to **analyze and summarize the stored data**, transforming raw streams into **actionable insights**. This intelligent layer **reduces user effort**, supports faster **decision-making**, and makes the system **user-friendly**. This project demonstrates a **resilient, scalable, and modular data pipeline architecture**, showing how distributed components — including **Bridge Server, Data Validation Server, Redis, and AI agent** — can be orchestrated to meet the requirements of **real-time, high-throughput applications**.

1.2 Problem Statement

In the modern IoT ecosystem, devices continuously generate massive amounts of data that must be processed, validated, and stored in real time. Traditional systems often rely on batch-oriented or monolithic architectures, which are unable to keep up with the high velocity, volume, and variability of IoT data. These approaches introduce latency, risk data loss during network fluctuations, and struggle to scale dynamically as new devices or data sources are added.

Furthermore, most existing data pipelines lack intelligent mechanisms for fault tolerance, real-time validation, and automated data summarization. Failures in one component can disrupt the entire pipeline, while unvalidated or inconsistent data leads to unreliable analytics and decision-making. Users also face challenges in interpreting large volumes of raw data without analytical support.

The DataFlow system addresses these challenges by introducing a modular, asynchronous, and fault-tolerant real-time data pipeline. It employs a FastAPI-based Bridge Server for distributed ingestion through MQTT, Redis for high-speed buffering, and Supabase for structured storage. A Data Validation Server allows dynamic model definitions to ensure data consistency, while an integrated AI Agent summarizes and interprets stored information for the user. This design overcomes the limitations of conventional data handling approaches, providing real-time, reliable, and scalable IoT data management with intelligent insight generation.

1.3 Objectives

The main objectives of this project are:

1. **Efficient Data Handling:**
To design a **real-time data pipeline** that can **ingest, buffer, and store continuous streams of data** with **minimal latency**.
2. **Reliable and Structured Storage:**
To implement **structured, secure, and persistent storage** using Supabase, ensuring **data integrity, consistency, and easy retrieval**.
3. **Scalability and Fault Tolerance:**
To enable **seamless addition or removal of multiple Redis instances** for **scaling the pipeline**, and to **automatically reroute data in case of instance failures**, ensuring **continuous operation**.
4. **Load Balancing:**
To implement a **round-robin load-balancing mechanism** that evenly distributes incoming data across Redis instances, **optimizing throughput and preventing bottlenecks**.
5. **Dynamic Data Validation through Model Definition:**
To allow **administrators to define custom validation models** dynamically, ensuring incoming data is **accurate, complete, and consistent** before storage.
6. **Intelligent Data Summarization:**
To integrate an **AI agent** that **analyzes and summarizes stored data**, presenting **concise, actionable insights** to users and reducing cognitive load.
7. **User-Friendly Design:**
To develop a system that **requires minimal configuration and intervention**, providing **interactive dashboards** and easy access to insights without exposing underlying complexity.
8. **Real-Time Application Suitability:**
To demonstrate a pipeline capable of handling **high-throughput, real-time data streams** efficiently and reliably.
9. **Monitoring and Administration:**
To provide **dashboards for both Bridge Server and Data Validation Server**, allowing administrators to **monitor data flow, instance health, and system performance** in real time.
10. **Modular and Extensible Architecture:**
To design a **flexible, modular pipeline** where components like **Redis instances, validation models, and AI analytics** can be extended or updated without disrupting the system.

1.4 Scope of the Project

The **DataFlow** system is designed to handle **real-time IoT data streams** efficiently, providing a **scalable, reliable, and intelligent pipeline** for processing and analyzing continuous data. The key capabilities within the scope of the project include:

1. **Real-Time IoT Data Ingestion:**
Continuously receives and processes data from **sensors and embedded devices**, ensuring timely availability for downstream processing.
2. **Dynamic Data Validation and Structuring:**
Validates incoming data through the **Data Validation Server**, allowing **dynamic definition of data models** to ensure accuracy, consistency, and completeness before storage.
3. **High-Speed Buffering and Load Balancing:**
Uses **Redis** as an **in-memory buffer** to manage bursts of data, with a **round-robin load-balancing mechanism** for even distribution across multiple Redis instances.
4. **Reliable and Secure Storage:**
Persists validated data in **Supabase**, ensuring **structured storage, integrity, and security** against unauthorized access.
5. **Scalability and Fault Tolerance:**
Supports **dynamic addition and removal of Redis instances**. Data is automatically rerouted in case of node failures, enabling **continuous operation**.
6. **Intelligent Data Summarization:**
Integrates an **AI agent** that analyzes stored data and generates **concise, actionable insights**, reducing user effort and simplifying decision-making.
7. **User-Friendly Dashboard:**
Provides **real-time visualizations, system monitoring, and control** over Redis instances and pipeline performance.
8. **Real-Time Application Suitability:**
Ideal for **IoT monitoring systems, smart devices, and analytics platforms** requiring **low-latency, high-throughput processing**.

Limitations (within scope)

- Focuses on **structured IoT data**; predictive analytics or complex machine learning models are not included.
- Performance is dependent on the **limitations of free-tier cloud services** (Redis, Supabase) and network connectivity.
- Physical IoT devices were **simulated using Python scripts** due to unavailability during demonstrations.

1.5 Key Features

- **Real-Time IoT Data Ingestion:**
Continuously receives and processes **data streams from IoT devices**, ensuring **minimal latency** and timely availability. The **Bridge Server** efficiently routes incoming data to the buffering layer.
- **Dynamic Backend Data Validation and Structuring:**
Validates incoming data through the **Data Validation Server**, allowing **dynamic definition of data models**. Ensures data is **accurate, complete, and correctly formatted** before storage.
- **High-Speed Buffering with Redis:**
Temporarily stores incoming data in **Redis** to handle bursts and ensure **smooth ingestion**. Implements **round-robin load balancing** across multiple Redis instances for optimal throughput.
- **Reliable and Secure Storage with Supabase:**
Persists **structured data securely**, maintaining **data integrity, consistency, and privacy**. Implements **access controls** and secure storage protocols to protect against unauthorized access.
- **Automatic Scaling and Fault Tolerance:**
Supports **dynamic addition or removal of Redis instances**, automatically rerouting data during instance failures to maintain **continuous and reliable operation**.
- **AI-Based Data Summarization:**
An **AI agent** analyzes stored data and generates **concise, actionable insights**, reducing user effort and simplifying decision-making.
- **User-Friendly Dashboard:**
Displays **real-time visualizations**, system status, and summaries of IoT data. Administrators can **monitor, manage, and dynamically scale Redis instances**.
- **Cloud-Compatible and Modular Design:**
The system is **modular, extensible, and cloud-ready**, allowing integration of additional IoT devices, analytics modules, or new validation models with minimal effort.

1.6 System Architecture Overview

The system architecture is designed as a modular and scalable pipeline to manage IoT data efficiently, securely, and reliably. Each component plays a specific role in ensuring smooth real-time data flow and intelligent insight generation.

1. **IoT Devices:** Sensors or embedded systems generate continuous streams of data. For demonstration, these devices are simulated using Python scripts that publish data to the bridge server via MQTT.
2. **Bridge Server:** Serves as an intermediary that receives data from IoT devices and forwards it to Redis through MQTT. This design decouples the data source from backend processing, improving scalability and maintainability.

3. **Redis Buffer:** Acts as a high-speed in-memory buffer that temporarily stores incoming IoT data. It efficiently handles data bursts and ensures asynchronous data access for downstream services.
4. **Data Structure and Validation Server:** Retrieves data from Redis, validates its accuracy, completeness, and format, and structures it before passing it on for storage. This ensures that only clean and reliable data is persisted.
5. **Supabase Database:** Provides persistent and structured storage for validated IoT data. Supabase ensures data integrity, easy retrieval, and smooth integration with analytics and visualization tools.
6. **AI Agent:** Analyzes stored data to generate concise summaries and actionable insights. This reduces the need for manual data interpretation and enables quick decision-making.
7. **Dashboard Interface:** Displays real-time data visualizations, trends, and AI-generated summaries. It allows users to monitor IoT system performance and make informed decisions easily.
8. **Scaling and Fault Tolerance Layer:** Supports multiple Redis instances for horizontal scaling. In case of an instance failure, data is automatically rerouted, ensuring uninterrupted operation and system resilience.

Data Flow:

IoT Devices → Bridge Server (via MQTT) → Redis Buffer → Data Structure & Validation Server → Supabase Storage → AI Agent → Dashboard → User

This architecture enables real-time processing, reliability, scalability, and intelligent data summarization while maintaining secure and efficient data handling—making it suitable for a wide range of IoT and data-driven applications.

1.7 Advantages

1. **Real-Time IoT Data Handling:**
Efficiently **ingests and processes continuous data streams** from multiple IoT devices with **minimal latency**.
2. **Reliable Data Validation and Structuring:**
Validates incoming data via the **Data Validation Server**, allowing **dynamic model definition**. Ensures data is **accurate, complete, and correctly formatted** before storage, improving reliability.
3. **High-Speed Buffering with Redis:**
Handles bursts of incoming data using **Redis as an in-memory buffer**, preventing data loss during peak loads and ensuring **smooth data flow**.
4. **Persistent and Secure Storage:**
Structured IoT data is securely stored in **Supabase**, maintaining **data integrity, confidentiality, and access control**.

5. **AI-Powered Data Summarization:**

An **AI agent** analyzes stored data and generates **concise, actionable insights**, reducing the need to manually process raw data.

6. **User-Friendly Dashboard:**

Provides **real-time visualizations, system monitoring, and summaries**, enabling users to **interpret IoT data and make informed decisions efficiently**.

7. **Scalability:**

Supports **multiple Redis instances** and additional IoT devices, allowing the system to **scale seamlessly** without major redesign.

8. **Fault Tolerance:**

Automatic **rerouting of data** and **buffering mechanisms** ensure **continuous operation**, even in case of Redis node or component failures.

9. **Modular and Extendable Architecture:**

Pipeline components are **individually upgradable and extendable**, allowing easy integration of future enhancements such as **predictive analytics, mobile dashboards, or end-to-end encryption**.

Chapter 2

Survey of Technologies

2.1 Introduction

This chapter provides an overview of the technologies used in developing the proposed real-time ETL (Extract, Transform, Load) data pipeline for IoT systems. The system extracts data from IoT devices using the MQTT protocol and transfers it to a Redis buffer through a FastAPI-based Bridge Server. Redis serves as a high-speed temporary store, ensuring smooth and asynchronous data flow between devices and backend services.

The FastAPI-based Validation Servers then retrieve, process, and validate the buffered data before storing it securely in a Supabase database for analysis and visualization. Each component—MQTT, FastAPI, Redis, and Supabase—was selected for its scalability, performance, and ability to support real-time operations.

This chapter discusses these core technologies and the rationale behind their integration in the proposed system.

2.2 Survey of Existing Systems

2.2.1 Traditional Batch Processing Systems

- Earlier data pipelines often used **batch processing**, where data is collected over time and processed in bulk (hourly, daily).
- **Limitations:** High latency, cannot handle real-time IoT data, prone to data loss during bursts, no dynamic validation.

2.2.2 Cloud-Based IoT Platforms

- Examples: **AWS IoT Core**, **Azure IoT Hub**, **Google Cloud IoT**
- These platforms provide real-time ingestion, storage, and some analytics.
- **Limitations:** Expensive, less customizable, complex setup for small-scale or self-hosted solutions.

2.2.3 Distributed Streaming Systems (Kafka & Zookeeper)

- **Apache Kafka:** High-throughput distributed streaming platform used to publish and subscribe to streams of records in real time.
- **Zookeeper:** Coordinates distributed components (like Kafka brokers) for fault tolerance and leader election.
- **Advantages:** Scalable, supports high-frequency streams, reliable message delivery.

- **Limitations:** Requires heavy setup, complex configuration, no built-in AI summarization, no dynamic data validation dashboards — more suitable for enterprise setups.

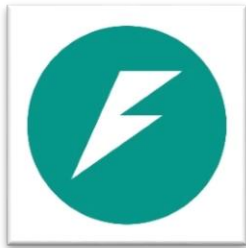
2.2.4 Limitations of Existing Systems

- Lack of integrated AI for automatic summarization.
- No simple interface for **dynamic model-based validation**.
- Complex scaling and fault-tolerance setup (Kafka/Zookeeper require manual cluster management).
- Expensive for small to medium-scale IoT projects.

2.3 Survey of Technologies Used

This section discusses the key technologies used in the **DataFlow** system, highlighting their roles, advantages, and reasons for selection. Each technology is chosen to ensure **scalability, reliability, real-time processing, and ease of integration** for IoT data pipelines.

2.3.1 FastAPI



- **Overview:** FastAPI is a modern, asynchronous Python web framework for building APIs.
- **Role in DataFlow:** Implements the **Bridge Server** and **Data Validation Server**, handling data ingestion, validation, and API communication asynchronously.
- **Why Chosen:** Lightweight, fast, supports concurrency, easy to integrate with MQTT, Redis, and AI modules.

2.3.2 MQTT Broker



- **Overview:** HiveMQ is a scalable, reliable MQTT broker designed for real-time messaging between IoT devices and backend systems.
- **Role in DataFlow:** Serves as the central message broker, receiving data from IoT devices and forwarding it to the Bridge Server, ensuring low-latency and reliable delivery of messages.
- **Why Chosen:** Supports high-throughput IoT messaging, is compatible with MQTT protocol, provides easy integration with backend services, and ensures reliable message delivery even under variable network conditions.

2.3.3 Redis



- **Overview:** Redis is an in-memory key-value store providing high-speed access.
- **Role in DataFlow:** Serves as a buffer layer, temporarily storing incoming IoT data to handle bursts and enable smooth downstream processing.
- **Why Chosen:** Extremely fast, supports multiple instances for scaling, and allows fault-tolerant operation with automatic rerouting.

2.3.4 Supabase



- **Overview:** Supabase is a cloud-based platform providing PostgreSQL databases with RESTful APIs.
- **Role in DataFlow:** Stores validated and structured IoT data for persistent storage, analysis, and visualization.
- **Why Chosen:** Structured storage, easy integration, cloud-ready, minimal setup effort.

2.3.5 AI Agent



Overview: Gemini is an AI platform capable of processing and analyzing data, providing natural language summaries, and generating actionable insights.

Role in DataFlow: Acts as the **intelligent layer** that reads validated IoT data from the database, analyzes it, and generates concise summaries and insights for users. It also communicates with the web dashboard to present results.

Why Chosen: Offers reliable AI-driven analytics, reduces manual data interpretation, integrates easily with backend systems, and provides natural language summarization for better user understanding.

2.3.6 Dashboard (Web Interface)

- **Overview:** A web-based interface for real-time monitoring and control.
- **Role in DataFlow:** Displays data visualizations, system status, and allows management of Redis instances and pipeline performance.
- **Why Chosen:** User-friendly, responsive, integrates easily with FastAPI APIs.

2.3.7 Cloud Hosting



- **Overview:** Render is a cloud platform for hosting web applications, APIs, and databases with automatic deployment and scaling.
- **Role in DataFlow:** Hosts the Bridge Server, Data Validation Server, and Web dashboard, enabling online access and continuous operation of the data pipeline.
- **Why Chosen:** Provides easy deployment, scalability, and reliability, with minimal infrastructure management, allowing the pipeline to run smoothly without manual server maintenance.

2.3.8 REST API



Overview: REST (Representational State Transfer) APIs allow communication between different components over HTTP using standard methods like GET, POST, PUT, and DELETE.

Role in DataFlow:

- Enables the **Bridge Server** and **Data Validation Server** to interact with Supabase for storing and retrieving IoT data.
- Allows the **web dashboard** to fetch processed data and AI-generated summaries in real-time.

Why Chosen:

- Simple, widely used, and compatible with cloud databases like Supabase.
- Enables modular design where backend services and front-end dashboards can operate independently.

2.4 Comparison and Justification of Chosen Technologies

Purpose: Explain why you chose each technology for DataFlow, highlighting how it addresses limitations of existing systems.

Suggested structure:

1. HiveMQ vs Kafka/Zookeeper
 - HiveMQ: Lightweight, easy setup, works well for IoT devices.
 - Kafka/Zookeeper: Complex, resource-heavy, overkill for small-to-medium IoT pipelines.
2. Redis for Buffering
 - Fast, in-memory storage for handling bursts.
 - Chosen over other in-memory stores (like Memcached) for ease of scaling, pub/sub support, and persistence options.
3. Supabase for Persistent Storage
 - Cloud-based, structured, and easy to integrate via REST API.
 - Chosen over Firebase or DynamoDB because of SQL compatibility and simplicity.
4. FastAPI for Backend Servers
 - Lightweight, asynchronous, integrates with MQTT/Redis easily.
5. Gemini AI Agent

- Provides real-time data summarization, integrates easily with your database, and supports natural language outputs.

6. Render for Deployment

- Cloud hosting simplifies deployment of servers and dashboards without managing infrastructure manually.

2.1 Survey of Technologies Choices and Benefits:

Component	Alternatives	Why Chosen	Benefits
MQTT Broker	Kafka, RabbitMQ	HiveMQ	Lightweight, IoT-friendly
Buffer	Memcached	Redis	High-speed, scalable
Storage	Firebase, DynamoDB	Supabase	SQL-based, structured, persistent
Backend	Django, Flask	FastAPI	Async, lightweight, modular
AI	Custom ML models	Gemini	Real-time summarization
Hosting	AWS, Heroku	Render	Easy deployment, scalable

2.5 Summary

The DataFlow project leverages a combination of modern technologies to ensure real-time, scalable, and fault-tolerant data handling. HiveMQ was chosen as the MQTT broker for lightweight and IoT-friendly messaging. Redis serves as a high-speed in-memory buffer, efficiently managing data bursts. Supabase provides structured, persistent storage for validated data. FastAPI enables asynchronous backend services for the Bridge Server and Data Validation Server, while Gemini powers AI-based real-time data summarization. Render was selected for cloud deployment due to its simplicity and scalability. This combination of technologies ensures an efficient, modular, and intelligent data pipeline suitable for high-throughput IoT applications.

Chapter 3

Methodology / System Design

The proposed system follows a modular and event-driven architecture designed to ensure scalability, efficiency, and real-time data processing. This chapter describes the overall system design, data flow, and components involved in the development of the IoT data pipeline. It also explains the interaction between different subsystems, covering both software and hardware perspectives.

3.1 System Architecture Overview

The proposed system follows a **modular real-time data pipeline architecture** designed to handle continuous IoT data efficiently from ingestion to intelligent analysis. The architecture integrates multiple lightweight yet powerful technologies to ensure **scalability, fault tolerance, and real-time responsiveness**.

At a high level, the system consists of five major layers — **Data Source (IoT Devices)**, **Data Ingestion (MQTT Broker)**, **Data Buffering (Redis)**, **Data Validation and Storage (FastAPI + Supabase)**, and **AI Summarization (Gemini Model)**. Each layer is responsible for a specific stage in the data flow, ensuring clear separation of concerns and easy maintainability.

1. **IoT Layer:**

The IoT Layer in this system is simulated using virtual sensor data to emulate real-world IoT devices. Each simulated node publishes readings such as temperature, humidity, or water level via MQTT to the HiveMQ broker, ensuring realistic data flow without requiring physical hardware.

2. **MQTT Broker Layer (HiveMQ):**

HiveMQ acts as the message broker for managing real-time communication between IoT clients and backend servers. It ensures reliable, low-latency message delivery and efficient topic-based message routing.

3. **Bridge Server (FastAPI + Redis):**

The Bridge Server receives MQTT messages asynchronously and temporarily stores them in Redis. Redis functions as a **buffer** to handle burst loads and ensure no data loss during high-throughput conditions. This server also applies **round-robin balancing** across validation instances.

4. **Validation and Storage Layer:**

Another FastAPI-based module retrieves buffered data from Redis, validates it according to dynamically defined data models, and stores it in **Supabase**, a PostgreSQL-backed structured database. This step ensures data integrity, consistency, and structure for further analysis.

5. **AI Analysis and Visualization Layer:**

Once validated, data is processed by an integrated **Gemini AI model** for real-time

summarization, anomaly detection, and report generation. The AI outputs are then displayed on an interactive dashboard, allowing administrators to monitor trends and system performance effectively.

Overall, the architecture ensures **end-to-end asynchronous flow**, reducing latency and improving performance. It can be easily scaled horizontally by adding more bridge or validation servers, making it suitable for large-scale IoT deployments.

Figure 3.1

High-Level System Architecture of the Proposed Real-Time IoT Data Pipeline

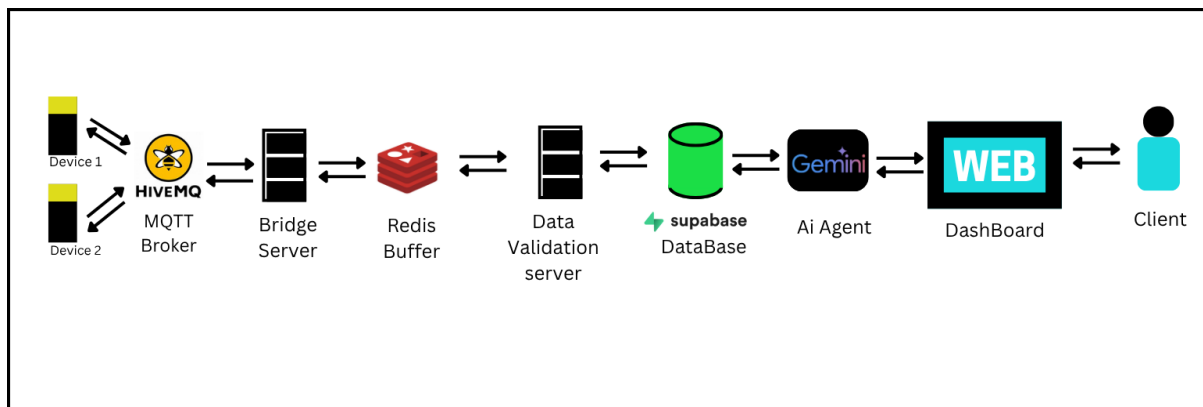


Figure Explanation:

The high-level system architecture (Figure 3.1) illustrates the complete data flow of the proposed IoT data pipeline. The process begins with IoT devices publishing sensor data to the HiveMQ broker. The Bridge Server receives and temporarily buffers the data in Redis for high-speed transfer. The Validation Server, built with FastAPI, processes and stores the verified data in Supabase. Finally, the Gemini AI model analyzes the stored data and generates real-time summaries and visual insights displayed on the dashboard.

3.2 Component-wise Design

This section details the individual components of the IoT data pipeline, highlighting their specific roles and responsibilities within the overall system.

3.2.1 IoT Devices

- Act as the data sources for the pipeline.
- Generate continuous telemetry (e.g., temperature, humidity, water level).
- For demonstration, Python scripts simulate devices and publish data via MQTT.

3.2.2 Bridge Server

- Receives MQTT messages asynchronously.
- Temporarily stores data in Redis to prevent loss during bursts.
- Implements load balancing across multiple Redis instances using a round-robin algorithm, ensuring even distribution of data and preventing any single instance from being overloaded.

3.2.3 Redis Buffer

- Serves as in-memory storage for fast access and temporary buffering.
- Smooths out data spikes or bursts from IoT devices.

- Handles data from multiple bridge servers in a round-robin fashion, supporting scalability and fault tolerance.
- Decouples data ingestion from validation and storage, allowing asynchronous and reliable data flow.

3.2.4 Data Validation Server

- Retrieves data from Redis and applies dynamic model-based validation.
- Ensures data accuracy, completeness, and consistency before storage.
- Forwards validated data to Supabase while logging invalid messages.

3.2.5 Supabase Storage

- Provides persistent, structured storage for validated data.
- SQL-based storage supports complex queries and cloud-friendly access.
- Serves as the foundation for analytics and AI processing.

3.2.6 AI Agent (Gemini)

- Processes validated data to generate summaries, insights, and anomaly detection.
- Interacts with the dashboard to provide real-time feedback and intelligence.

3.2.7 Dashboard Interface

- Visualizes live IoT data, AI outputs, and system metrics, Add/Del Instances.
- Provides controls for managing Redis nodes and monitoring system health.

3.2.8 Scaling & Fault Tolerance

- Redis nodes can be added/removed dynamically without interrupting the pipeline.
- Bridge and validation servers can scale horizontally to handle higher loads.
- Ensures robust, continuous operation under high traffic or node failures.

3.3 Data Flow and Interaction

The system operates as a continuous, event-driven pipeline, where IoT data flows seamlessly from generation to analysis in real time. Each component works asynchronously, allowing the system to handle high-throughput streams without delays or bottlenecks. Incoming data is buffered, validated, and stored efficiently, ensuring consistency and accuracy throughout the pipeline. By decoupling ingestion, processing, and storage, the architecture maintains high reliability and can scale horizontally to accommodate additional devices or increased traffic. This design enables robust performance even during sudden spikes in data, while supporting real-time insights and visualization for end users.

Figure 3.2: High-level data flow of the IoT pipeline, showing data movement from IoT devices to dashboard visualization.

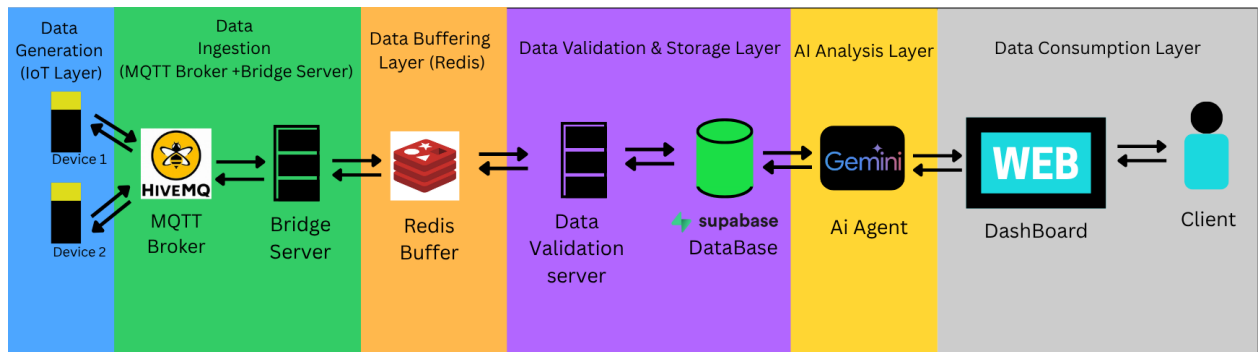


Figure Explanation:

The high-level data flow of the IoT pipeline (Figure 3.2) shows data moving from IoT devices to dashboard visualization. Sensor readings are published to HiveMQ, received by the Bridge Server, and buffered in Redis to smooth spikes and balance load. Validation servers check data against dynamic models and store verified data in Supabase. The Gemini AI agent processes the stored data to generate summaries and insights, which are displayed on the dashboard for real-time monitoring and administrative control. This modular design ensures efficient, asynchronous, and scalable processing of continuous IoT streams.

IoT Layer (Data Source)

- Highlight all device icons or simulation scripts in one box.
- Label it “IoT Devices / Data Source”.
- Represents the origin of all data, continuously producing sensor readings such as temperature, humidity, or water level.
- For demonstration, Python scripts simulate IoT devices and publish data via MQTT.

Data Ingestion Layer (MQTT Broker + Bridge Server)

- Group the MQTT broker and Bridge Server in a single section.
- Use arrows from IoT Devices → MQTT Broker → Bridge Server.
- Bridge Server responsibilities:
 - Receives messages asynchronously from MQTT.
 - Buffers messages in Redis to handle bursts and smooth data spikes.
 - Implements round-robin load balancing to distribute incoming data across multiple Redis instances.

Data Buffering Layer (Redis)

- Draw a box around all Redis instances.
- Include a note: “Buffers incoming data, smooths spikes, supports asynchronous access”.
- Provides in-memory storage for high-speed access and temporary buffering.
- Decouples ingestion from validation, ensuring reliable data flow and supporting multiple concurrent bridge or validation servers.

Data Validation & Storage Layer

- Group Validation Servers and Supabase DB together.
- Note: “Dynamic model-based validation → Structured storage in Supabase”.
- Validation Server responsibilities:
 - Retrieves data from Redis.
 - Performs dynamic model-based checks for accuracy, completeness, and consistency.

- Logs invalid or incomplete messages.
- Forwards validated data to Supabase for persistent structured storage.

AI Analysis Layer

- Show Gemini AI connected to Supabase.
- Note: “Generates summaries, insights, and anomaly detection”.
- Processes stored data to produce real-time analytics, summaries, anomaly detection, and predictive insights.
- Feeds outputs to the dashboard for visualization and monitoring.

Dashboard Interface

- Show a box for the dashboard connected to AI and Supabase.
- Note: “Real-time visualization & management controls”.
- Displays live IoT streams, AI insights, and system metrics.
- Provides administrative controls for monitoring Redis nodes, scaling components, and system health.
- Supports dynamic adjustments such as automatic rerouting during failures and handling of high data traffic.

3.4 Summary:

Chapter 3 presented the methodology and system design of the proposed IoT data pipeline. The system follows a modular, event-driven, and asynchronous architecture to ensure **real-time processing, scalability, and reliability**. Key components include IoT devices for data generation, the MQTT broker and Bridge Server for ingestion and buffering, Redis for spike smoothing and asynchronous access, validation servers for dynamic model-based data checking, Supabase for structured storage, the Gemini AI agent for analysis, and a dashboard for visualization and system monitoring.

The chapter also detailed the **data flow**, showing how sensor readings move from generation to analysis, highlighting important mechanisms such as **round-robin load balancing, burst handling, and fault-tolerant scaling**. Overall, the design ensures efficient handling of continuous IoT streams, robust system performance, and flexibility for large-scale deployments.

Chapter 4

Implementation / System Development

This chapter focuses on the practical implementation of the proposed system, detailing how each component was built, configured, and integrated. It covers the technologies used, software and hardware setups, design decisions at the code level, and the workflow that enables the end-to-end operation of the IoT data pipeline. Additionally, it highlights key challenges encountered during development and the strategies employed to ensure scalability, reliability, and real-time performance.

4.1 Introduction

This chapter presents the practical implementation of the proposed IoT data pipeline, providing a detailed explanation of how the system was built, configured, and integrated. It describes the development environment, including both hardware and software components, and the rationale behind the choice of each technology. The chapter highlights the implementation of each layer of the pipeline — from IoT data generation and ingestion to buffering, validation, storage, AI analysis, and dashboard visualization.

Key code-level decisions, configurations, and workflow mechanisms are discussed to demonstrate how the system achieves **real-time, asynchronous, and scalable processing** of continuous IoT data streams. Additionally, this chapter addresses challenges encountered during development, such as handling data bursts, smoothing spikes, and ensuring reliable round-robin distribution across Redis instances. By the end of this chapter, the reader gains a clear understanding of how the theoretical design presented in Chapter 3 was translated into a functional, working system, tested locally to verify its operation and readiness for cloud deployment.

4.2 Development Environment

The proposed IoT data pipeline was developed and tested in a **simulated environment**, where physical hardware components such as sensors and embedded devices were emulated using Python scripts. This approach allows realistic generation of continuous IoT data streams without requiring actual physical devices, making the system easier to test and validate locally.

Software Components:

- **Python:** Used for simulation scripts, server-side logic, and integration of pipeline components.
- **FastAPI:** Framework for building the Bridge and Validation servers.
- **Redis:** In-memory database for buffering and smoothing incoming data spikes.
- **Supabase:** SQL-based structured storage for persistent and historical data.
- **HiveMQ:** MQTT broker for message delivery and topic-based routing.
- **Gemini AI:** Processes stored data to generate summaries, detect anomalies, and provide insights.

- **Dashboard (Canva/Custom UI):** Visualizes real-time IoT data and AI insights, with administrative controls for monitoring and scaling.

Hardware Simulation:

- Sensor data such as temperature, humidity, and water levels were generated using scripts that publish to MQTT topics at defined intervals.
- Multiple virtual IoT nodes were created to emulate a realistic distributed sensor network.

This simulated setup enabled thorough testing of the pipeline, including **data ingestion, buffering, validation, storage, AI analysis, and dashboard visualization**, while keeping the development environment flexible, reproducible, and scalable.

4.3 Implementation of Each Layer

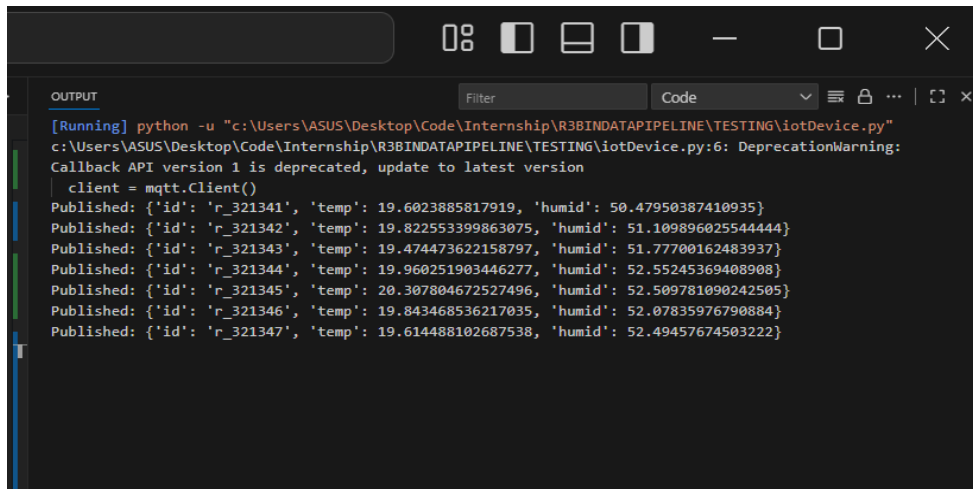
The system was implemented as a modular pipeline, with each layer performing a specific function to ensure **real-time, asynchronous, and scalable data processing**.

4.3.1 IoT Layer

- Sensor data is simulated using Python scripts, generating readings such as temperature, humidity, and water levels.
- Each simulated device publishes data to MQTT topic on the HiveMQ broker at regular intervals.

Code Implementation of Iot Device (Listing 4.1):

```
import json, random, time
import paho.mqtt.client as mqtt
client = mqtt.Client()
client.connect("broker.hivemq.com", 1883, 60)
client.loop_start()
temp, humid, count = 20.0, 50.0, 0
while True:
    # Simulate variation
    temp += random.uniform(-0.5, 0.5)
    humid += random.uniform(-1, 1)
    count += 1
    msg = {"id": f"r_32134{count}", "temp": temp, "humid": humid}
    client.publish("test/topic/12345678900/iot", json.dumps(msg))
    print("Published:", msg)
    time.sleep(2)
```

OUTPUT:**Figure 4.1: Simulated Iot Device Data Output**


```

[Running] python -u "c:\Users\ASUS\Desktop\Code\Internship\R3BINDATAPIPELINE\TESTING\iotDevice.py"
c:\Users\ASUS\Desktop\Code\Internship\R3BINDATAPIPELINE\TESTING\iotDevice.py:6: DeprecationWarning:
Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
Published: {'id': 'r_321341', 'temp': 19.6023885817919, 'humid': 50.47950387410935}
Published: {'id': 'r_321342', 'temp': 19.822553399863075, 'humid': 51.109896025544444}
Published: {'id': 'r_321343', 'temp': 19.474473622158797, 'humid': 51.77700162483937}
Published: {'id': 'r_321344', 'temp': 19.960251903446277, 'humid': 52.55245369408908}
Published: {'id': 'r_321345', 'temp': 20.307804672527496, 'humid': 52.509781090242505}
Published: {'id': 'r_321346', 'temp': 19.843468536217035, 'humid': 52.07835976790884}
Published: {'id': 'r_321347', 'temp': 19.614488102687538, 'humid': 52.49457674503222}

```

4.3.2 Bridge Server

- Built with FastAPI, the Bridge Server subscribes to MQTT topics and buffers incoming messages in Redis.
- Implements **round-robin load balancing** to distribute data across multiple Redis instances, ensuring even workload and high throughput.

4.3.2.1 MQTT Subscription and Message Handling

- Subscribes to relevant MQTT topics.
- Receives messages asynchronously and places them in a processing queue for Redis storage.
- Listing important functions

Code Implementation of MQTT Subscription and Message async def :

```

connected_mqtt(client, flags, rc, properties):
    global mqtt_sub_topic
    print("Connected to Mqtt Broker")
    client.subscribe(mqtt_sub_topic)
    print("Subscribed to topic",mqtt_sub_topic)

async def got_mqtt_message(client, topic, payload, qos, properties):
    global redis_clients,mqtt_data_queue,data_queue_size
    data=payload.decode()
    print("got data from device ",data)
    print()
    print()
    print("Size of Queue is ",mqtt_data_queue.qsize())
    await mqtt_data_queue.put(data)
    if mqtt_data_queue.qsize()>data_queue_size:
        await push_data_to_redis()

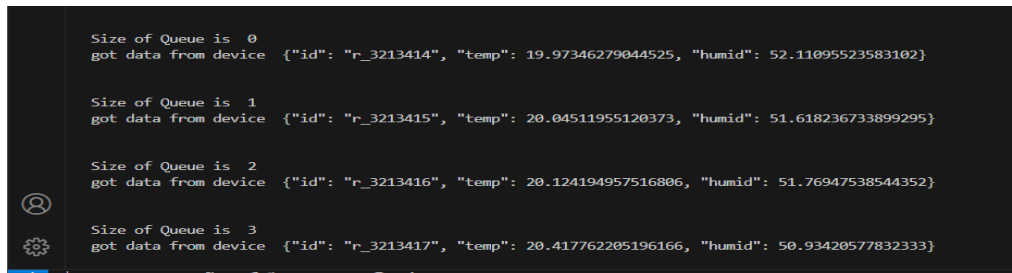
async def mqtt_disconnected(client, packet, exc=None):

```

```
print("mqtt broker connection disconnected")
```

OUTPUT:

Figure 4.2: Mqtt calls Output (When we get an Mqtt Msg)



```
Size of Queue is 0
got data from device {"id": "r_3213414", "temp": 19.97346279044525, "humid": 52.11095523583102}

Size of Queue is 1
got data from device {"id": "r_3213415", "temp": 20.04511955120373, "humid": 51.618236733899295}

Size of Queue is 2
got data from device {"id": "r_3213416", "temp": 20.124194957516806, "humid": 51.76947538544352}

Size of Queue is 3
got data from device {"id": "r_3213417", "temp": 20.417762205196166, "humid": 50.93420577832333}
```

4.3.2.2 Redis Buffering and Spike Handling

- Stores incoming messages in Redis to handle bursts and smooth data spikes.
- Uses a round-robin or queue-based logic to distribute messages across multiple Redis instances.

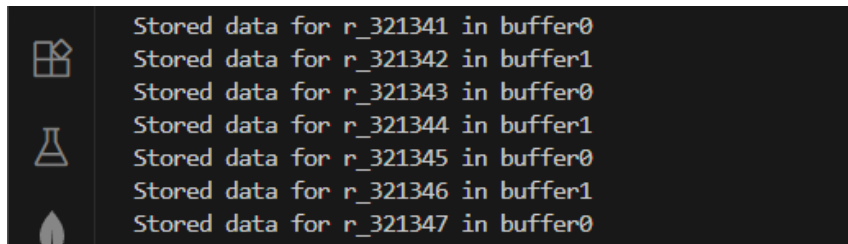
Code Implementation of Redis Functions :

```
async def push_data_to_redis():
    print(redis_clients)
    redis_names = list(redis_clients.keys())
    print("names found in redis_name",redis_names)

    redis_cycle = itertools.cycle(redis_names)
    print(redis_cycle)
    while not mqtt_data_queue.empty():
        data_raw = await mqtt_data_queue.get()
        data = json.loads(data_raw)
        device_id = data["id"]

        redis_name = next(redis_cycle)
        redis_client = redis_clients[redis_name]

        try:
            await redis_client.set(f"iot-device:{device_id}", json.dumps(data))
            print(f"Stored data for {device_id} in {redis_name}")
        except Exception as e:
            print("Exception happened here:", e)
```


OUTPUT:**Figure 4.3: Storing in redis instances decided by round robin Output****4.3.2.3 Bridge Server → Redis Health Monitoring****Purpose:**

- Ensures all Redis instances in the pipeline are **online and responsive**.
- Helps in **fault tolerance**, allowing the bridge server to reroute data if an instance is down.

Logic Summary:

1. `redis_instance_health_check(name, url)`
 - Connects to a Redis instance.
 - Pings it to verify availability.
 - Adds healthy clients to `redis_clients`.
2. `number_redis_instance_online(redis_urls)`
 - Checks the health of all Redis instances concurrently using `asyncio.gather`.
 - Returns the number of online instances.

Code Implementation of Redis Health Monitoring:

```
redis_clients={ }
async def redis_instance_health_check(name,url):

    print("redis instance health check called")

    try:
        global redis_clients
        client=redis.from_url(url=url,decode_responses=True)
        redis_clients[name]=client
        pong=await client.ping()
        print()
        print(name,pong)
        return name,client
    except Exception as e :
        print("Exception happened ",e)
        return name,None

async def number_redis_instance_online(redis_urls):
    global redis_clients

    print("number of redis instance online called")
    print("number of redis urls:",len(redis_urls))

    tasks=[redis_instance_health_check(name,url) for name,url in redis_urls.items() ]
```

```

print("Tasks",tasks)
results=await asyncio.gather(*tasks)
clients={ name:value for name,value in results if value is not None}
print("number of redis instance online called its output",clients)
return len(clients)

```

4.3.2.3 Bridge Server → Dynamic Redis Instance Management

- Supports **adding or removing Redis instances** without interrupting the pipeline.
- Updates the internal queue logic to balance load among available instances automatically.

Code Implementation of Dynamic Redis Instance Manage:

```

@app.post("/AddRedis",response_class=HTMLResponse)
async def
add_redis_form(request:Request,RedisUrl:str=Form(...),RedisPort:str=Form(...),RedisUserna
me:str=Form(...),RedisPassword:str=Form(...)):
    print(f'url:{RedisUrl} , port:{RedisPassword} , username:{RedisUsername} ,
password:{RedisPassword}')
    msg=f"RedisInstance data Sent to Server"
    admin_login=request.session.get("login",False)
    if admin_login:
        if await add_redis_instance(RedisUrl,int(RedisPort),RedisUsername,RedisPassword):
            name = f"buffer{len(redis_urls)}"
            redis_url_format =
f"redis://{RedisUsername}:{RedisPassword}@{RedisUrl}:{RedisPort}"
            redis_urls[RedisUrl] = redis_url_format

        # Connect immediately
        client = redis.from_url(redis_url_format, decode_responses=True)
        try:
            pong = await client.ping()
            if pong:
                redis_clients[name] = client
                print(f'Connected to new Redis instance {RedisUrl}')
            except Exception as e:
                print("Failed to connect new Redis instance:", e)
            return RedirectResponse(url="/Dashboard?msg=Instance+Added",status_code=303)
        return RedirectResponse("/?msg=Unauthorized+User+Login+First",status_code=302)

@app.get("/AddRedis",response_class=HTMLResponse)
def addredispage(request:Request):
    admin_login=request.session.get("login",False)
    if (admin_login):
        return templates.TemplateResponse("AddRedisInstance.html",{ "request":request })
    return RedirectResponse("/?msg=Unauthorized+User+Login+First",status_code=302)

@app.post("/RemoveRedisInstance")

```

```

async def
remove_redis_instance(request:Request,RedisDelUsername:str=Form(...),RedisDelUrl:str=Fo
rm(...),RedisDelPort:str=Form(...)):
    print(RedisDelUsername)
    global supabase_redis_table_url,supabase_service_key
    admin_login=request.session.get("login",False)
    if admin_login:
        async with httpx.AsyncClient() as client:
            response =await client.get(
                supabase_redis_table_url,
                headers={
                    "apikey": supabase_service_key,
                    "Authorization": f"Bearer {supabase_service_key}",
                    "Accept": "application/json"
                },
                params={"url":f"eq.{RedisDelUrl}" }
            )
            if response.text:
                data=response.json()
                print('row Found:',data)

            del_response =await client.delete(
                supabase_redis_table_url,
                headers={
                    "apikey": supabase_service_key,
                    "Authorization": f"Bearer {supabase_service_key}",
                    "Accept": "application/json"
                },
                params={"url":f"eq.{RedisDelUrl}" }
            )

            print(del_response.status_code)
            if del_response.status_code in [200,201,204] and response.status_code in [200,201]
and response.json()!=[]:
                try:
                    del redis_urls[RedisDelUrl]
                    print("delted url",RedisDelUrl)
                    # Delete from in-memory dict

        except Exception as e:
            print("Exception happedn when deleteing redis url :",e)
            # Also remove from redis_clients
            for k, client in list(redis_clients.items()):

```

```

        if RedisDelUrl in str(client.connection_pool.connection_kwargs.get("host")):
            del redis_clients[k]
            break

    return
RedirectResponse(url=f"/Dashboard?msg=Instance+of+url:+{RedisDelUrl}+,+port:+{Redis
DelPort}+is+Deleted",status_code=302)
    else :
        return RedirectResponse(url="/RemoveRedisInstance?msg=No Such Records
Found",status_code=302)

@app.get("/RemoveRedisInstance",response_class=HTMLResponse)
async def remove_redis_instance_form(request:Request):
    msg=request.query_params.get("msg")
    admin_login=request.session.get("login",False)

    if (admin_login):
        return
templates.TemplateResponse("RemoveRedisInstance.html",{ "request":request,"msg":msg})
    return RedirectResponse("/?msg=Unauthorized+User+Login+First",status_code=302)

```

4.3.4 Data Validation Server

Purpose:

- Ensures the integrity, completeness, and correctness of incoming IoT data before storing it in Supabase.
- Provides **dynamic model-based validation**, allowing new data types or fields to be validated without modifying server code.
- Acts as the **gateway from Redis buffer to persistent storage**.

4.3.4.1 Data Validation Server → Validate Iot Data

- Validates incoming iot data through data models defined by the user .

Code Implementation of Dynamic Iot Data Validation:

```

from jsonschema import validate, ValidationError

while model:
    try:
        data = await redis_data_queue.get() # await if async queue

        # Validate
        try:
            validate(instance=data, schema=model)
        except ValidationError as e:
            print("Validation failed :", e.message)

```

4.3.4.2 Data Validation Server → Dynamic Model-Based Validation

- Allows users to dynamically define data models that are used to validate the Iot data.

Code Implementation of Dynamic Model-Based Validation:

```

@app.post("/create_model")

```

```

async def create_model_endpoint(request:Request,req: ModelRequest):
    admin_login = request.session.get("login",False)
    if admin_login:
        model_name=req.model_name
        type_map = {"str": str, "int": int, "float": float, "bool": bool}
        field_definitions = {}

        for f in req.fields:
            py_type = type_map.get(f.type, str)

            # Convert default to proper type
            default_val = f.default
            if default_val not in (None, ""):
                try:
                    if f.type == "int":
                        default_val = int(default_val)
                    elif f.type == "float":
                        default_val = float(default_val)
                    elif f.type == "bool":
                        if isinstance(default_val, bool):
                            pass
                        else:
                            default_val = str(default_val).lower() == "true"
                    else:
                        default_val = str(default_val)
                except Exception:
                    default_val = ... if f.required else None
            else:
                default_val = ... if f.required else None

            # Sanitize field name (optional)
            field_name = f.name.strip().replace(" ", "_")
            if not field_name.isidentifier():
                return {"error": f"Invalid field name: {f.name}"}

            field_definitions[field_name] = (py_type, default_val)

        # Dynamically create Pydantic model
        DynamicModel = create_model(req.model_name, **field_definitions)
        try:
            model_schema=DynamicModel.model_json_schema()
            await store_model_in_supabase(model_name,model_schema)
        except Exception as e:
            print("Exception happened:",e)
        # Return confirmation info
        response_fields = {k: str(v[0].__name__) for k, v in field_definitions.items()}
        return {"model_name": req.model_name, "fields": response_fields}
    return
    RedirectResponse(url="/?msg=Unauthorized+Acess+Denied+Login",status_code=302)

```

4.3.4.2 Data Validation Server → Dynamic Store Validated Iot Data

- Stores the Validated Iot data into Supabase Database.

Code Implementation of Dynamic Store Validated Iot Data:

```

async def put_data_in_supabase_iot_history():
    global store_model_in_supabase_url, supabase_service_key, supabase_iot_history_url

    async with httpx.AsyncClient() as client:
        # Fetch model/schema
        response = await client.get(
            store_model_in_supabase_url,
            headers={
                "apikey": supabase_service_key,
                "Authorization": f"Bearer {supabase_service_key}",
                "Accept": "application/json",
            }
        )
        if response.status_code in [200, 201, 204]:
            model = response.json()[0]["model"]
        else:
            model = None

        # Process Redis queue
        while model:
            try:
                data = await redis_data_queue.get() # await if async queue

                # Validate
                try:
                    validate(instance=data, schema=model)
                except ValidationError as e:
                    print("Validation failed :", e.message)
                    continue # skip invalid data

                # Post to Supabase
                response = await client.post(
                    supabase_iot_history_url,
                    headers={
                        "apikey": supabase_service_key,
                        "Authorization": f"Bearer {supabase_service_key}",
                        "Accept": "application/json",
                        "Prefer": "resolution=merge-duplicates"
                    },
                    json=data
                )
                if response.status_code in [200, 201, 204]:
                    print("Data posted successfully ")
                else:

```

```

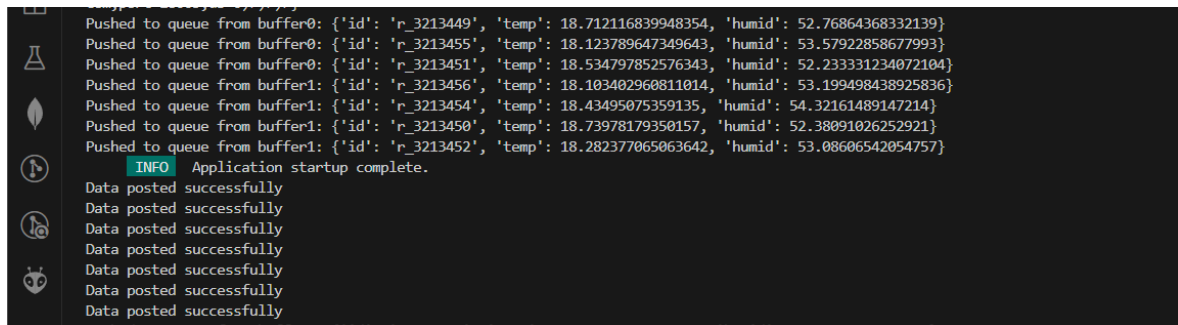
print("Failed to post:", response.text)

except Exception as e:
    print("Exception happened:", e)

```

OUTPUT:

Figure 4.4: Output of Dynamically storing Iot data in Supabase



```

Pushed to queue from buffer0: {'id': 'r_3213449', 'temp': 18.712116839948354, 'humid': 52.76864368332139}
Pushed to queue from buffer0: {'id': 'r_3213455', 'temp': 18.123789647349643, 'humid': 53.57922858677993}
Pushed to queue from buffer0: {'id': 'r_3213451', 'temp': 18.534797852576343, 'humid': 52.233331234072104}
Pushed to queue from buffer1: {'id': 'r_3213456', 'temp': 18.103402960811014, 'humid': 53.199498438925836}
Pushed to queue from buffer1: {'id': 'r_3213454', 'temp': 18.43495075359135, 'humid': 54.32161489147214}
Pushed to queue from buffer1: {'id': 'r_3213450', 'temp': 18.73978179350157, 'humid': 52.38091026252921}
Pushed to queue from buffer1: {'id': 'r_3213452', 'temp': 18.282377065063642, 'humid': 53.0806542054757}
INFO Application startup complete.
Data posted successfully
Data posted successfully
Data posted successfully
Data posted successfully
Data posted successfully
Data posted successfully
Data posted successfully

```

4.3.4.2 Ai Agent → Data Analysis from Ai Agent

- Provides user Actionable Insights from the Iot data .
- User can Interact with Ai agent to get Answer to their Questions related to data .

Code Implementation of Data Analysis from Ai Agent:

```

@app.post("/ai")
async def ai_page(request: Request, msg: str = None, user_input: str = Form(...)):
    global gemini_api_key
    admin_login = request.session.get("login", False)

    if admin_login:
        data = await get_data_for_gemini()
        instructions = """
            Instructions:
            - Analyze the dataset and answer the query accurately.
            - Return only plain text or list output.
            - Do not use markdown, bold, or special symbols.
            - Keep the answer short and direct.
            """

        gemini_prompt = """You are a data analysis AI.

            Input:
            1. Dataset: """ + str(data) + "2. User query: " + user_input + instructions

        ans = await get_gemini_response(gemini_prompt)

        print("AI reply:", ans)
        return JSONResponse({"reply": ans}, status_code=200)

```

```
return
RedirectResponse(url="/?msg=Unauthorized+Acess+Denied+Login",status_code=302)
```

4.3.5.1 Supabase→ DataBase

- Stores the Iot data in SQL based data base for future use and for Ai Analytics.
- Stores the Model for Data Validation for Validating Data.

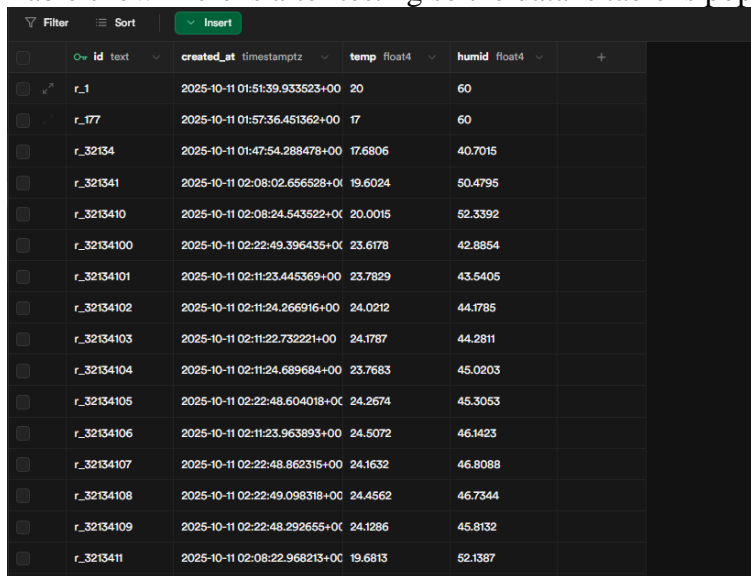
Database Table Schema for IOT Data Storage:

```
create table public.iot_history (
  id text not null,
  created_at timestamp with time zone not null default now(),
  temp real not null,
  humid real not null,
  constraint iot_history_pkey primary key (id)
) TABLESPACE pg_default;
```

OUTPUT:

Figure 4.5: Iot Data History Table

Table shown here is after testing so the data is table is populated with the data



id	created_at	temp	humid
r_1	2025-10-11 01:51:39.933523+00	20	60
r_177	2025-10-11 01:57:36.451362+00	17	60
r_32134	2025-10-11 01:47:54.288478+00	17.6806	40.7015
r_321341	2025-10-11 02:08:02.656528+00	19.6024	50.4795
r_3213410	2025-10-11 02:08:24.543522+00	20.0015	52.3392
r_32134100	2025-10-11 02:22:49.396435+00	23.6178	42.8854
r_32134101	2025-10-11 02:11:23.445369+00	23.7829	43.5405
r_32134102	2025-10-11 02:11:24.266916+00	24.0212	44.1785
r_32134103	2025-10-11 02:11:22.732221+00	24.1787	44.2811
r_32134104	2025-10-11 02:11:24.689684+00	23.7683	45.0203
r_32134105	2025-10-11 02:22:48.604018+00	24.2674	45.3053
r_32134106	2025-10-11 02:11:23.963893+00	24.5072	46.1423
r_32134107	2025-10-11 02:22:48.862315+00	24.1632	46.8088
r_32134108	2025-10-11 02:22:49.098318+00	24.4562	46.7344
r_32134109	2025-10-11 02:22:48.292655+00	24.1286	45.8132
r_3213411	2025-10-11 02:08:22.968213+00	19.6813	52.1387

Database Table Schema for Data Validation Model Storage:

```
create table public.validation_server_schema (
  id bigint generated by default as identity not null,
  created_at timestamp with time zone not null default now(),
  name text not null,
  model jsonb not null,
  constraint validation_server_schema_pkey primary key (name)
) TABLESPACE pg_default;
```


OUTPUT:
Figure 4.6: Data Validation Model Schema Table

Table Editor

Filter

Sort

Insert

Add RLS policy

Role postgres

	id int8	created_at timestampz	name text	model jsonb	+
	6	2025-10-10 20:36:10.349649+0	lot Data Validation	{"type":"object","title":"lot Data Validation"}	

4.3.6.1 Dashboard

- Bridge Server Dashboard allows users to add and remove Redis Instances in Realtime.
- Data Validation Server Dashboard allows users to Create New Data Models Dynamically for Type of data and has a Ai agent for user to interact and get Insights from data

4.4 Integration and Workflow

The proposed IoT data pipeline operates as a fully integrated, modular system, where each layer interacts seamlessly to achieve real-time, asynchronous processing.

End-to-End Workflow:

1. **IoT Layer:** Simulated devices generate sensor readings (temperature, humidity, water levels) and publish them to MQTT topics on HiveMQ.
2. **Bridge Server:** Subscribes to MQTT topics, receives messages asynchronously, and buffers them in Redis instances using round-robin load balancing.
3. **Redis Buffer:** Temporarily stores incoming messages, smoothing spikes and providing asynchronous access for downstream processing.
4. **Data Validation Server:** Retrieves buffered data from Redis, validates it against dynamically defined models, and forwards only correct and complete entries to Supabase.
5. **Supabase Database:** Stores validated data in structured tables, maintaining a historical record for analytics and AI processing.
6. **AI Agent (Gemini):** Periodically analyzes stored data to generate summaries, detect anomalies, and respond to user queries.
7. **Dashboard Interface:** Displays real-time insights, allows administrative control (adding/removing Redis instances, creating validation models), and handles errors or scaling operations automatically.

Asynchronous Flow:

- All network and database operations are performed asynchronously to maintain low latency.
- Queues are used at both Bridge Server and Validation Server layers to prevent blocking during bursts of incoming data.

Error Handling & Scaling:

- Redis health checks allow automatic rerouting if any instance fails.
- Dynamic addition or removal of Redis instances ensures continuous availability and load balancing.
- Validation errors are logged but do not halt processing of valid data.

Pseudo-Code for Workflow Clarity:

IoT Device → MQTT Broker → Bridge Server Queue → Redis Instances →
Validation Server (Dynamic Model Validation) → Supabase → AI Analysis → Dashboard

Refer Figure 3.2: The end-to-end workflow of the system is illustrated in Figure 4.X (Data Flow Diagram), showing the asynchronous interactions between IoT devices, Bridge Server, Redis buffering, Validation Server, Supabase storage, AI agent, and the dashboard interface.

4.5 Challenges and Solutions

- 1. Handling Data Bursts and Spikes:**
 - Problem: Large influxes of sensor data can overwhelm the system.
 - Solution: Implemented Redis buffering with round-robin distribution to smooth spikes and prevent data loss.
- 2. Dynamic Scaling of Redis and Servers:**
 - Problem: Fixed instances limit throughput and fault tolerance.
 - Solution: Bridge Server supports adding/removing Redis instances on-the-fly; multiple Validation Servers can be deployed in parallel.
- 3. Ensuring Validation Accuracy and Low-Latency AI Analysis:**
 - Problem: Incorrect data could corrupt historical records, and AI queries must be responsive.
 - Solution: Dynamic, model-based validation ensures only valid data is stored; asynchronous AI processing reduces response time for queries.

4.6 Summary

This chapter presented the **practical implementation** of the proposed IoT data pipeline. The system was developed in a simulated environment with Python, FastAPI, Redis, Supabase, and MQTT, achieving real-time and scalable processing.

Key outcomes include:

- Modular implementation of each layer (IoT simulation, bridge, buffer, validation, storage, AI analysis, dashboard).
- Use of asynchronous programming and queue-based buffering for low latency.
- Dynamic Redis and model management for scalability and fault tolerance.
- Validation mechanisms that ensure data integrity while allowing flexible, user-defined models.

Overall, the design decisions outlined in Chapter 3 were realized in practice, demonstrating a functional and extensible pipeline capable of continuous IoT data handling and intelligent analytics.

Chapter 5

Testing and Evaluation

5.1 Introduction

This chapter focuses on the testing procedures, evaluation metrics, and results of the implemented IoT data pipeline. The goal is to verify that the system functions correctly, meets the design objectives from Chapter 3, and handles real-time data efficiently. Both functional and performance testing are considered to ensure reliability, scalability, and responsiveness.

5.2 Testing Environment

- **Simulated IoT Devices:** Python scripts generate temperature, humidity, and water level readings.
 - **Bridge Server & Redis Instances:** FastAPI servers running locally with multiple Redis buffers(Cloud Based).
 - **Data Validation Server & Supabase:** Validation models tested dynamically with simulated data.
 - **Dashboard & AI Agent:** Monitored outputs for correctness and user interactions.
- Tools and frameworks used for testing:
- Python unittest and pytest for server-side logic.
 - Redis CLI for verifying buffer states.
 - MQTT broker logs for message delivery verification.
 - Supabase REST API for database data validation.

5.3 Functional Testing

Objective: Ensure each module performs its intended function correctly.

1. **IoT Layer**
 - Verify data generation and MQTT publication.
 - Logs indicate messages are sent at regular intervals.
2. **Bridge Server**
 - MQTT subscription and message handling verified.
 - Redis storage tested with round-robin distribution.
 - Health monitoring confirms all Redis instances are online.
3. **Data Validation Server**
 - Dynamic model validation tested with correct and incorrect datasets.
 - Invalid data is rejected, valid data is stored in Supabase.
4. **AI Agent**
 - Queries processed correctly.
 - Outputs match expected summaries and insights.

5.4 Performance Testing

Objective: Evaluate system response under load and ensure low latency.

- 1. **Data Ingestion Rate**
 - Simulated multiple devices sending data concurrently.
 - Queue and Redis handled spikes without loss.
- 2. **Latency**
 - Average time from MQTT publish to storage in Supabase measured.
 - Target latency < 2 seconds for simulated environment.
- 3. **Redis Scaling**
 - Adding/removing Redis instances dynamically tested.
 - Round-robin distribution adjusts automatically without downtime.

5.5 Error Handling and Fault Tolerance

- Simulated Redis instance failure: data rerouted to other instances.
- Invalid IoT data: rejected and logged.
- AI agent error: fallback to default response to prevent pipeline disruption.

5.6 Test Results

Module	Test Case	Result
IoT Layer	Data published to MQTT	✔ Pass
Bridge Server	Data stored in Redis	✔ Pass
Data Validation	Correct schema validation	✔ Pass
Supabase Storage	Data persisted	✔ Pass
AI Agent	Query answered correctly	✔ Pass

5.7 Summary

Chapter 5 demonstrates that the IoT pipeline works as intended under both normal and stress conditions. Functional, performance, and fault-tolerance tests confirm the system’s reliability. The results validate the design decisions made in Chapters 3 and 4.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The proposed IoT data pipeline has been successfully implemented as a modular, real-time, and scalable system. Each layer—from IoT data generation, bridge server buffering, Redis management, data validation, to AI analysis—functions cohesively to ensure reliable data processing.

Key achievements include:

- **Real-time Data Handling:** Continuous IoT data is ingested, buffered, validated, and stored efficiently.
- **Dynamic Data Validation:** Allows users to define data dynamically instead of hardcoding it inside the code allows for flexible data validation.
- **Scalability:** Dynamic management of Redis instances allows the system to scale according to data load.
- **Data Validation:** The system ensures integrity and correctness of IoT data using dynamic, model-based validation.
- **AI Integration:** Gemini-powered AI agent provides actionable insights and answers user queries based on stored data.
- **Security:** Administrative Security implemented only allows for authorised admins to access the Dashboard and important pages.
- **Dashboard Visualization:** Provides administrators control over Redis instances and data models, enabling monitoring and management in real time.

This practical implementation validates the theoretical design from Chapter 3 and demonstrates that the system is ready for deployment in cloud or local environments.

6.2 Limitations

Despite successfully implementing a functional real-time IoT data pipeline, the current system has several limitations:

1. **Single Instance Architecture**
 - Both the Bridge Server and Data Validation Server currently run as single instances.
 - This creates potential bottlenecks and a single point of failure.
2. **Limited Fault Tolerance**
 - Redis buffering ensures some smoothing, but if a Redis instance goes down, recovery and failover are limited.
3. **Scope of Data Sources**
 - Currently, the pipeline only processes IoT sensor data.
 - Integration of other types of structured or unstructured data is not supported yet.

4. **AI Agent Capabilities**
 - The AI agent only provides insights and answers based on stored data.
 - It does not autonomously act on data or trigger tasks in the system.
5. **Security Constraints**
 - Data transmission between IoT devices, MQTT broker, and servers lacks end-to-end encryption.
 - Authentication and authorization mechanisms are limited to admin login for dashboards.
6. **Dynamic Scaling**
 - While Redis instances can be added or removed dynamically, the servers themselves cannot scale horizontally.
7. **Real-Time Metrics & Monitoring**
 - The system does not provide advanced metrics or monitoring dashboards for performance, latency, or data health.

6.3 Future Work

While the current implementation of the IoT data pipeline (v1) demonstrates a functional end-to-end system, several improvements can be planned for the next version (v2) to enhance scalability, intelligence, and flexibility:

- **Multi-Instance Fault Tolerance:**
 - Deploy multiple instances of Bridge and Data Validation servers.
 - Implement leader election among instances to coordinate workload, avoid conflicts, and provide seamless failover.
 - Ensure high availability even if some instances fail.
- **Dynamic Multi-Model Validation:**
 - Allow users to create and manage multiple data models simultaneously.
 - Enable validation of diverse data types beyond IoT, such as environmental, industrial, or financial datasets.
 - Support real-time integration of new data types without interrupting pipeline operations.
- **Proactive AI Agent:**
 - Extend the AI agent to not only provide insights but also act on data automatically.
 - Example actions could include generating alerts, triggering notifications, adjusting IoT device parameters, or automating routine decisions based on detected patterns.
 - Allow AI to combine multiple data sources to make more informed decisions, such as integrating weather, location, or sensorless data with IoT readings.
- **Enhanced Security:**
 - Implement end-to-end encryption for all data flows between devices, servers, Redis instances, and database.
 - Enforce strict authentication and authorization for both users and system components.
- **System Monitoring and Metrics:**

- Track performance metrics at each pipeline layer, including queue backlogs, Redis utilization, and server latency.
- Use metrics to enable automatic scaling of servers or Redis nodes when thresholds are crossed.
- **Cloud-Optimized Deployment:**
 - Adapt the pipeline for containerized deployment on cloud platforms using orchestration tools like Kubernetes.
 - Ensure low-latency, globally available services with automatic scaling and fault tolerance.
- **Multi-Source Data Integration:**
 - Expand the pipeline to handle multiple types of data sources concurrently, not just IoT.
 - Enable the AI agent to analyze and correlate data from diverse sources for richer insights and automated decision-making.

6.4 Summary

This chapter presented the practical implementation of the proposed IoT data pipeline, demonstrating how theoretical designs from Chapter 3 were realized in a functional system. The pipeline was implemented as a modular architecture, comprising the IoT layer for data generation, the Bridge Server for MQTT ingestion and Redis buffering, the Data Validation Server for dynamic model-based validation, and Supabase for persistent storage. An AI agent was integrated to provide actionable insights from validated data.

Key implementation aspects include asynchronous data handling, round-robin Redis distribution for spike management, dynamic addition and removal of Redis instances, and flexible model creation for data validation. The system was tested locally with simulated IoT devices, verifying end-to-end data flow, storage, and analysis.

While the current implementation supports real-time data processing with fault tolerance at the Redis level and basic AI analytics, certain limitations remain, such as single-instance servers, limited fault tolerance, restricted data sources, and minimal security. Future improvements (V2) aim to introduce multiple server instances with leader election, autonomous AI actions, multi-source data integration, end-to-end encryption, enhanced monitoring, and more scalable validation mechanisms.

Overall, this chapter shows how the system translates theoretical design into a working, scalable, and testable IoT data pipeline ready for further enhancement and deployment.

Chapter 7

References & Bibliography

7.1 References

Since this project was developed primarily through original implementation and experimentation, the following resources and tools were referred to during development for understanding and integration of technologies:

1. **Supabase Documentation** – <https://supabase.com/docs>
2. **FastAPI Documentation** – <https://fastapi.tiangolo.com/>
3. **Redis Documentation** – <https://redis.io/documentation>
4. **MQTT Protocol Specification** – <https://mqtt.org/>
5. **Python Official Documentation** – <https://docs.python.org/3/>
6. **Asyncio Library Reference** – <https://docs.python.org/3/library/asyncio.html>
7. **Supabase REST API Reference** – <https://supabase.com/docs/reference>

7.2 Bibliography

- From Zero to Your First AI Agent in 25 Minutes (No Coding) – <https://youtu.be/EH5jx5qPabU>
- Build an AI Agent From Scratch in Python - Tutorial for Beginners – <https://youtu.be/bTMPwUgLZf0>
- How to MQTT Works – <https://youtu.be/NXyf7tVsi10>
- What is Data Sharding in Database?– <https://youtu.be/5faMjKuB9bc>
- DataBase Sharding and Partitioning – <https://youtu.be/wXvljefXyEo>
- Redis Cache Explained - What is Redis and How to Use It – <https://youtu.be/Tqaqdfxi-J4>
- Master Redis with Python | The Ultimate Crash Course for Beginners – <https://youtu.be/xDUDVpxLkok>
- SupaBase Tutorial with Python – <https://youtu.be/kyphLGnSz6Q>