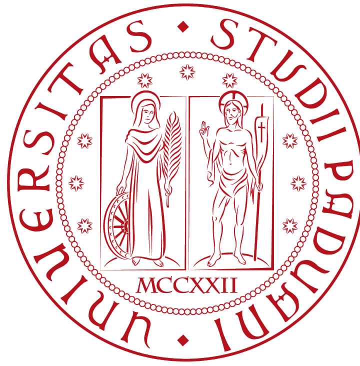

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Degree in ICT for Internet and Multimedia

Design and Simulation of Smart Device Control Systems for IoT-Based Automation

Supervisor:
Prof.ssa Federica Battisti

Co-Supervisor:
Dr-Ing Mazyar Seraj

Candidate:
Zahra Shahrooei
2072543

Academic Year 2024-2025

Abstract

The *Internet of Things (IoT)* has appeared as an innovative idea, enabling networked devices to automate tasks, control energy, and ensure better accessibility. However, building large and effective IoT systems remains a challenge, especially when creating systems that combine real-time sensor data, control mechanisms, and easy-to-use interfaces. This project tackles these issues by introducing a Python-based IoT simulation framework for automating smart devices. The system developed in this work includes a simulation module for generating realistic sensor readings, a decision-making control logic module, and a Flask-based web dashboard for real-time visualization and interaction. The virtual sensors track important environmental factors, including timestamp, temperature, and motion, and the control logic manages actuators such as fans and lights based on user-defined thresholds. Data visualization is achieved using a lightweight and responsive dashboard to ensure a seamless user experience. This thesis makes a considerable contribution through the development of an extensible *Internet of Things* (IoT) simulation framework, adding real-time control logic, and demonstrating the usefulness of interactive visualization for the automation of smart systems. The findings open avenues for further research in areas such as machine learning-based optimization of IoT and hardware integration. By its contribution to the creation of a solid base for the simulation and design of IoT systems, this research takes part in building the field of IoT automation subjects, offering findings for both academic research and practical applications.

Acknowledgements

I would like to express my deepest gratitude to my research supervisors, **Prof. Federica Battisti** and **Dr. Mazyar Seraj**, whose expertise and knowledge were invaluable during this master's thesis.

I am deeply grateful to the **University of Padua (Unipd)**, where I had the privilege to pursue my studies, and to the **Technical University of Eindhoven (TU/e)** for providing me the opportunity to conduct my research in an inspiring academic environment.

Dad, thank you for teaching me resilience, for showing me what it means to work hard and never give up.

Mom, thank you for your endless love, for your sacrifices, and for always believing in me.

Sahar, **Mina**, my dear sisters, my biggest supporters, your love and belief in me have carried me through every struggle.

My Partner, thank you for your patience, your kindness, and for always standing by my side.

To my **dear friends**, who have been my second family, thank you for your laughter and support through all the challenges.

And finally, to **Iran**, the place where I was born, raised, and learned to dream. I carry you with me always, and I hope to see your freedom one day.

Contents

Abstract **III**

Acknowledgements **V**

1 Introduction **1**

1.1 Background and Motivation 2

1.2 Problem Statement 4

1.3 Research Objectives 5

1.4 Thesis Structure 7

2 Literature Review **9**

2.1 IoT-Based Automation 9

2.1.1 Evolution of IoT in Smart Systems 9

2.1.2 IoT Device Communication Architecture 10

2.2 Sensor Data Logging in IoT 12

2.2.1 SQLite Database for Efficient IoT Data Storage 13

2.2.2 Real-Time Data Processing & Query Optimization 14

2.3 Visualization in IoT Dashboards 16

2.3.1 Web-Based Monitoring Systems 17

2.3.2 Flask vs. Other Web Technologies 18

2.4 Control Logic in IoT 19

2.4.1 Actuator Control Using Threshold-Based Logic 19

2.4.2 Integration with Machine Learning for Smart Decisions 20

2.5 IoT Security and Performance 21

2.5.1 Data Encryption Methods in IoT 22

2.5.2 Protecting Data Transmission and Storage 23

2.5.3 API Rate Limiting and Security Hardening 24

2.5.4 Authentication & Authorization for API Security 24
2.6 Gaps & Opportunities 25
2.6.1 Current Limitations in IoT Control Systems 25

3 System Design and Implementation 27
3.1 System Architecture 27
3.1.1 IoT Device Simulation Workflow 28
3.1.2 Interaction Between Sensors, Data Logger, and Dashboard 29
3.1.3 Secure User Authentication & OTP Verification 31
3.2 Technologies Used 32
3.2.1 Python for Data Processing 33
3.2.2 Flask for Web-Based IoT Dashboard 35
3.2.3 Chart.js & Bootstrap for Front-end Visualization 35
3.2.4 SQLite Database for Sensor Data Logging 36
3.3 Sensor Simulation Design 36
3.3.1 Simulating Temperature, Humidity, and Motion Sensors 37
3.3.2 Real-Time Weather API Integration for Environmental Data 37
3.3.2.1 Fetching Live Weather Data 37
3.3.2.2 Blending Simulated and Real-World Data 38
3.3.2.3 Storing and Visualizing the Data 38
3.3.3 Advantages of the Sensor Simulation Approach 38
3.4 Data Storage & Management 41
3.4.1 SQLite Database for IoT Data 41
3.4.1.1 Database Schema Design 41
3.4.1.2 Data Insertion Strategy 41
3.4.1.3 Data Retention & Cleanup Policy 42
3.4.2 Query Optimization for Real-Time Performance 42
3.4.2.1 Indexing for Faster Data Lookup 43
3.4.2.2 Query Scope Limitation to Reduce Overhead 43
3.4.2.3 Optimized Batch Processing for Bulk Inserts 43
3.4.2.4 Implementing Asynchronous Data Fetching for Real-Time Updates 44
3.5 Control Logic & Actuator Automation 44
3.5.1 Defining Rules for Fan & Light Control 44

3.5.1.1 Fan Activation Criteria 45
3.5.1.2 Light Activation Criteria 45
3.5.2 Implementing Threshold-Based Decision Logic 45
3.5.2.1 Core Implementation Strategy 45
3.5.2.2 Database Logging for Actuator Decisions 46
3.5.2.3 Potential Machine Learning Enhancements 47
3.6 Alert & Notification System 47
3.6.1 Defining Alert Triggers (High Temperature, Motion Detection, Humidity Levels) 48
3.6.1.1 Key Alert Conditions 48
3.6.2 Implementing UI-Based Alerts for Real-Time Warnings 49
3.6.2.1 Visual Alert System 49
3.6.2.2 JavaScript-Based Alert Mechanism 49
3.6.2.3 Real-Time Alert Activation Based on Sensor Readings 50
3.6.2.4 Sound Alerts 50
3.6.2.5 Backend Alert Logging in Database 51
3.6.3 Benefits of the Alert & Notification System 51

4 Methodology **53**
4.1 Full-Stack Development Approach 53
4.1.1 Key Components of the Full-Stack System 53
4.1.2 Integration of Front-end, Backend, and Database 54
4.1.2.1 Front-end and Backend Integration 54
4.1.2.2 Backend and Database Integration 54
4.1.2.3 Database and Sensor Integration 55
4.1.3 System Workflow 55
4.1.4 Benefits of Full-Stack Integration 55
4.2 Frontend Implementation (UI & Visualization) 56
4.2.1 User Authentication UI (Signup, Login, OTP Verification) 56
4.2.1.1 Signup & Login Pages 56
4.2.1.2 OTP-Based Verification for Enhanced Security 56
4.2.2 Real-Time Sensor Data Display 57
4.2.3 Interactive Data Visualization (Chart.js) 58
4.2.4 Dark Mode & UI Enhancements 59

4.3 Backend Implementation (Flask API & Business Logic) 62
4.3.1 User Authentication & OTP System 62
4.3.1.1 Secure User Access Control 62
4.3.1.2 User Registration & Login API 63
4.3.1.3 OTP-Based Verification System 63
4.3.1.4 Security Enhancements in Authentication 63
4.3.2 REST API for Real-Time Sensor Data Retrieval 63
4.3.2.1 API Endpoints for Sensor Data 64
4.3.2.2 How Data is Retrieved from SQLite 64
4.3.2.3 Performance Considerations 64
4.3.3 Actuator Control API for IoT Devices 64
4.3.3.1 Automated Actuator Control Logic 65
4.3.3.2 Manual Actuator Control API 65
4.3.3.3 Secure API Access for Actuator Control 65
4.3.4 Query Optimization for Data Fetching 65
4.3.4.1 Techniques Used for Query Optimization 65
4.3.4.2 Impact of Query Optimizations 66
4.4 Database Implementation (SQLite for Persistent Storage) 66
4.4.1 Users Table (Login, OTP Verification) 66
4.4.1.1 Schema Design 67
4.4.2 Sensor Data Table (Logging IoT Readings) 68
4.4.2.1 Schema Design 68
4.4.3 Optimized Querying for Real-Time Performance 69
4.4.3.1 Indexing for Faster Searches 69
4.4.3.2 Query Optimization for Historical Data 69
4.4.3.3 Connection Pooling 69
4.4.3.4 Efficient Data Fetching 69
4.4.3.5 Batch Processing for Data Logging 69
4.5 API Communication & Data Flow 70
4.5.1 Data Retrieval from IoT Sensors 70
4.5.2 REST API Integration with Frontend 71
4.5.3 Real-Time Updates & Asynchronous Processing 71
4.6 Security & Authentication Measures 72
4.6.1 OTP-Based User Verification 72

4.6.2 Password Hashing & Secure Login 73
4.6.3 Secure API Communication (CSRF Protection) 73
4.6.4 Data Integrity & Database Security 73

5 Results and Analysis 75
5.1 System Performance Analysis 75
5.1.1 Response Time & Data Refresh Rate 75
5.1.2 UI Interaction with Actuator Controls 76
5.2 Sensor Data Analysis 77
5.2.1 Accuracy of Simulated Sensor Readings 77
5.2.2 Real-Time Sensor Data Processing in SQLite 78
5.2.3 Historical Data Trends & Visualization 78
5.3 Control Logic Performance 79
5.3.1 Accuracy of Actuator Automation Decisions 80
5.3.2 Error Handling in Edge Cases 80
5.4 Security & Data Protection Evaluation 81
5.4.1 Effectiveness of OTP Verification System 81
5.4.2 Password Hashing & Unauthorized Access Prevention 83
5.4.3 API Security & Protection Against Attacks 84

6 Conclusion and Future Work 87
6.1 Summary of Research Contributions 87
6.2 Practical Applications of the System 87
6.2.1 Smart Home Automation 88
6.2.2 Industrial IoT Monitoring 88
6.2.3 Remote Sensor Data Management 88
6.3 Future Enhancements 88
6.3.1 Integration with Physical IoT Devices 88
6.3.2 AI & Machine Learning for Predictive IoT 88
6.3.3 User Availability & Smart Energy Suggestions 89
6.3.4 Mobile Notifications & Remote Access 89
6.3.5 Edge AI for On-Device Decision Making 89
6.3.6 Scalability & Multi-User Support 89

Acronyms 91

Bibliography **91**

List of Figures

1.1 IoT Temperature Sensor System: Data flow from devices to cloud-based analysis [18]. 4

2.1 IoT-Based Smart Home Automation System. The diagram illustrates how IoT devices communicate through a LAN manager, cloud infrastructure, and smart home servers to enable automation and remote control. 12

2.2 Example of a Web-Based IoT Dashboard for Real-Time Data Visualization [25]. 19

3.1 IoT Device Simulation Workflow: This figure illustrates the step-by-step process of IoT sensor simulation, from data acquisition to automated actuator control. 29

3.2 Interaction between Sensors, Data Logger, and Dashboard 31

3.3 Workflow of Python in IoT Data Processing: From Sensor Collection to Visualization 34

3.4 Sensor simulation workflow. It represents the generation, processing, and storage of IoT sensor data. 39

3.5 Real-time sensor data logging output. The system records temperature, humidity, and motion detection at regular intervals, ensuring continuous data collection. 40

4.1 Full-Stack Architecture Diagram. The diagram represents the interaction between the frontend, backend, and database layers, showcasing real-time data flow, actuator controls, and secure authentication mechanisms. 55

4.2 Homepage of the Smart IoT Dashboard, providing access to user authentication. 59

4.3 Signup interface where users create an account and receive an OTP for verification. 60

4.4 OTP email verification sent to the user for secure authentication. 60

4.5 OTP verification page where users enter their received OTP to complete authentication. 61
4.6 Login page requiring user credentials for secure access. 61
4.7 Smart IoT Dashboard displaying real-time sensor data, interactive charts, and actuator controls. 62

5.1 Network Performance: API requests every 3 seconds ensure fast response and updates. 76
5.2 Fan actuator state change logged by backend. 77
5.3 Light actuator state change logged by backend. 77
5.4 Comparison of simulated sensor readings with actual weather conditions in Eindhoven at the same timestamp. 78
5.5 SQLite Database: Displays real-time sensor data with timestamps, temperature, humidity, and actuator states. 78
5.6 Historical Data Trends: Chart.js visualization shows temperature and humidity changes over the last 7 days. 79
5.7 Error Handling: System logs 404 errors for fan alert endpoint while maintaining stability. 81
5.8 Email containing the OTP code sent to the user. 82
5.9 OTP verification page showing both successful and failed attempts. 82
5.10 User database storing hashed passwords and OTP verification records. 83
5.11 Database storing hashed passwords with bcrypt encryption. 84
5.12 Password hashing and verification process using bcrypt. 84
5.13 API Security Architecture Diagram 85

List of Tables

2.1 Comparison of Flask with Other Web Technologies 18

3.1 SQLite Database Schema for Sensor Data Logging 36

3.2 Sensor Log Table Schema 41

5.1 Summary of Actuator Decision Accuracy under Simulated Scenarios 80

Chapter 1

Introduction

The fast development of the **Internet of Things (IoT)** has improved the digital ecosystem by interconnecting countless devices, enabling real-time monitoring, automation, and data-driven decision-making through various sectors[3].IoT-based automation is crucial in optimizing efficiency across various domains, including smart homes, industrial automation, healthcare, and energy management [13].IoT-based automation has become the backbone of intelligent automation by integrating sensors, actuators, and cloud computing, facilitating real-time monitoring and decision-making.

IoT has great potential for automation, there are several challenges that stop it from being fully adopted. One main issue is dealing with the huge amounts of real-time data that IoT devices generate. Efficient data collection, storage, and processing mechanisms are essential for ensuring performance [14]. Furthermore, security threats such as unauthorized access and cyberattacks present major challenges to IoT systems, making strong encryption and secure data transmission essential[23].

Another challenge is the adaptability of traditional rule-based control systems, which often lack the flexibility needed for dynamic environments. The integration of machine learning into IoT control logic can enhance decision-making by enabling predictive analytics, anomaly detection, and adaptive automation.

This research focuses on designing and simulating a smart IoT-based control system that integrates real-time data logging, decision-making algorithms, and security mechanisms. The objective of this study is to implement a robust IoT data logging and simulation system that ensures real-time monitoring while maintaining data integrity. Furthermore, it seeks to develop an interactive dashboard that provides a user-friendly interface for visualizing sensor data and controlling actuators efficiently. Another key aspect of this research is improving

actuator control logic by incorporating threshold-based decision mechanisms and machine learning techniques, optimizing automation efficiency and adaptability to changing conditions.

To achieve these objectives, this thesis is structured into six chapters. The first chapter introduces the concept of IoT-based automation, highlights research challenges, and defines the study objectives. Chapter 2 presents a literature review, exploring IoT communication frameworks, data logging methods, visualization tools, security mechanisms, and existing gaps in research. Chapter 3 outlines the proposed system design and implementation, detailing the technologies used, sensor simulation, and control logic development. Chapter 4 discusses the practical implementation, covering real-time data processing, dashboard visualization, and security enhancements. Chapter 5 analyzes the results, evaluating system performance, responsiveness, and accuracy of control decisions. Finally, Chapter 6 concludes the thesis by summarizing key contributions, discussing real-world applications, and suggesting future research directions.

Through the design and simulation of an IoT-based smart control system, this study aims to contribute to the advancement of intelligent automation by improving system adaptability, security, and real-time decision-making.

1.1 Background and Motivation

The integration of IoT into automation has transformed many industries. It allows devices to communicate easily, monitor things in real-time, and make smart decisions.IoT technology is important for making processes better in areas like industrial automation, healthcare, smart cities, and energy management. It helps reduce the need for human involvement and makes systems work more efficiently[27]. Even with its possibilities, IoT-based automation still deals with major data handling, security, and flexibility issues.

One of the primary challenges in IoT automation is managing the large quantity of real-time sensor data. Ensuring that data is collected, processed, and stored efficiently while maintaining system responsiveness is crucial for real-world applications[23].

Moreover, IoT devices are naturally at risk of cybersecurity issues, such as unauthorized access, data breaches, and network intrusions. To effectively address these concerns, it is essential to implement robust data transmission protocols, employ advanced encryption techniques, and establish stringent access control mechanisms. These measures will ensure the integrity and reliability of the system, thereby enhancing overall security[23]. Traditional control systems

in IoT automation have some issues. Besides data security and processing problems, they are not very flexible in changing situations because they depend on fixed rules. This makes it hard for them to adapt when conditions change. By adding machine learning to IoT controls, we can enable things like predicting issues, spotting unusual behavior, and making decisions in real-time. This makes the systems work better and adapt more easily to new circumstances[24].

This research was conducted as part of my Erasmus internship at TU/e within the Software Engineering and Technology (SET) Cluster, where I was involved in the design and simulation of an IoT-based smart device control system. The project was in line with TU/e's research goals in embedded systems, real-time data management, and smart automation, particularly aimed at enhancing IoT control strategies via secure, flexible, and scalable automation architectures. My role focused on designing and deploying real-time data logging systems, establishing secure communication protocols for sensor-actuator networks, and creating smart decision-making processes for automated control systems. The project offered important insights into the real-world difficulties of deploying IoT automation on a large scale, emphasizing the need for creating more robust and flexible control strategies that can address the increasing complexities of contemporary automation settings.

Motivated by these challenges, this research focuses on the design and simulation of an IoT-based smart device control system. The objective is to develop a secure, scalable, and adaptive automation framework that enhances system responsiveness and decision-making capabilities. This work aligns with ongoing advancements in real-time data processing, sensor-actuator interaction, and intelligent automation, addressing critical issues related to latency, security, and system adaptability. By implementing a structured approach to data management, control optimization, and security enhancement, this study aims to contribute to the next generation of IoT-based automation solutions.

The growing dependence on smart, self-adjusting IoT control systems highlights the importance of developing solutions that offer real-time monitoring, anticipatory control, and robust communication protocols for security. Traditional IoT automation approaches rely heavily on static rules and predefined thresholds, which often fail to provide the necessary adaptability in dynamic environments. This study aims to close this gap by taking advantage of recent advances in secure IoT architectures and data-driven real-time decision-making.

The findings of this study will provide valuable insights into enhancing the reliability, scalability, and security of IoT-based automation frameworks, with potential applications in industrial automation, smart home systems, healthcare monitoring, and intelligent infrastructure management.
By addressing these challenges, this research contributes to the ongoing development of more intelligent and resilient IoT automation solutions, enabling future systems to be more efficient, adaptive, and secure in the face of evolving technological demands.

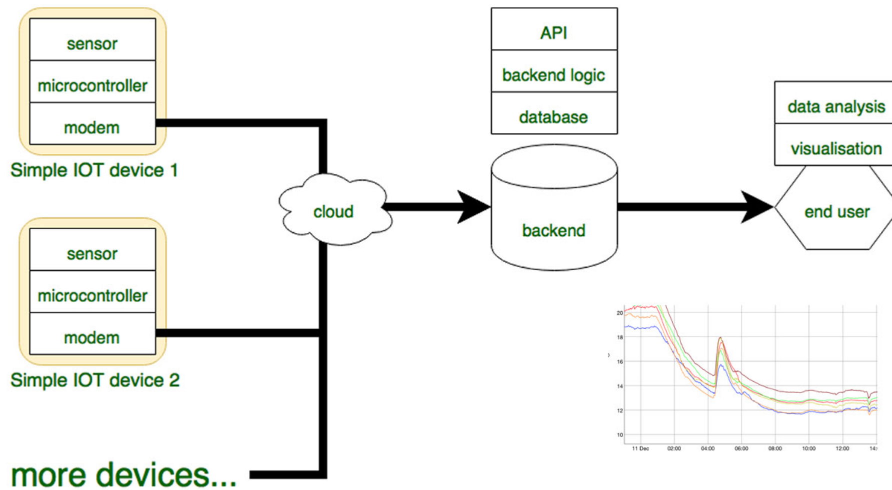

Figure 1.1: IoT Temperature Sensor System: Data flow from devices to cloud-based analysis [18].

1.2 Problem Statement

While the growth of IoT-based automation is quick, significant challenges are stopping its full-scale adoption. IoT systems must handle large amounts of real-time sensor data efficiently while maintaining system responsiveness.The continuous flow of high-frequency data leads to issues with processing, causing delays and risking data loss, especially in big IoT networks[14]. Many current data collection and processing systems are insufficient because they have difficulty in efficiently aggregating, filtering, and analyzing various data streams in real-time. Automation systems risk delayed responses, inaccurate decision-making, and reduced operational reliability without efficient data handling mechanisms[27].
Security concerns make IoT automation more complex, as connected devices can be vulnerable to unauthorized access, cyberattacks, and data breaches. Numerous IoT systems often fail to achieve in terms of strong encryption methods, effective authentication processes, and intrusion detection features, which puts vital infrastructures at risk of potential vulnerabilities. Numerous IoT systems often fail to achieve in terms of strong encryption methods, effective

authentication processes, and intrusion detection features, which puts vital infrastructures at risk of potential vulnerabilities. Relying on centralized cloud-based IoT platforms brings about extra risks, such as the potential for data interception, unauthorized changes, and compromised device security [11]. Implementing secure, real-time data logging systems is crucial to prevent security risks while ensuring system dependability and data integrity. Moreover, attaining reliability in the face of errors continues to be a significant challenge in the domain of IoT automation. Since IoT networks typically operate in distributed and resource-limited environments, unexpected issues like network interruptions, power failures, or sensor malfunctions can compromise system functionality and result in data errors or periods of downtime [2].

A significant disadvantage of traditional IoT automation is the inflexibility of rule-based control systems. Traditional IoT control methods rely on fixed thresholds and preset rules, which can delay their effectiveness in managing dynamic environments, unexpected conditions, and changing operational requirements [24]. These systems struggle to adjust to unexpected sensor fluctuations, real-time anomalies, and shifting environmental parameters, leading to inefficient decision-making and resource mismanagement. The incorporation of machine learning-driven predictive control within IoT automation shows an exciting opportunity, enabling the identification of anomalies, predictive maintenance, and automated self-optimization [14]. However, implementing predictive intelligence in IoT systems endures challenges as a result of high computational demands, limited edge processing capabilities, and security risks associated with real-time AI models [21].

This research addresses important issues by creating a smart device control system based on IoT technology. This system combines real-time data gathering, secure automation processes, and advanced predictive decision-making features. This study aims to enhance the adaptability, resilience, and security of future IoT control systems by increasing data processing efficiency, strengthening security measures, and adopting smart automation techniques.

1.3 Research Objectives

As automation powered by IoT becomes increasingly complex, we need to develop control systems that are not only smart but also secure and able to grow with our needs. These systems should be capable of processing real-time data effectively, ensuring safety, and improving our ability to make informed decisions. As the IoT continues to grow in areas like healthcare, smart cities, and industrial automation, we are increasingly in need of systems that can han-

dle vast amounts of sensor data. These systems not only need to process this data efficiently but also ensure that they remain secure and adaptable to changing demands. This research is focused on finding solutions to existing challenges by creating and testing a smart device control system that uses IoT technology. We want to make sure it not only processes data effectively and securely but also adapts to different situations to make intelligent decisions. One of the main reasons for this study is to create a transparent and scalable way to manage real-time IoT data. To ensure smooth communication between connected devices, it's crucial to focus on efficient data collection and flexible synchronization techniques.

This research will explore optimization techniques to reduce data bottlenecks and enhance system performance. Special attention will be given to edge computing and fog computing strategies, which allow localized data processing to minimize latency and reduce dependence on cloud infrastructures[14, 27].

Another focus of this research is to enhance security in IoT-based automation. With the increasing dangers of cyberattacks and unauthorized access in IoT networks, this research will improve secure communication protocols, encryption methods, and real-time intrusion detection systems. It's important to have strong security measures to protect sensitive data and keep IoT automation systems trustworthy. We will look at different security methods, such as ways of verifying users, improvements using blockchain technology, and AI tools that detect threats , to reduce new risks[23, 11].

Accurate real-time data logging is essential for effectively monitoring systems, diagnosing faults, and enabling predictive analysis. This research focuses on developing strong data logging frameworks capable of enduring network interruptions and ensuring data integrity, particularly in environments with limited resources. Decentralized logging systems, including distributed ledger technology, will be assessed to improve data traceability and safeguard against unauthorized alterations[2].

Traditional rule-based control systems in IoT automation have limitations, especially in changing situations. This research looks at using machine learning algorithms to improve predictive control, detect anomalies, and make better decisions automatically. By leveraging AI-driven analytics, the system will anticipate failures, optimize resource allocation, and enhance automation efficiency. Reinforcement learning models and federated learning techniques will be incorporated to create a more intelligent and decentralized approach to decision-making[24, 14]. In order to validate the proposed system, comprehensive performance evaluations will be carried out utilizing real-world IoT scenarios alongside simulations. The study will look at important factors like system speed, how well it can grow, its security strength,

and how accurately it decides to ensure it works well in practice. Moreover, an analysis of energy consumption will be conducted to assess the system's efficiency in low-power IoT environments[21]. This research contributes to advancing next-generation IoT-based automation systems by addressing these objectives. The results will offer important perspectives on creating IoT control frameworks that prioritize efficiency, security, and real-time flexibility, leading to the development of smarter and more self-sustaining automation solutions for industrial automation, smart city initiatives, and healthcare monitoring systems.

1.4 Thesis Structure

This thesis is divided into six chapters, each addressing a specific aspect of the research. This arrangement enables obvious progress from the foundational concepts of IoT-based automation to the design, implementation, and evaluation of the proposed smart device control system.

Chapter 1 explains the research, including why the study is important, the main problem being investigated, and the key goals. It also discusses the importance of using IoT for automation and the challenges involved, such as collecting and processing data, security issues, and the need for better predictive control.

Chapter 2 provides an overview of existing research about IoT architectures, real-time data processing methods, security measures, and automation strategies using machine learning. It highlights the gaps in current studies and explains how this research fits into the larger area of IoT control systems.

Chapter 3 explains the design of the IoT-based smart device control system. It covers the choice of technologies, how data is managed, security measures, and the use of predictive analytics for better control. The chapter also describes the setup and simulation environment used to test the system.

Chapter 4 talks about setting up the system. It covers the software and hardware used in development. This includes how to set up data logging, security features, and machine learning models for automating predictions.

Chapter 5 discusses the results and analysis of how well the system works. It looks at important factors like how efficiently data is processed, how secure the system is, how well it adapts in real-time, and how accurate its decisions are. There are comparisons with traditional IoT control models to show the improvements made by the new system.

Chapter 6 concludes the thesis by highlighting the main contributions of the research. Talk

about what the findings mean, how they can be used in real-life IoT automation, and suggests possible future research to improve the scalability and security of IoT-based intelligent control systems.

Chapter 2

Literature Review

2.1 IoT-Based Automation

The *Internet of Things* (IoT) has greatly revolutionized automation by allowing devices to connect and communicate with one another, analyze data, and make smart decisions independently. IoT-based automation is widely applied in various domains, including **smart cities, healthcare, industrial control systems, and home automation**. The *Internet of Things* (IoT) improves real-time monitoring and decision-making by combining sensors, actuators, cloud computing, and artificial intelligence[27]. Although the *Internet of Things* (IoT) has made significant progress, issues related to **scalability, security risks**, and the need for **better interoperability** remain influential in its development. As the deployment of IoT devices continues to expand, the challenges associated with real-time data processing, security, and system scalability become increasingly critical. raditional automation systems, which rely on rule-based control mechanisms, struggle to adapt to dynamic environments, requiring the integration of *Artificial Intelligence* (AI) and *Machine Learning* (ML) to enhance **adaptability** and **decision-making** capabilities. This section explores the evolution of IoT automation, its key communication protocols, and the architectural frameworks that support it.

2.1.1 Evolution of IoT in Smart Systems

The evolution of IoT in smart systems can be traced back to the development of early embedded computing and machine-to-machine machine-to-machine (M2M) communication technologies. Initially, IoT applications were mostly about simple data transmission between connected devices.

However, with advancements in wireless networks, cloud computing, and big data analytics, IoT has transformed into a complex ecosystem that can process large volumes of real-time data and make predictive decisions [14].

In its early stages, IoT relied on wired sensor networks for remote monitoring and basic automation tasks. The introduction of wireless communication protocols, including **Wi-Fi, Bluetooth, Zigbee**, and **LoRaWAN**, allowed devices to communicate over longer distances without the limitations of **physical connections** [15]. The shift to **cloud-based IoT** solutions enabled the collection and storage of large datasets, which supported the development of advanced analytics and automation systems.

Modern IoT automation integrates edge computing to reduce latency and improve response times. Unlike traditional cloud-centric models, edge computing processes data closer to the source, allowing for faster decision-making and reduced bandwidth usage. This has led to the emergence of **smart manufacturing**, **intelligent traffic systems**, and **healthcare monitoring**, all of which utilize IoT-driven automation to optimize performance and improve **user experience** [13]. The upcoming stage in the evolution of the *Internet of Things* (IoT) focuses on incorporating **blockchain technology** to enhance device **authentication security, leveraging AI for automated, self-learning systems**, and utilizing quantum computing to achieve **rapid data processing**. These advancements are designed to tackle current issues concerning **security, scalability,** and **interoperability**, ultimately making IoT automation more **effective** and **robust** against cyber threats.

2.1.2 IoT Device Communication Architecture

The success of IoT-driven automation is significantly influenced by the communication protocols and system architecture that enable smooth interactions among devices, networks, and cloud systems. IoT architecture is typically structured into multiple layers, each responsible for a specific function, including **data acquisition, transmission, processing, and application deployment** [8].

1. Perception Layer: This layer includes **sensors** and **actuators** that gather **real-time environmental data**, including **temperature, humidity, motion, pressure**, and **energy consumption**. These devices transform physical signals into digital information suitable for processing. Conversely, actuators perform control actions according to the commands they receive, facilitating automation [27].

2. Network Layer: The network layer handles data transmission between IoT devices and central processing units. Various communication protocols are utilized depending on the specific requirements of the application, including:

Wi-Fi and Bluetooth: hort-range wireless communication for home automation and wearables.

Zigbee and Z-Wave: Low-power, medium-range protocols for smart home and industrial automation.

LoRaWAN and NB-IoT: Long-range, low-power communication for remote monitoring applications [15].

3. Edge and Cloud Computing: Contemporary IoT architectures often incorporate edge computing to process data closer to the source before it is sent to cloud services. This approach **decreases** latency and enhances response times for real-time needs. Cloud platforms like **AWS IoT**, **Google Cloud IoT**, and **Microsoft Azure IoT** offer centralized **data storage, analytical capabilities**, and **remote management** for extensive automation systems[14].

4. Application Layer: This layer focuses on **user engagement**, **data representation**, and **automated decision-making** processes. IoT-driven automation systems utilize **AI-powered dashboards**, **mobile applications**, and i**ndustrial control frameworks** to deliver real-time insights and operational control. Applications can vary widely, including **smart city traffic management, automated healthcare monitoring**, and **predictive maintenance** in manufacturing environments [21].

For effective communication in IoT networks, it's crucial to achieve interoperability among various devices and protocols. Initiatives like standardizing the **MQTT** (Message Queuing Telemetry Transport) protocol aim to facilitate **smooth data transfer** across different IoT environments. Moreover, the importance of **cybersecurity** cannot be overstated, as IoT devices frequently become targets for cyberattacks. Therefore, implementing robust **encryption, reliable authentication methods**, and **effective intrusion detection** systems is vital for safeguarding IoT networks [11].

The future of IoT communication and architecture will likely move towards **self-healing networks, decentralized computing models**, and **automation frameworks** that improve s**calability, security**, and **adaptability** in dynamic environments. As IoT continues to expand, creating robust and efficient communication infrastructures will be essential to unlocking the full potential of automation across different industries.

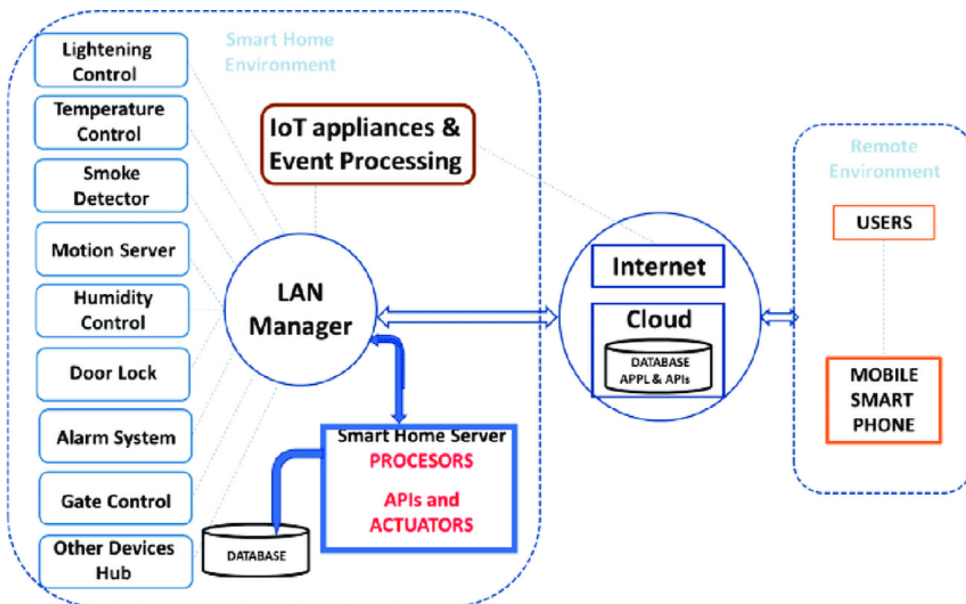

Figure 2.1: IoT-Based Smart Home Automation System. The diagram illustrates how IoT devices communicate through a LAN manager, cloud infrastructure, and smart home servers to enable automation and remote control.

2.2 Sensor Data Logging in IoT

As the automation driven by the *Internet of Things* (IoT) expands, the efficient collection and tracking of sensor data plays a vital role in ensuring reliable and scalable operations. IoT devices generate enormous streams of data, capturing a range of metrics such as **temperature, humidity, motion**, and **energy usage** in real-time. Effectively handling the surge of information is essential for facilitating **intelligent automation, efficient monitoring**, and **management** in various fields, such as **smart cities, industrial automation, precision agriculture**, and **healthcare**.

Despite the potential benefits brought by IoT, there are significant hurdles to overcome regarding data management. The sheer volume of data produced at high speed necessitates robust storage solutions and demands that organizations have real-time access. Additional challenges include **securing the data, managing processing constraints**, and **meeting the performance** needs typical of IoT datasets.

Traditional database systems often struggle with these demands due to their inability to handle the high rate of reads and writes that IoT applications require, especially when low-latency responses and efficient processing of large time-series data are essential. As a result, lightweight databases like **SQLite** have emerged as **popular choices** for IoT frameworks.

Optimizing query performance and focusing on real-time data processing are critical to enhancing system efficiency, supporting predictive analytics, and improving decision-making processes. This discussion will explore how SQLite is effectively utilized for data storage in IoT applications and will delve into advanced techniques for real-time processing and query optimization.

2.2.1 SQLite Database for Efficient IoT Data Storage

SQLite is a widely used l**ightweight, embedded**, and **self-contained database** engine, making it an excellent solution for resource-constrained IoT environments. Unlike traditional relational database management systems **(RDBMS)** that require dedicated servers, SQLite operates directly on the IoT device itself with minimal computational overhead.

Key Advantages of SQLite for IoT:

- **Minimal Resource Consumption:** Unlike MySQL or PostgreSQL, SQLite has no background processes, making it ideal for low-power IoT devices.
- **Atomic Transactions & Data Integrity:** SQLite supports **ACID**-compliant transactions, ensuring data consistency and security even in cases of power failure or unexpected shutdowns.
- **Lightweight & Self-Contained:** SQLite does not require a separate database server, reducing overhead and making it easy to integrate into IoT devices.
- **Efficient Time-Series Data Management:** Since IoT systems primarily generate time-series data, SQLite allows for fast indexing and querying of large-scale sensor readings.
- **Scalability for Edge & Cloud Computing:** SQLite efficiently manages local storage on edge devices, while cloud synchronization mechanisms allow periodic data offloading to centralized systems.

Schema Design for IoT Sensor Data Logging: A well-structured SQLite table for storing IoT sensor logs may be designed as follows:

```
CREATE TABLE sensordata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp DATETIME DEFAULT CURRENTTIMESTAMP,
    sensorid TEXT NOT NULL,
    sensortype TEXT NOT NULL,
    value REAL NOT NULL
);
```

Enhancing SQLite Performance for IoT: To improve performance, the following optimization techniques can be applied:

- **Indexing for Faster Queries:** Creating indexes on frequently queried columns enhances performance.
- **Data Partitioning:** Splitting data into smaller chunks based on time intervals improves query execution speed.
- **Write-Ahead Logging (WAL) Mode:** Enabling WAL mode improves SQLite's handling of high-frequency writes.

2.2.2 Real-Time Data Processing & Query Optimization

Challenges in Real-Time IoT Data Processing: Handling IoT sensor data in real time requires advanced processing techniques to manage:

- High-frequency data ingestion from multiple IoT devices.
- Low-latency query execution for real-time analytics.
- Efficient memory and computational resource management.
- Data consistency and security measures.

Techniques for Efficient IoT Query Execution:

1. **Using In-Memory Caching for Fast Lookups:** Frequently accessed sensor data can be cached in memory using Redis to reduce query latency.

2. **Batch Processing Instead of Single Queries:** Instead of inserting sensor data row by row, batch insertions can improve database write efficiency:

```
BEGIN TRANSACTION;
INSERT INTO sensordata (sensorid, sensortype, value) VALUES ('S
 INSERT INTO sensordata (sensorid, sensortype, value) VALUES ('
COMMIT;
```

3. **Stream Processing with Edge Computing:** Instead of sending all data to the cloud, edge computing enables real-time local analysis, reducing network congestion.

4. **Query Optimization Using Prepared Statements:** Using prepared statements minimizes redundant query parsing, optimizing database performance.

5. **Optimized Query for Real-Time Sensor Monitoring:** Retrieving the latest temperature readings from an IoT sensor can be done using:

```
SELECT timestamp, value
FROM sensordata
 WHERE sensortype = 'temperature' ORDER BY timestamp DESC
LIMIT 10;
```

Future Directions: As IoT networks continue to expand, emerging trends such as:

- AI-powered predictive analytics for anomaly detection,
- Blockchain technology for tamper-proof data logging,
- Federated learning for decentralized, privacy-preserving IoT analytics,
- 5G-enabled IoT ecosystems for ultra-low-latency data streaming,

are expected to further enhance efficiency, security, and scalability in IoT data management.

2.3 Visualization in IoT Dashboards

The growth of automation through IoT has resulted in an enormous amount of real-time data from various interconnected sensors and devices. While this raw sensor data is often stored in formats like **JSON, CSV**, or **databases**, it usually lacks the clarity required for effective decision-making. This is where visualization becomes crucial. It helps to turn complex data into actionable insights, allowing users to monitor the current status of systems, identify issues, and enhance automation processes. IoT dashboards offer **user-friendly interfaces** filled with interactive **charts, graphs**, and **notifications**, so that users can make well-informed decisions easily and efficiently [27].

IoT dashboards are widely adopted across **smart cities, industrial automation**, **environmental monitoring**, and **healthcare applications**, where real-time decision-making is crucial. A well-structured dashboard can display **live metrics, historical trends**, and **automated control options**, ensuring that IoT ecosystems operate efficiently. An ideal IoT dashboard integrates:

- **Real-time data visualization:** Displaying live sensor readings in formats such as line graphs, bar charts, and heatmaps.
- **Historical data analysis:** Comparing past trends with real-time data to optimize system performance.
- **Interactive UI elements:** Allowing users to filter, zoom, and customize displayed data.
- **Automated alerts and notifications:** Triggering real-time alerts based on predefined sensor thresholds.
- **Remote control and automation:** Enabling users to interact with actuators and adjust IoT parameters remotely.

These capabilities enhance the **scalability, accessibility, and usability** of IoT systems, ensuring that real-time data monitoring is intuitive and actionable.

2.3.1 Web-Based Monitoring Systems

Web-based monitoring systems offer an efficient way to **visualize IoT data**. They enable users to access dashboards remotely via their web browsers, eliminating the need for any software installation. Dashboards are usually set up on **cloud servers** or **local edge devices**, allowing for smooth real-time data processing and visualization.Web-based IoT dashboards have become the industry standard due to the following key benefits:

- **Remote Accessibility:** Users can monitor IoT devices from anywhere via an internet-connected device, enhancing flexibility and remote management capabilities.
- **Cross-Platform Compatibility:** Supports multiple platforms, including desktops, tablets, and smartphones, without requiring specialized software installation.
- **Scalability with Cloud Integration:** Cloud platforms such as **AWS IoT, Google Cloud IoT, and Microsoft Azure IoT** enable the handling of vast IoT data streams, providing scalable storage and processing solutions [14].
- **Real-Time Updates & Synchronization:** Web dashboards automatically refresh data without requiring manual intervention, ensuring up-to-date sensor readings and system status.
- **Security Features:** Implements secure authentication mechanisms, **role-based access controls (RBAC)**, and encrypted data transmissions to protect IoT networks against cyber threats [11].

Many industries rely on web-based dashboards for real-time monitoring and automation, including:

- **Industrial IoT (IIoT):** Monitoring factory machines, tracking production efficiency, and predicting failures to optimize industrial operations [21].
- **Smart Energy Grids:** Displaying live power consumption trends and optimizing energy distribution to improve sustainability.
- **Environmental Monitoring:** Visualizing temperature, humidity, and air quality trends to assess environmental conditions.

- **Healthcare & Wearable IoT:** Tracking patient vitals, monitoring medical devices, and ensuring remote healthcare management [13].

- **Smart Home & Security Systems:** Managing connected home devices, security cameras, and automation triggers to enhance safety and convenience.

To achieve high performance and reliability, web-based dashboards are built using modern web technologies such as **Flask, Django, Node.js, and React.js**. The following subsection discusses the advantages and trade-offs of these technologies.

2.3.2 Flask vs. Other Web Technologies

To develop an effective IoT visualization dashboard, selecting the appropriate web framework is crucial for ensuring **scalability, performance, and ease of development**. **Flask**, a lightweight **Python-based web framework**, is widely adopted for IoT dashboards due to its simplicity, RESTful API support, and integration with data processing libraries [14]. However, alternative frameworks such as **Django, Node.js, and React.js** offer different advantages.

Table 2.1: Comparison of Flask with Other Web Technologies

Feature	Flask	Django	Node.js	React.js
Language	Python	Python	JavaScript	JavaScript
Performance	Lightweight	Feature-rich but heavier	Fast (async)	Frontend-focused
Ease of Use	Simple	Built-in features	Requires config	Requires backend API
Best For	IoT dashboards	Large-scale apps	Real-time apps	UI-focused systems

Why Choose Flask for IoT Dashboards?

- **Lightweight and Flexible:** Unlike Django, Flask offers **minimal overhead**, making it ideal for **embedded IoT systems**.

- **RESTful API Support:** Simplifies the creation of APIs for retrieving sensor data and managing actuator controls.

- **Python Integration:** Seamlessly works with **NumPy, Pandas, Matplotlib, and TensorFlow** for **data analytics and visualization**.

- **WebSocket Support:** Enables real-time **bi-directional communication** between IoT devices and dashboards.

While Flask is ideal for lightweight applications, **Django** is preferred for large-scale platforms that require advanced security features, **Node.js** excels in high-frequency data stream-

ing, and **React.js** is used for dynamic front-end interfaces. The selection of the web technology stack depends on the specific requirements of the IoT application.

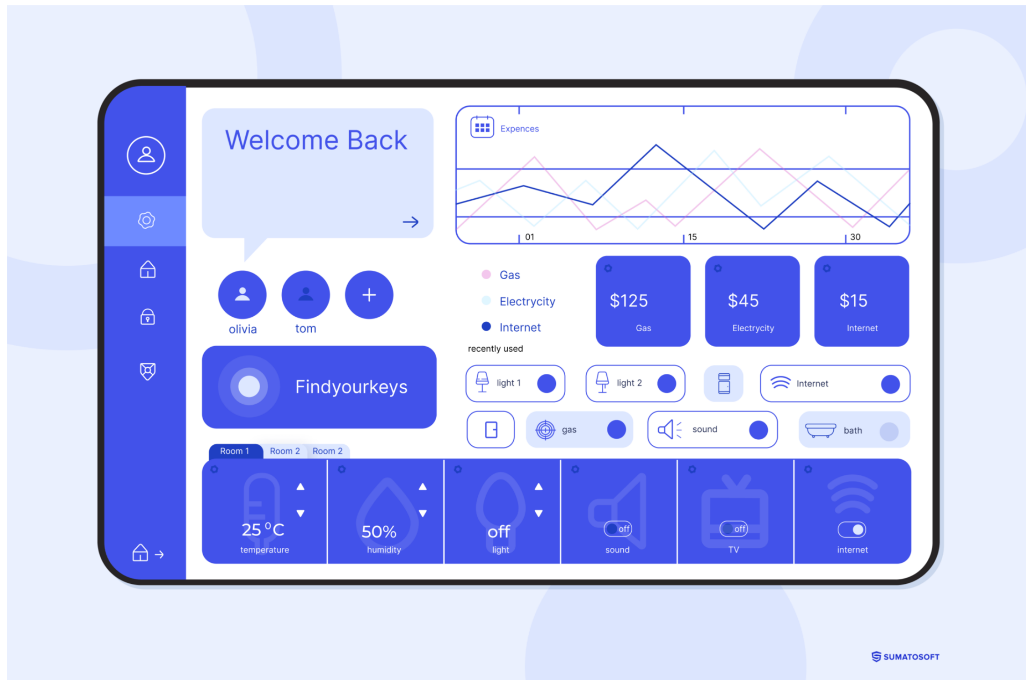

Figure 2.2: Example of a Web-Based IoT Dashboard for Real-Time Data Visualization [25].

2.4 Control Logic in IoT

The success of automation based on IoT depends significantly on the control logic that directs actuators and the processes involved in decision-making.In IoT systems, control logic dictates how devices react to environmental data, enabling automated processes to operate effectively and dependably. Conventional IoT control systems often utilize rule-based approaches, where specific thresholds determine how actuators respond.However, as IoT systems become more complex, the integration of *Machine Learning* (ML) is essential for achieving adaptive, data-driven automation [27].

2.4.1 Actuator Control Using Threshold-Based Logic

Threshold-based control logic is one of the most widely implemented methodologies in IoT automation. In this approach, actuators respond to sensor readings based on predefined conditions. For example, a **temperature sensor** in a smart home may be programmed to activate an air conditioning unit when the temperature exceeds a set threshold (e.g., 25°C) [13].

In industrial settings, humidity sensors can activate ventilation systems when moisture levels exceed safe operating thresholds. This type of control is deterministic, meaning the response is immediate and predictable. The benefits of threshold-based logic include:

- **Simplicity:** Easy to implement with minimal computational resources.
- **Reliability:** Provides consistent performance under static conditions.
- **Low Latency:** Responses occur in real-time, ensuring minimal delays in automation.

Threshold-based logic, while beneficial, does have its drawbacks. Relying on fixed thresholds can overlook varying environmental conditions, which might result in inefficiencies like unwarranted activation of actuators or missing out on slow changes. For instance, in a temperature-controlled warehouse, a static threshold might lead to inefficient cooling system performance as it fails to adjust for seasonal differences. To address these issues, implementing more sophisticated control methods, such as machine learning, is necessary [14].

2.4.2 Integration with Machine Learning for Smart Decisions

Incorporating machine learning into IoT control logic enables systems to make decisions based on data rather than relying solely on fixed rules. By examining sensor data trends over time, these algorithms can recognize patterns and anomalies, as well as determine the best responses to changes in environmental conditions. This approach improves the adaptability and efficiency of automation in IoT systems [9]. Key benefits of ML-driven control systems include:

- **Predictive Decision-Making:** ML models can forecast potential failures and optimize control strategies before an issue arises.
- **Adaptive Thresholding:** Instead of fixed limits, ML algorithms dynamically adjust thresholds based on historical data.
- **Anomaly Detection:** Identifies irregular sensor patterns that may indicate malfunctioning hardware or cybersecurity threats [17].

For example, in smart grid systems, **ML-enhanced control logic** can optimize power distribution by learning consumption patterns and predicting peak usage hours. In industrial IoT settings, predictive maintenance algorithms detect early signs of equipment degradation, triggering repairs before failures occur [21].

IoT devices continuously improve decision-making by learning from environmental feedback. **Neural networks and deep learning** models can also be trained to recognize complex relationships between multiple sensor inputs, enabling autonomous decision-making without predefined rules [14]. To effectively implement ML-based control systems, IoT devices require:

- **Edge Computing:** Processing data closer to the source to reduce latency.
- **Cloud-Based AI Models:** Leveraging cloud resources for complex model training.
- **Secure Data Pipelines:** Ensuring encrypted communication between IoT sensors and ML processing units [11].

Although ML-based IoT control systems offer significant advantages, challenges such as **high computational requirements, data privacy concerns, and model interpretability** remain. Future research aims to enhance energy-efficient AI models and develop secure, decentralized IoT architectures for intelligent automation.

2.5 IoT Security and Performance

The *Internet of Things* (IoT) plays a vital role in today's automation landscape, allowing for the creation of advanced environments through a network of interconnected devices.As the number of IoT devices increases, concerns about security and system performance also rise. IoT ecosystems handle vast amounts of sensitive data, making them prime targets for cyber threats, breaches, and unauthorized access. Ensuring robust security and optimal performance are fundamental to IoT's long-term viability, particularly as devices become more sophisticated and applications demand real-time responsiveness [11].

Unlike conventional computing environments, IoT networks consist of **diverse and resource-constrained devices** with varying processing power, memory, and communication capabilities. This makes implementing traditional security and performance enhancement mechanisms challenging. Many IoT devices are deployed in remote or harsh environments, making it difficult to perform firmware updates, security patches, and real-time monitoring. Moreover, as IoT networks expand, they generate enormous volumes of data that require **secure transmission, efficient storage, and real-time analysis** [28]. The need for scalable security measures that do not compromise system performance is more critical than ever.

Addressing these challenges requires **a multi-layered approach to security and performance optimization**, integrating advanced encryption techniques, secure data transmission protocols, and robust API management strategies. In this section, we discuss critical security and performance-enhancing methods, including data encryption, protecting IoT data during transmission and storage, and ensuring secure access to IoT APIs through rate limiting and security hardening mechanisms.

2.5.1 Data Encryption Methods in IoT

Encryption is important for keeping sensitive information safe in IoT systems. It protects data by making it unreadable to anyone without the right key. Since many IoT devices do not have a lot of processing power or battery life, it is important to choose the right encryption method. This way, you can keep data secure without slowing down the devices [20]. **Symmetric vs. Asymmetric Encryption in IoT**

IoT systems primarily use **symmetric and asymmetric encryption algorithms** for data protection.

Symmetric encryption algorithms (such as AES-128, AES-256) are computationally efficient and widely used in IoT applications due to their fast processing speeds. They use a single key for both encryption and decryption, making them ideal for **low-power IoT devices** where performance efficiency is critical [2]. - **Asymmetric encryption algorithms** (such as RSA and Elliptic Curve Cryptography (ECC)) provide higher security but are computationally expensive. Unlike symmetric encryption, asymmetric encryption uses a pair of keys (public and private), which makes it more suitable for **secure IoT authentication, digital signatures, and key exchange protocols** [10].

End-to-End Encryption and Lightweight Cryptography

As IoT networks continue to expand, implementing **end-to-end encryption (E2EE)** has become essential to protect data from unauthorized access at every stage of transmission. This approach ensures that only authorized devices can decrypt transmitted data, significantly reducing the risk of data interception by malicious actors. **Lightweight cryptographic algorithms**, such as PRESENT and SPECK, have also been developed specifically for IoT devices, balancing security with minimal resource consumption [26].

The adoption of **post-quantum cryptography** is another emerging trend, aimed at securing IoT networks against potential quantum computing threats. As quantum computers advance, traditional encryption methods may become vulnerable, necessitating new cryptographic protocols designed to withstand quantum decryption techniques.

2.5.2 Protecting Data Transmission and Storage

Keeping IoT data secure when it is being sent or stored is very important. IoT devices send a lot of real-time information, and if this data is intercepted or changed, it can harm the entire system's trustworthiness and effectiveness. Protecting data transmission and storage requires a combination of **secure communication protocols, access control mechanisms, and distributed storage solutions** [4].

Securing Data Transmission

IoT devices often rely on wireless communication protocols such as **Wi-Fi, Bluetooth, Zigbee, LoRaWAN, and NB-IoT**. These protocols, while efficient for connectivity, introduce vulnerabilities related to **eavesdropping, replay attacks, and unauthorized data modification** [6]. To mitigate these risks, IoT networks must implement secure transmission protocols:

TLS/SSL Encryption: Ensures secure data transfer over the internet by encrypting communications between IoT devices and cloud platforms.

MQTT with Transport Layer Security (TLS): A widely used protocol for IoT messaging that provides lightweight and encrypted message delivery [22].

CoAP with Datagram TLS (DTLS): A security-enhanced protocol for constrained IoT devices that require efficient and secure communication.

Additionally, **blockchain technology** is being explored for IoT security to enable **decentralized and tamper-proof data verification**. By leveraging distributed ledger systems, IoT applications can securely store and validate transactions without relying on a single centralized entity [1]. **Secure IoT Data Storage**

IoT applications generate enormous amounts of **historical and real-time data**, necessitating secure storage solutions that prevent unauthorized access. Cloud storage services such as **AWS IoT Core, Microsoft Azure IoT Hub, and Google Cloud IoT** offer robust storage solutions with built-in security features. However, **on-premises and edge storage solutions** are gaining traction for organizations that require **low-latency data access and greater control over security**.

To enhance security, organizations are implementing **hardware security modules (HSMs)** and **Trusted Execution Environments (TEEs)** to provide tamper-resistant storage for cryptographic keys and sensitive data. These security measures help protect **IoT credentials, authentication tokens, and critical system logs** from unauthorized modifications [19].

2.5.3 API Rate Limiting and Security Hardening

IoT platforms heavily rely on **Application Programming Interfaces (APIs)** to facilitate communication between devices, cloud services, and user applications. However, improperly secured APIs can serve as entry points for cyberattacks, including **DDoS (Distributed Denial-of-Service) attacks, injection attacks, and unauthorized data access** [5]. To address these threats, implementing **API rate limiting, authentication mechanisms, and security-hardening techniques** is crucial for protecting IoT infrastructure. **Rate Limiting Strategies for API Security**

Rate limiting controls the number of requests a client can send within a specific time frame, preventing system overload and malicious exploitation. Common rate limiting techniques include:

Token Bucket Algorithm: Assigns tokens to each API request, rejecting excessive requests once the token limit is exceeded.

Fixed Window Limiting: Restricts requests within fixed time intervals to prevent system abuse.

Sliding Window Rate Limiting: Provides a more flexible approach by dynamically adjusting request limits based on past usage patterns [1].

2.5.4 Authentication & Authorization for API Security

To further enhance API security, **strong authentication and authorization mechanisms** must be in place to prevent unauthorized access and data breaches. In IoT environments, where multiple devices interact with cloud services and APIs, implementing secure authentication frameworks is essential to ensure reliable and protected communication.

One of the widely adopted authentication models in IoT is **OAuth 2.0 and OpenID Connect (OIDC)**. These protocols enable secure authorization by allowing IoT devices to authenticate with minimal exposure of sensitive credentials. OAuth 2.0 is particularly useful for granting limited access permissions to third-party applications while maintaining strict security policies.

Another common security approach is the use of **JSON Web Tokens (JWTs)**. JWTs provide a lightweight and stateless method for secure API authentication, allowing encrypted tokens to be exchanged between clients and servers. This method reduces the risks associated with traditional session-based authentication and prevents unauthorized access to IoT networks [28].

To further strengthen API security, **Mutual TLS (mTLS)** is implemented to establish a secure communication channel between IoT devices and servers. Unlike standard TLS, which authenticates only the server, mTLS requires both client and server authentication to verify each other's identity before allowing data exchange. This adds an additional layer of security, particularly for industrial IoT applications that demand high levels of integrity and confidentiality.

Additionally, IoT security hardening techniques such as **code obfuscation, secure API gateways, and real-time threat monitoring** play a crucial role in preventing cyberattacks. By using **intrusion detection systems (IDS)** and **machine learning-based anomaly detection**, organizations can actively monitor IoT traffic and identify potential threats before they compromise system security.

Ensuring a secure API environment in IoT systems is fundamental for protecting sensitive data and preventing unauthorized manipulations. By integrating these authentication and authorization measures, IoT networks can establish a secure foundation for scalable and resilient automation systems.

2.6 Gaps & Opportunities

Despite significant advancements in IoT-based automation, several challenges persist, limiting its widespread adoption and efficiency. These challenges present opportunities for future research and innovation, particularly in improving scalability, security, adaptability, and decision-making capabilities in IoT control systems.

2.6.1 Current Limitations in IoT Control Systems

Traditional IoT control systems use fixed rules to operate, which makes them inflexible in changing situations. They work well for routine tasks but have trouble making quick decisions when things are unpredictable. One big problem is their struggle to handle large amounts of different types of data efficiently. As more IoT devices produce enormous streams of real-time sensor data, current control systems find it hard to manage and respond to this information quickly [14].

Security remains another major bottleneck in IoT implementations. The lack of end-to-end encryption, weak authentication mechanisms, and susceptibility to Distributed Denial of Service (DDoS) attacks create vulnerabilities that compromise the integrity of IoT networks [11]. Many IoT control systems have weak anomaly detection and self-repair features, which makes them more reactive than proactive in stopping security breaches. Another big issue is that IoT devices from different brands often use their own protocols, making it hard for them to communicate and work together. The lack of common standards for IoT communication makes it difficult to implement these systems on a large scale in industries [21]. Energy efficiency is important because battery-powered IoT sensors need to use less power so they can last longer and don't need to be maintained often.

Chapter 3

System Design and Implementation

3.1 System Architecture

The Smart IoT Dashboard is designed to easily connect **real-time monitoring, effective data logging**, and **smart automation**. Because IoT-based automation changes quickly, it is important to design a system that can grow, work well, and adapt over time. The main goal of this system is to create a connected setup where virtual IoT devices can produce, handle, save, and show sensor data in real-time. This way, users can keep an eye on environmental conditions, look for patterns, and start automated actions based on set rules.

The three-layered design of the system consists of:

- **IoT Device Simulation Layer:** Simulates temperature, humidity, and motion sensors while integrating real-world data from external sources.
- **Data Logging and Storage Layer:** Ensures structured, secure, and efficient storage of sensor readings in an SQLite database.
- **Visualization and Control Layer:** Displays real-time sensor data on a web-based dashboard, enabling dynamic user interaction and automated actuator control.

These three layers work together to **maintain system integrity, ensure fast data retrieval, and support intelligent automation**. The following sections describe in detail how the **IoT device simulation workflow** operates and how **sensor data, storage, and dashboard interaction** occur within the system.

3.1.1 IoT Device Simulation Workflow

In real-life IoT systems, sensors gather data from the environment and send it to a central unit for analysis. For this project, we chose to simulate how sensors act instead of using actual devices.This approach allows for **scalability, flexibility in testing different scenarios, and easier system optimizations** without the limitations imposed by physical hardware constraints.

The IoT device simulation workflow is structured into multiple stages to ensure seamless data generation, validation, storage, and automated response execution. The IoT simulation workflow follows a structured process consisting of the following stages:

1. Data Generation The system fetches real-time weather data from an external API, providing temperature and humidity readings. Motion detection is probabilistically simulated using a 10% occurrence rate per cycle, ensuring sporadic yet realistic motion detection events. Each generated reading is assigned a timestamp to maintain data accuracy.

2. Data Preprocessing Sensor readings undergo preprocessing to ensure consistency and reliability:

- Missing or inconsistent values are filtered out.
- Timestamps are standardized for uniform time synchronization.
- Readings are rounded to two decimal places to improve readability.
- Threshold-based event detection flags anomalies for automated responses.

3. Data Storage Validated sensor readings are stored in an SQLite database (`sensor_data.db`), maintaining structured logs that include:

- Sensor ID: Identifies the data source.
- Timestamp: Stores the exact time of measurement.
- Temperature & Humidity: Captures environmental conditions.
- Motion Status: Indicates movement detection.
- Actuator Status: Logs whether devices such as fans or lights were activated.

4. Data Transmission via API A Flask-based REST API enables secure data retrieval and system interaction through endpoints such as:

- `/api/latest-readings` – Retrieves the most recent sensor entry.
- `/api/historical-readings?timeframe=week` – Fetches sensor logs for a given timeframe.

5. Automated Actuator Control Based on predefined control rules:

- The fan activates if temperature exceeds 25°C.
- An alert is triggered if humidity surpasses 70%.
- Motion detection at night results in a security notification.

To ensure real-time responsiveness, new sensor data is generated every three seconds.

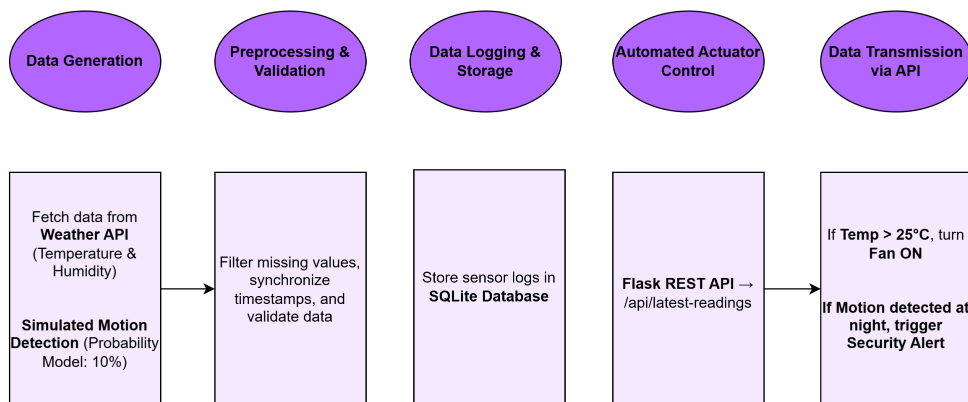

Figure 3.1: IoT Device Simulation Workflow: This figure illustrates the step-by-step process of IoT sensor simulation, from data acquisition to automated actuator control.

3.1.2 Interaction Between Sensors, Data Logger, and Dashboard

A well-structured interaction between sensors, the SQLite database, and the web dashboard ensures **real-time data accuracy, efficient storage, and dynamic visualization**. The interaction process follows a structured pipeline:

1. Sensor Data Generation The simulated sensors continuously generate environmental readings based on **real-world API data and probability-based motion detection**. Each reading is timestamped and processed to determine if it exceeds predefined automation thresholds.

2. Data Logging in SQLite All validated sensor readings are persistently stored in the database, enabling:

- Reliable historical records for analysis.
- Fast query retrieval for real-time visualization.
- Structured data management to prevent data loss.

3. Backend API Processing The Flask-based backend API:

- Processes sensor data requests from the frontend.
- Provides RESTful API endpoints for real-time and historical data retrieval.
- Ensures secure data transmission and access control mechanisms.

4. Frontend Dashboard and Visualization The web dashboard provides a **centralized interface for real-time monitoring and automation**. It offers:

- Live updates through dynamic charts.
- Historical trend visualization using **Chart.js**.
- Alerts for motion detection, temperature fluctuations, and high humidity.
- A configurable time selector allowing users to view:
  - The last 24 hours.
  - The last week.
  - The last month.
  - The last year.

5. Automated Actuator Responses The system autonomously controls actuators when sensor readings exceed predefined thresholds:

- The fan turns ON if temperature rises above 25°C.
- The light turns ON if motion is detected at night.
- Alerts are issued when environmental conditions become critical.

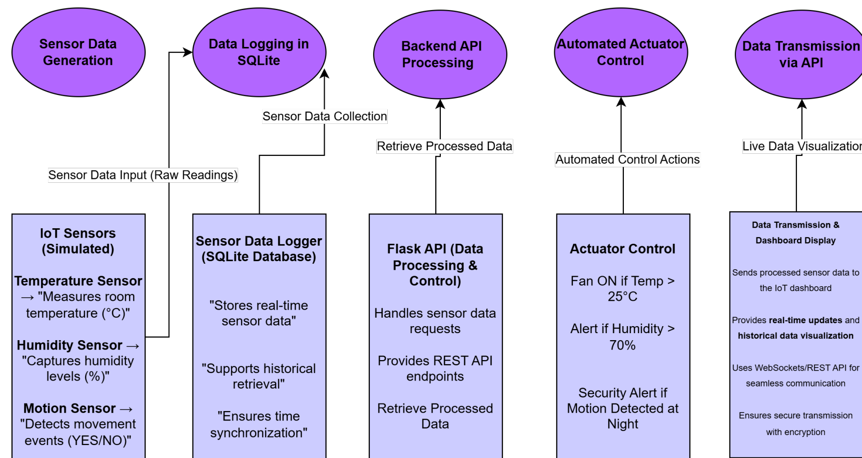

Figure 3.2: Interaction between Sensors, Data Logger, and Dashboard

The designed architecture ensures a highly scalable and responsive IoT system, providing **efficient data collection, real-time visualization, and intelligent automation**.

3.1.3 Secure User Authentication & OTP Verification

The Smart IoT Dashboard incorporates a secure authentication process, including OTP-based verification, to ensure that only authorized users can access the system. This approach strengthens account protection and provides a safer, more reliable user experience.

Signup Process: The signup process starts when a user enters their email address and sets a password. After submitting this information, the system generates a unique OTP and sends it to the user's email. This OTP acts as an additional verification step, ensuring the authenticity of the account before it is fully activated.

OTP Verification: To verify their account, the user enters the received OTP into the verification form. The system then checks the entered OTP against the one it generated. If the OTP is correct and has not expired (within a one-minute validity window), the account is verified and securely stored in the database. This step ensures that only users with valid email addresses and matching OTPs can complete the registration process.

Login & Session Management: Once the account is verified, the user can log in by entering their email and password. The system validates the entered credentials by comparing them to securely stored data in the database. If the credentials match, the system creates a session for the user, enabling persistent login. This session-based authentication allows the user to remain logged in without needing to re-enter their credentials each time, making the experience both convenient and secure.

Security Features: The authentication system incorporates multiple security measures to safeguard user data:

- **Password Hashing:** All user passwords are hashed using bcrypt, a strong password-hashing algorithm. This ensures that even if the database is compromised, the original passwords cannot be easily retrieved.
- **Session-Based Authentication:** Persistent sessions are created upon successful login. These sessions are securely stored and managed, providing a seamless user experience without compromising security.
- **OTP Expiration and Resend Option:** To prevent stale OTPs from being used, each OTP is set to expire after one minute. If an OTP expires, users have the option to request a new one, ensuring that the verification process remains secure and timely.

3.2 Technologies Used

The Smart IoT Dashboard is a multi-layered system that integrates real-time data collection, processing, and visualization to enable seamless monitoring of IoT devices. Developing such a system requires a carefully selected technology stack that ensures:

- **Scalability** to handle increasing sensor data loads.
- **Efficiency** in processing and storing real-time information.
- **Security** to prevent unauthorized access to sensitive data.
- **User-Friendly Interfaces** for intuitive interaction with the IoT system.

To achieve these goals, the Smart IoT Dashboard is designed using Python, Flask, Chart.js, Bootstrap, and SQLite—each playing a distinct role in ensuring high-performance *Internet of Things* (IoT) monitoring and automation.

3.2.1 Python for Data Processing

Python is widely regarded as the go-to programming language for IoT applications due to its simplicity, versatility, and rich ecosystem of libraries that support data acquisition, processing, and analysis [16]. In this project, Python serves as the backend engine for:

- **Real-time Sensor Data Handling** – Python scripts continuously fetch data from simulated IoT sensors (temperature, humidity, and motion detection).
- **Database Management** – Using `sqlite3`, the system efficiently logs, retrieves, and updates sensor data in an embedded SQLite database.
- **Data Preprocessing** – Python's `pandas` library is utilized for filtering noisy data, normalizing and structuring data, and conducting time-series analysis.
- **API Interaction** – Python's **Flask!** (**Flask!**) framework exposes RESTful endpoints for real-time data retrieval and actuator control.
- **Automation & Control Logic** – Python executes predefined rules such as:
  - Turning on the fan if the temperature exceeds 25°C.
  - Triggering a security alert if motion is detected at night.

Advantages of Using Python in IoT

Python has become a widely adopted programming language in the field of IoT due to its flexibility, ease of use, and extensive library support. Some of the key advantages of using Python for IoT applications include:

- **Platform Independence** – Python runs seamlessly across multiple platforms, from embedded devices such as Raspberry Pi to cloud-based servers. This ensures compatibility with a variety of IoT infrastructures.
- **Scalability** – Python can handle increasing data loads as IoT sensor networks expand. Its ability to manage large-scale data processing makes it suitable for real-time IoT applications.
- **Large Developer Community** – Python is supported by a vast community of developers, with thousands of open-source libraries available for IoT development. This active ecosystem accelerates problem-solving and provides extensive documentation for efficient development.

- **Rich Library Support** – Python offers built-in libraries for data handling, networking, and machine learning, such as `pandas` for data manipulation, `flask` for web APIs, and `scikit-learn` for predictive analytics.
- **Ease of Integration** – Python seamlessly integrates with IoT communication protocols, databases, and cloud services, enabling efficient data flow between devices and platforms.

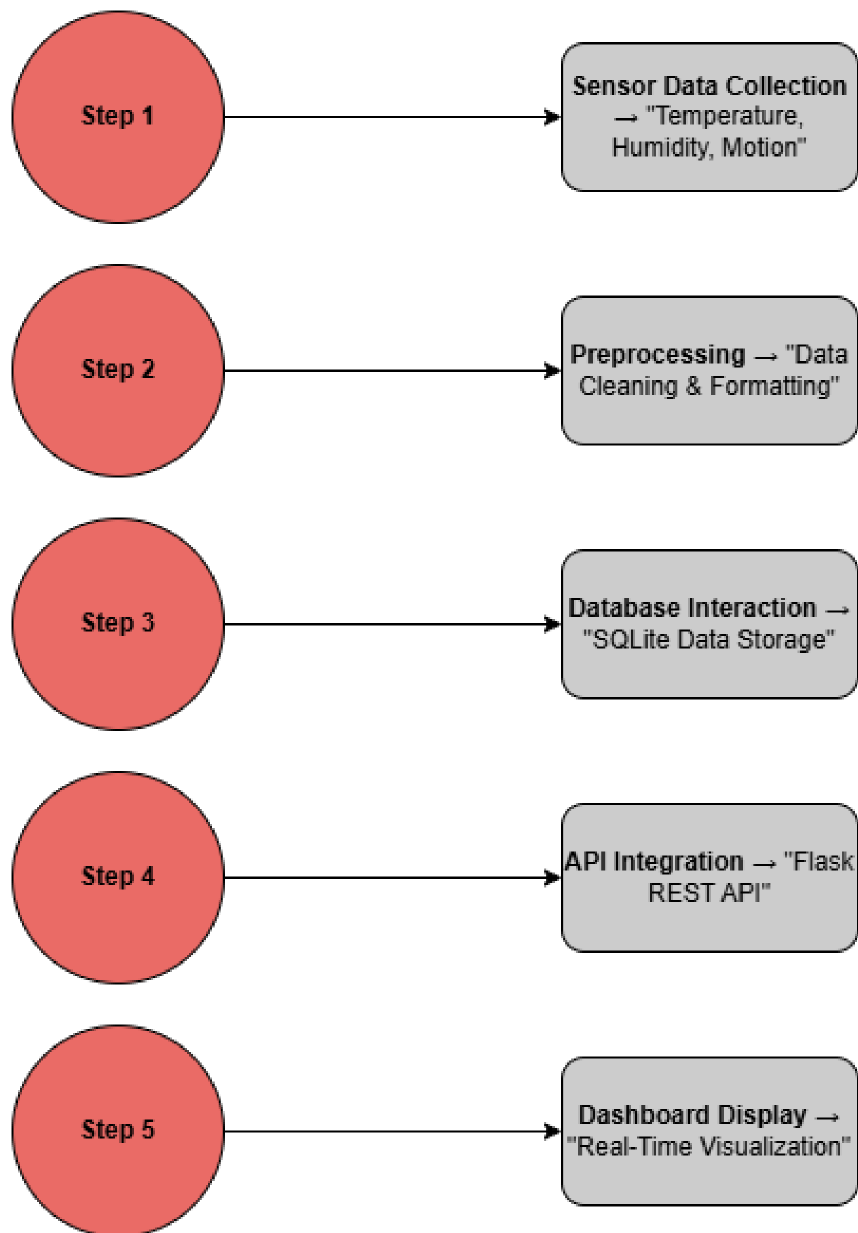

Figure 3.3: Workflow of Python in IoT Data Processing: From Sensor Collection to Visualization

3.2.2 Flask for Web-Based IoT Dashboard

Flask is a lightweight and modular web framework that provides the backend infrastructure for serving real-time IoT data through a RESTful API [7]. Flask ensures:

- Low-latency responses for real-time data updates.
- Modular expansion with additional IoT devices and cloud storage.
- Secure API handling, including authentication tokens and encrypted data transmission.

The API includes:

- `/api/latest-readings` – Fetches the most recent sensor readings.
- `/api/historical-readings?range=week` – Retrieves sensor logs over a specified period.
- `/api/control/actuator` – Allows users to trigger actuators remotely.

3.2.3 Chart.js & Bootstrap for Front-end Visualization

The front-end interface of the Smart IoT Dashboard is built using **Chart.js** for **interactive data visualization** and **Bootstrap** for **designing a responsive UI**. **Chart.js** allows the system to:

- Display live temperature & humidity graphs that update automatically.
- Visualize historical sensor data with user-selected time filters.
- Provide motion event logs with timestamps.

Bootstrap ensures:

- Mobile-friendly design that adjusts for different screen sizes.
- Clean UI elements, including interactive buttons and forms.
- A consistent, professional look for the dashboard.

3.2.4 SQLite Database for Sensor Data Logging

SQLite is an embedded relational database system chosen for storing IoT sensor data due to its:

- Minimal resource requirements, making it suitable for embedded IoT systems.
- Fast read/write operations optimized for local data storage.
- ACID compliance, ensuring data consistency even in case of power failures [12].

Column Name	Data Type	Description
`id`	INTEGER (Primary Key)	Unique identifier for each reading
`timestamp`	DATETIME	Time of measurement
`temperature`	FLOAT	Temperature in °C
`humidity`	FLOAT	Humidity in %
`motion_detected`	BOOLEAN	Whether motion was detected (YES/NO)
`fan_state`	BOOLEAN	Status of the fan (ON/OFF)
`light_state`	BOOLEAN	Status of the light (ON/OFF)

Table 3.1: SQLite Database Schema for Sensor Data Logging

3.3 Sensor Simulation Design

The sensor simulation approach in this system is designed to realistically model IoT-based environmental monitoring by generating synthetic **temperature, humidity**, and **motion sensor data** while integrating real-time weather information. This approach allows for a **scalable**, flexible, and reliable IoT automation framework that can operate in real-world conditions. By blending synthetic and real-world data, the system provides **accurate** and **continuous** environmental monitoring without relying solely on physical sensors.

The design of this simulation consists of two primary components:

- **Synthetic Sensor Data Generation:** Simulated IoT sensors generate temperature, humidity, and motion data using mathematical models and probabilistic functions.
- **Real-World Data Integration:** The system retrieves live weather data from an API and blends it with simulated data to enhance accuracy.

This section explores the detailed implementation of these methodologies and their role in ensuring **real-time data streaming, event detection**, and **intelligent automation**.

3.3.1 Simulating Temperature, Humidity, and Motion Sensors

Simulated sensors play a critical role in IoT-based systems, enabling real-time monitoring, analysis, and automation without the limitations of physical devices. In this system, three primary environmental parameters are simulated:

- **Temperature:** The simulated temperature follows a pattern that mimics natural variations throughout the day. The values change dynamically to reflect typical daily fluctuations, ensuring that the dataset appears realistic.
- **Humidity:** The simulated humidity values are generated in correlation with temperature. By ensuring that humidity levels change based on environmental temperature, the system maintains **consistency in sensor behavior**.
- **Motion Detection:** Instead of physical motion sensors, the system employs probabilistic modeling to simulate movement. Motion events occur at random intervals, with a higher frequency during active hours (e.g., daytime) and a lower probability at night.

Each of these simulated sensors **operates continuously,** generating data at predefined intervals and storing it in an SQLite database for **real-time analysis and historical retrieval.**

3.3.2 Real-Time Weather API Integration for Environmental Data

To further improve accuracy, the system integrates real-time environmental data from a public weather API. This ensures that the simulated IoT environment reflects actual conditions rather than relying solely on synthetic values.

3.3.2.1 Fetching Live Weather Data

The system queries an external weather API at regular intervals, retrieving temperature, humidity, and atmospheric conditions. The API response is structured in JSON format, allowing for easy parsing and integration into the database.

By continuously updating sensor values with live data, the system remains adaptive to changing weather conditions, ensuring that automation triggers (such as activating actuators based on temperature) remain valid under real-world scenarios.

3.3.2.2 Blending Simulated and Real-World Data

Rather than using only simulated or only real-world data, the system adopts a hybrid model, allowing both sources to contribute to final sensor readings. This blended data model provides a more flexible and reliable representation of environmental conditions.

This approach benefits IoT automation by:

- Maintaining realism in sensor behavior.
- Reducing dependency on physical sensor hardware.
- Supporting adaptive environmental modeling, making the system applicable in different scenarios.

3.3.2.3 Storing and Visualizing the Data

The combined dataset consisting of both simulated and real-time API readings is stored in an SQLite database, which allows for historical analysis and predictive modeling. The web-based IoT dashboard retrieves and visualizes this data, providing users with real-time updates and trend analysis tools.

3.3.3 Advantages of the Sensor Simulation Approach

The integration of simulated and real-world sensor data offers several advantages:

By leveraging this **hybrid IoT data model**, the system provides a **robust, scalable**, and **cost-effective** approach to environmental monitoring in smart automation frameworks.

- **Scalability:** The system can be expanded to multiple locations without requiring additional hardware.
- **Flexibility:** Users can modify sensor parameters to suit different IoT applications.
- **Cost-Efficiency:** The need for physical sensors is minimized, reducing hardware expenses.
- **Reliability:** The **blended data approach** ensures that the system remains accurate under different environmental conditions.

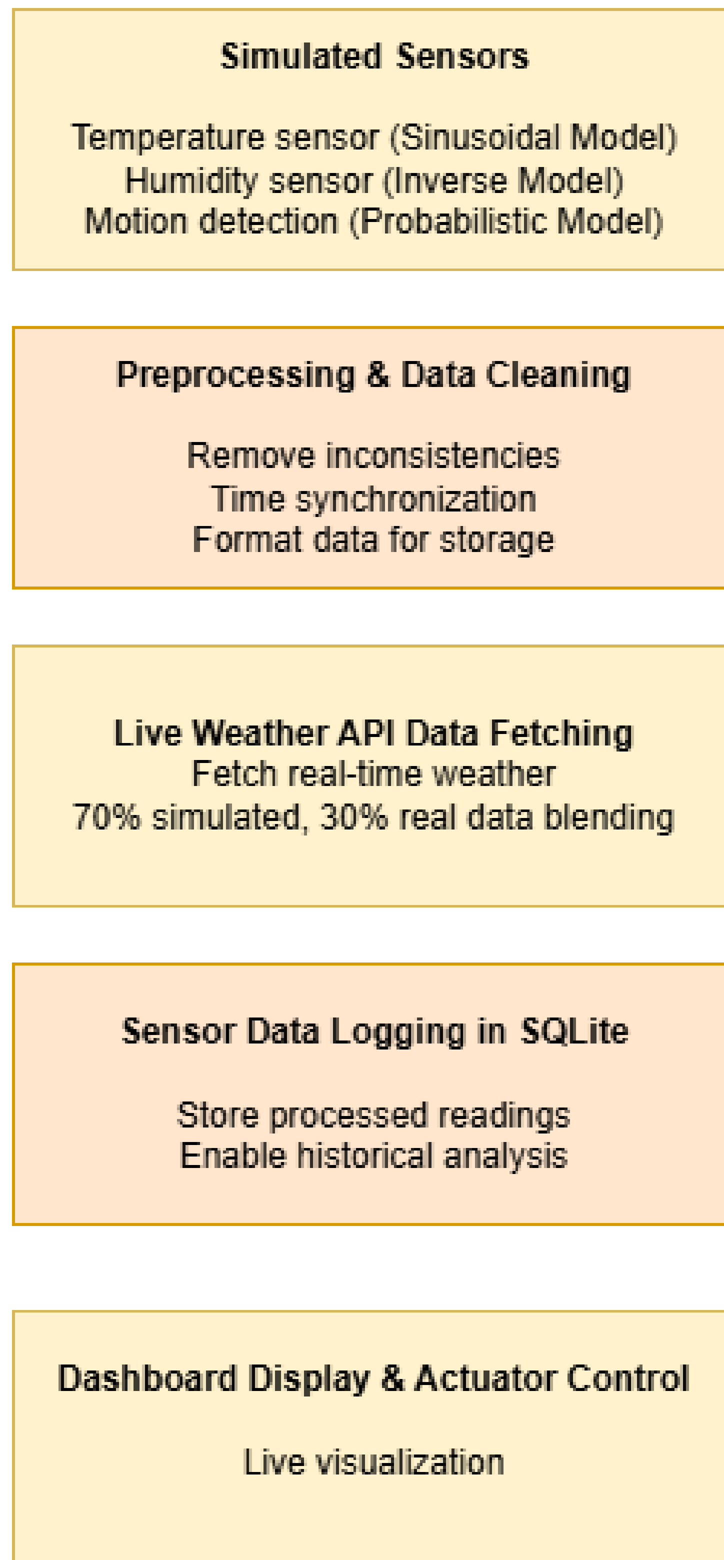

Figure 3.4: Sensor simulation workflow. It represents the generation, processing, and storage of IoT sensor data.

```
PROBLEMS 4    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    COMMENTS

PS C:\Users\Zahra Shahrooei\smart_iot> python sensor_simulator.py
Data Logged: 2025-03-04 21:20:35, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
Data Logged: 2025-03-04 21:20:45, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
Data Logged: 2025-03-04 21:20:58, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
Data Logged: 2025-03-04 21:21:08, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
Data Logged: 2025-03-04 21:21:21, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
Data Logged: 2025-03-04 21:21:34, Temp: 4.3°C, Humidity: 81%, Motion: YES, Fan: OFF, Light: ON
```

Figure 3.5: Real-time sensor data logging output. The system records temperature, humidity, and motion detection at regular intervals, ensuring continuous data collection.

3.4 Data Storage & Management

Storing and managing data well is crucial for making sure IoT systems work reliably and correctly. The **Smart IoT Dashboard** is built to manage a steady flow of real-time sensor data while also keeping past data for analysis. To achieve this, the system uses an **SQLite database**, which is lightweight and allows for quick data retrieval. This section will explain the database structure, how data is **added, tips for improving queries, and methods for long-term data management** to keep the system efficient and able to grow.

3.4.1 SQLite Database for IoT Data

SQLite is selected as the database management system for this project due to its lightweight nature, easy integration, and capability to handle structured sensor data efficiently. It is embedded into the application, eliminating the need for a separate database server while ensuring a robust and persistent data storage solution.

3.4.1.1 Database Schema Design

The database schema is designed to store real-time sensor readings, actuator states, and environmental data. The primary table, `sensor_log`, is structured as follows:

Table 3.2: Sensor Log Table Schema

Column Name	Data Type	Description
id	INTEGER (Primary Key)	Unique identifier for each record
sensor_id	TEXT	Identifier for the sensor source
timestamp	TEXT	Date and time of the sensor reading
temperature	REAL	Temperature value in Celsius
humidity	REAL	Humidity percentage
motion	TEXT	Motion detection state (YES/NO)
fan	TEXT	Fan state (ON/OFF)
light	TEXT	Light state (ON/OFF)

The `timestamp` column follows the ISO 8601 format (YYYY-MM-DD HH:MM:SS), ensuring accurate chronological ordering of data.

3.4.1.2 Data Insertion Strategy

Sensor readings are inserted into the database at fixed intervals of 10 seconds, simulating a real-world IoT logging system. The data is inserted using parameterized SQL queries to prevent injection attacks and ensure efficient execution.

```sql
INSERT INTO sensorlog (sensorid, timestamp, temperature, humidity, motion
    , fan, light)
VALUES (?, ?, ?, ?, ?, ?, ?);
```

Listing 3.1: Inserting Sensor Data into SQLite

By continuously logging sensor data, the system maintains a historical record, enabling trend analysis, predictive analytics, and anomaly detection.

3.4.1.3 Data Retention & Cleanup Policy

Since IoT systems generate large amounts of data over time, a **data retention policy** is applied to prevent excessive database growth. The Smart IoT Dashboard retains only the **most recent 30 days of sensor readings**, while older records are archived or deleted.

To maintain database efficiency, the following SQL query is executed periodically to remove outdated records:

```sql
DELETE FROM sensorlog WHERE timestamp ¡ DATETIME('now', '-30 days');
```

Listing 3.2: Deleting Old Data Beyond Retention Period

Additionally, **data backup and archival mechanisms** are implemented to ensure that historical sensor logs are not lost. These mechanisms allow old sensor readings to be **exported to CSV files** for offline analysis and long-term trend monitoring. This approach ensures that the system maintains a **balance between real-time performance and historical data preservation**.

3.4.2 Query Optimization for Real-Time Performance

An IoT dashboard needs an efficient database to respond quickly and provide a smooth experience. Unlike regular databases that check data at set times, IoT systems constantly create and update data, which means they need fast query processes. If the amount of data grows too large without proper management, it can slow down queries, leading to delays in visualizing information, decreased responsiveness, and higher resource use.

To tackle these issues, the system uses various strategies designed specifically for managing IoT data. These strategies focus on retrieving data as quickly as possible while keeping the database reliable and able to grow as needed.

3.4.2.1 Indexing for Faster Data Lookup

Indexes are created on frequently queried columns, such as `timestamp` and `sensor_id`, allowing the database engine to retrieve records significantly faster than a full table scan. Without indexing, each query would require scanning the entire dataset, leading to slower performance as data volume grows.

```
1 CREATE INDEX idxtimestamp ON sensorlog (timestamp);
```

Listing 3.3: Creating an Index on the Timestamp Column

This indexing method ensures that queries fetching the latest sensor readings execute with minimal latency, improving the overall responsiveness of the IoT dashboard.

3.4.2.2 Query Scope Limitation to Reduce Overhead

Instead of retrieving all stored sensor readings, queries are optimized to fetch only the most recent and relevant data. This reduces computational load and ensures that the dashboard remains responsive under high-frequency data updates.

```
1 SELECT * FROM sensorlog ORDER BY timestamp DESC LIMIT 50;
```

Listing 3.4: Retrieving Only the Latest 50 Sensor Readings

By limiting the number of returned records, the system avoids unnecessary memory usage and accelerates response time.

3.4.2.3 Optimized Batch Processing for Bulk Inserts

Since IoT devices generate a high frequency of sensor readings, writing data to the database efficiently is essential. Instead of executing individual insert queries for each reading, the system groups multiple insertions into a single transaction, reducing the number of disk I/O operations.

```
1 BEGIN TRANSACTION;
2 INSERT INTO sensorlog (sensorid, timestamp, temperature, humidity, motion
    ) VALUES (?, ?, ?, ?, ?);
3 INSERT INTO sensorlog (sensorid, timestamp, temperature, humidity, motion
    ) VALUES (?, ?, ?, ?, ?);
4 COMMIT;
```

Listing 3.5: Batch Insert for Improved Write Performance

By batching insert operations, the system significantly reduces the time required for database writes, optimizing performance for high-frequency data logging.

3.4.2.4 Implementing Asynchronous Data Fetching for Real-Time Updates

To prevent performance degradation due to blocking database requests, the frontend dashboard employs asynchronous AJAX-based queries to fetch sensor data without requiring a full-page refresh. This ensures that real-time updates do not interfere with user interactions.

- WebSockets and REST API endpoints are utilized for efficient client-server communication.
- The dashboard refreshes sensor data at defined intervals, retrieving only new entries instead of reloading the entire dataset.

This method enables seamless real-time monitoring, providing continuous visibility into IoT device activity.

By integrating these **query optimization techniques**, the system achieves **low latency, high efficiency, and real-time responsiveness**, allowing the IoT dashboard to handle increasing data volumes without performance degradation. These methods ensure that users can **monitor sensor activity in real-time, execute control logic efficiently, and scale the system for future expansion** without experiencing bottlenecks.

3.5 Control Logic & Actuator Automation

Controlling devices like **fans and lights** automatically is an important part of smart systems that use the *Internet of Things* (IoT). This helps manage environmental conditions efficiently without needing people to intervene. The Smart IoT Dashboard uses set limits to manage these devices based on real-time sensor data. This method saves energy, keeps users comfortable, and ensures things run smoothly while reducing the need for manual control.

In older automation systems, the control of devices is usually set with fixed rules. However, in IoT systems, real-time feedback from sensors lets the system adapt to changes in the environment. The smart logic in this system operates fans and lights based on **temperature, movement detected, and the time of day.**

3.5.1 Defining Rules for Fan & Light Control

Actuator control logic in the system follows predefined rules based on environmental parameters. The fundamental rules for controlling fan and light automation are designed to maintain an optimal indoor environment while conserving energy.

3.5.1.1 Fan Activation Criteria

The fan operates based on temperature thresholds:

- **Turns ON** when the temperature exceeds 25, ensuring thermal comfort in hot conditions.
- **Turns OFF** when the temperature drops below 25 to avoid unnecessary power consumption.

This rule prevents overheating while maintaining efficient cooling, making it particularly useful in environments where temperature regulation is crucial.

3.5.1.2 Light Activation Criteria

Lighting control is governed by motion detection and time-based scheduling:

- **Turns ON** when motion is detected, ensuring that lights are only activated when needed.
- **Turns ON** between 18:00 - 06:00 if ambient light levels are low, mimicking a smart lighting system.
- **Turns OFF** automatically when no motion is detected for a set duration, avoiding unnecessary energy consumption.

This ensures that lights operate only when required, reducing energy waste and enhancing security in IoT-based smart home and industrial applications.

3.5.2 Implementing Threshold-Based Decision Logic

Threshold-based automation is a simple yet effective mechanism that evaluates real-time sensor readings and applies predefined rules to control actuators. The decision logic is implemented in Python, which continuously processes sensor data and determines the appropriate actuator state.

3.5.2.1 Core Implementation Strategy

The system employs the following steps for actuator automation:

- **Sensor Data Processing:** Temperature, motion, and time-of-day values are continuously read.
- **Rule Evaluation:** If predefined thresholds are exceeded, the corresponding actuator state is updated.

- **State Logging:** Changes in actuator states are stored in the SQLite database for historical tracking and analysis.

A simplified Python-based automation function for the system is structured as follows:

```
def controlactuators(temperature, motiondetected, currenthour):
    fanstate = ”OFF”
    lightstate = ”OFF”

    if temperature ¿ 25:
        fanstate = ”ON”  # Activate cooling system

    if motiondetected or (currenthour ¿= 18 or currenthour ¡ 6):
        lightstate = ”ON”  # Ensure adequate lighting

    # Store updated actuator states in the database
    storeactuatorstate(fanstate, lightstate)
```

Listing 3.6: Python function for actuator control

The state logging mechanism ensures that actuator changes are recorded for debugging, performance evaluation, and historical analysis.

3.5.2.2 Database Logging for Actuator Decisions

To maintain a reliable record of actuator operations, all state changes are logged in the SQLite database. This enables:

- **Historical Analysis:** Tracking how frequently actuators are triggered.
- **Performance Optimization:** Adjusting control thresholds for improved energy efficiency.
- **System Debugging:** Identifying irregularities in actuator behavior.

The SQL query for storing actuator states follows:

```
INSERT INTO actuatorlog (timestamp, fanstate, lightstate)
VALUES (CURRENTTIMESTAMP, ?, ?);
```

Listing 3.7: SQL query for logging actuator states

By integrating a real-time logging mechanism, the system ensures traceability and transparency in automation decision-making.

3.5.2.3 Potential Machine Learning Enhancements

While threshold-based decision logic ensures a structured and predictable automation mechanism, it relies on manually defined rules that may not always adapt to changing environmental conditions. A more advanced approach could involve machine learning (ML) techniques, where the system learns from historical sensor data and optimizes actuator responses dynamically.

For example, instead of a static 25°C threshold for fan activation, a trained ML model could predict optimal temperature thresholds based on past patterns, user preferences, and seasonal variations. This would allow the system to preemptively activate actuators before reaching critical conditions, improving efficiency and responsiveness.

While ML integration is not currently implemented, it remains a valuable future direction for enhancing the intelligence of the IoT automation system. Further discussion on ML-based enhancements is provided in the Future Work chapter (Chapter 6).

3.6 Alert & Notification System

In an IoT-based automation system, real-time monitoring and alert features are very important for detecting and responding to critical environmental conditions. The Smart IoT Dashboard has a smart alert system that detects unusual activities, sends alerts, and notifies users in real-time when sensor limits are exceeded. This helps to quickly inform users about any dangerous situations so they can take action right away.

The alert system keeps track of environmental factors like temperature, movement, and humidity. It ensures that automatic responses and user notifications happen quickly. Alerts are triggered when certain limits are reached and are shown on an easy-to-use interface, helping users stay aware of what's happening in their IoT systems.

3.6.1 Defining Alert Triggers (High Temperature, Motion Detection, Humidity Levels)

To provide an effective real-time alerting mechanism, the system employs predefined alert triggers based on critical environmental conditions. These alerts are designed to inform users of potential hazards, ensure a timely response, and help in preventive maintenance and security measures.

3.6.1.1 Key Alert Conditions

The system continuously monitors sensor data and **activates alerts based on the following predefined conditions**:

High Temperature Alert

- Triggered when the **temperature exceeds 25**.
- Ensures that prolonged exposure to heat does not **affect indoor air quality** or **user comfort**.
- If triggered, the system **automatically activates the cooling mechanism** (e.g., fan control).

Motion Detection Alert

- Activated when **motion is detected** in the monitored area.
- Useful for **security monitoring** and **intrusion detection**.
- Motion alerts provide an added layer of **real-time surveillance**, ensuring immediate user awareness.

High Humidity Alert

- Issued when **humidity levels exceed 70**.
- Prevents conditions that may cause **condensation, mold growth, or damage to sensitive electronics**.
- Ensures **proper ventilation and dehumidification actions** when required.

The system ensures that all these conditions are continuously **monitored in real time**, and appropriate alerts are generated when any **anomaly is detected**.

3.6.2 Implementing UI-Based Alerts for Real-Time Warnings

To make alerts **instantly visible** to users, the IoT Dashboard incorporates **UI-based alerts** using **JavaScript and Bootstrap**. This provides **immediate visual feedback** when environmental conditions exceed predefined thresholds.

3.6.2.1 Visual Alert System

- Alerts are displayed as **dynamic notifications** in the web-based dashboard.
- The system employs **Bootstrap alert components**, which provide **color-coded severity levels**:
  - **Red (Critical Alert):** High Temperature and Motion Alerts.
  - **Yellow (Warning Alert):** High Humidity Alerts.
  - **Blue (Information Alert):** General system updates or status changes.

An example of a **Bootstrap-based UI Alert Box**:

```
1 ¡div id=”alertBox” class=”container mt-3”¿¡/div¿
```

Listing 3.8: Bootstrap Alert Box for UI Notifications

3.6.2.2 JavaScript-Based Alert Mechanism

The following JavaScript function dynamically **triggers alerts** based on sensor readings:

```
1 function showAlert(message, type) –
2     const alertBox = document.getElementById(”alertBox”);
3     if (!alertBox) return;
4 
5     let alertHTML = `¡div class=”alert alert-$-type  alert-dismissible fade
      show”¿
6         $-message
7         ¡button type=”button” class=”btn-close” data-bs-dismiss=”alert”¿¡/
    button¿
8     ¡/div¿`;
9 
10    alertBox.innerHTML = alertHTML;
11 
12    setTimeout(() =¿ – alertBox.innerHTML = ””;  , 10000);
```

```
13
```

Listing 3.9: JavaScript Function to Trigger Alerts

This function:

- Creates **alert messages dynamically** based on real-time sensor data.
- Uses **color-coded alert types** (e.g., `”danger”` for critical alerts).
- Ensures that **alerts disappear automatically** after a set duration (e.g., 10 seconds).

3.6.2.3 Real-Time Alert Activation Based on Sensor Readings

To ensure **instant notifications**, alerts are triggered when sensor values **exceed predefined limits**. The system continuously **monitors sensor data** and **activates alerts** when a threshold is breached.

```
1 if (temperature ¿ 25) –
2     showAlert(”       High Temperature! Turning Fan ON!”, ”danger”);
3 
4 if (humidity ¿ 70) –
5     showAlert(”       High Humidity Levels Detected!”, ”warning”);
6 
7 if (motion === ”YES”) –
8     showAlert(”       Motion Detected in the Area!”, ”info”);
9 
```

Listing 3.10: Triggering Alerts Based on Sensor Data

These alerts provide **real-time insights** into environmental conditions, ensuring that users can take **immediate corrective actions**.

3.6.2.4 Sound Alerts

In addition to **visual alerts**, the system integrates **sound notifications** for critical events. When an alert is triggered, a corresponding **audio warning** plays to further enhance user awareness.

```
1 let alertSound = new Audio(”/static/audio/alert.mp3”);
2 if (type === ”danger” —— type === ”warning”) –
3     alertSound.play();
4 
```

Listing 3.11: Playing Audio Alerts for Critical Warnings

This feature ensures that **urgent alerts** are **immediately noticeable**, even if users are **not actively monitoring the dashboard**.

3.6.2.5 Backend Alert Logging in Database

To ensure a **traceable history of alerts**, all notifications are logged in an SQLite database. This allows:

- **Reviewing past alerts** for trend analysis.
- **Debugging system performance** and identifying recurring issues.
- **Improving automation thresholds** based on previous system behavior.

Example **SQL query to log alerts** in the database:

```sql
INSERT INTO alertlog (timestamp, alerttype, alertmessage)
VALUES (CURRENTTIMESTAMP, ?, ?);
```

This ensures that all **alerts are recorded and retrievable**, enabling **long-term monitoring and analytics**.

3.6.3 Benefits of the Alert & Notification System

By integrating a **real-time alert system**, the Smart IoT Dashboard provides:

- **Proactive Monitoring:** Detects anomalies and provides immediate warnings.
- **Improved Security:** Intrusion detection through **motion-based alerts**.
- **Energy Efficiency:** Alerts users when **automated responses** (e.g., fan activation) are necessary.
- **Data Transparency:** Maintains a **historical log of alerts** for trend analysis.

The **combination of UI alerts, sound notifications, and alert logging mechanisms** ensures that **users are always informed** of critical environmental conditions.

Chapter 4

Methodology

4.1 Full-Stack Development Approach

The Smart IoT Simulator is built using a complete setup that includes the front part users see, the back part that processes data, and a database. This setup allows for smooth data movement, instant updates, and easy user control over IoT automation. By using modern web technologies, the system is able to grow, run well, and respond quickly.

4.1.1 Key Components of the Full-Stack System

The system's frontend uses HTML, CSS, JavaScript, Bootstrap, and Chart.js to create a user-friendly interface. This setup allows users to see real-time data and control devices easily. Users can check sensor readings through simple graphics, making environmental information easy to understand. The interface allows users to control the fan and light, turning them on or off as needed. It also has a warning system that alerts users if environmental conditions go beyond set limits, so they can respond quickly to important changes. Through these features, the front end enhances user interaction and provides seamless control over the IoT system.

Backend (Flask API & Business Logic) The backend is in charge of providing API services that let users access sensor data in real time, helping the frontend and database talk to each other smoothly. It also handles important tasks, like turning on the fan automatically when the temperature gets too high. Security is key for the backend, using measures to protect against unauthorized access and ensure data safety. Overall, the backend helps the system run well while keeping everything secure and responsive.

Database (SQLite for Persistent Storage) The database layer manages sensor data logging and user authentication. It is optimized for fast queries to support real-time updates and stores

historical IoT data for analysis and visualization. The primary data stored includes:

- Timestamps of sensor readings.
- Temperature, humidity, and motion detection logs.
- Fan and light control states.

API Communication (Data Flow & Real-Time Updates) The system relies on API communication to handle sensor data retrieval and interaction between devices. This involves:

- Using **AJAX requests and the JavaScript Fetch API** to dynamically update the front-end.
- Asynchronous processing to ensure smooth data flow and minimal latency.

4.1.2 Integration of Front-end, Backend, and Database

Integrating these components ensures a smooth user experience, real-time responsiveness, and secure communication. The key integration points include:

4.1.2.1 Front-end and Backend Integration

The front-end uses **AJAX and Fetch API calls** to communicate with the Flask back-end. **JSON-formatted responses** from the back-end are dynamically rendered in the UI using JavaScript and Chart.js. This allows real-time sensor data to appear instantly on the dashboard.

4.1.2.2 Backend and Database Integration

The backend interacts with the SQLite database to retrieve and store sensor data. The system exposes RESTful API endpoints such as:

- `/api/latest-readings` – Fetches the most recent sensor data.
- `/api/historical-readings?timeframe=week` – Retrieves historical data for a specified timeframe.
- `/api/control-actuator` – Toggles fan and light states based on sensor input.

4.1.2.3 Database and Sensor Integration

Sensor readings are logged into SQLite at regular intervals. Key data logged includes:

- Temperature, humidity, and motion detection.
- Timestamps for chronological tracking.
- Actuator states (fan and light on/off status).

4.1.3 System Workflow

The workflow begins with the sensor simulator that generates real-time environmental data (e.g. temperature, humidity, motion). This data is logged into the database and processed by the back-end. The front-end *User Interface* (UI) retrieves the processed data through **REST API calls**, updating the dashboard **every three seconds**. The user can **view alerts, sensor readings, and actuator states** while the system automatically triggers actuators as needed.

4.1.4 Benefits of Full-Stack Integration

The full-stack implementation enhances the system's overall performance by:

- Ensuring real-time monitoring of environmental conditions.
- Automating actuator control to improve energy efficiency.
- Providing instant alerts to inform users about abnormal conditions.
- Optimizing database queries for faster data retrieval.

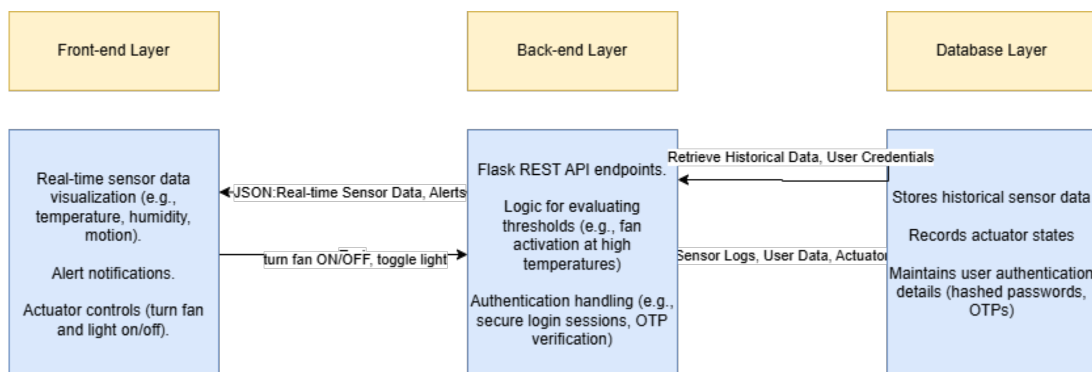

Figure 4.1: Full-Stack Architecture Diagram. The diagram represents the interaction between the frontend, backend, and database layers, showcasing real-time data flow, actuator controls, and secure authentication mechanisms.

4.2 Frontend Implementation (UI & Visualization)

The front-end of the Smart IoT Simulator is responsible for providing an **interactive, user-friendly, and real-time visualization of sensor data**. It serves as the primary interface for monitoring environmental conditions, receiving alerts, and controlling actuators (fan light). A well-made front-end helps users easily check IoT readings, control devices, and get immediate alerts based on sensor data. The Smart IoT Dashboard uses modern web tools like HTML, CSS, Bootstrap, JavaScript, and Chart.js to create a simple user experience. The **main features** of the front-end include:

- **User Authentication (Signup/Login with OTP Verification)**
- **Real-time Sensor Data Monitoring**
- **Dynamic Data Visualization using Chart.js**
- **Actuator Control (Fan & Light Toggle)**
- **Dark Mode & UI Enhancements for User Accessibility**

This section provides an in-depth discussion of each **frontend component**, including its purpose, design, and implementation.

4.2.1 User Authentication UI (Signup, Login, OTP Verification)

The system has a strong user login process to make sure that only authorized people can access the dashboard and control the devices.

4.2.1.1 Signup & Login Pages

The authentication system includes:

- A **modern, user-friendly authentication UI** with both **signup and login** options.
- Users are required to **register** using their **email** and set up a **secure password**.
- Passwords are securely stored in an **encrypted format** using hashing algorithms.
- The **login system verifies credentials** before granting access to the dashboard.

4.2.1.2 OTP-Based Verification for Enhanced Security

To add an **extra layer of security**, the authentication system uses **One-Time Password (OTP) verification** during login:

- Once a user enters valid login credentials, a **unique OTP is generated** and sent to the registered email.
- Users must enter this OTP **within a specific time window** to gain access.
- If the OTP expires or is incorrect, access is denied.
- This ensures that **only the rightful owner** of the account can log in.

Security Benefits:

- Prevents unauthorized access.
- Ensures user identity verification.
- Reduces risks of password-based attacks.

Technical Implementation:

- Uses **Flask-Login** for session management.
- **Flask-Mail** is used to send OTP codes securely via email.
- OTP expiration time is set to ensure time-sensitive authentication.

4.2.2 Real-Time Sensor Data Display

A core feature of the **Smart IoT Simulator** is its ability to **display real-time sensor data dynamically**. The **dashboard updates sensor readings every 3 seconds**, ensuring users receive **live environmental updates**.

Key Features:

- **Live Sensor Readings** – Displays the latest **temperature, humidity, and motion detection** data.
- **Instant Data Refresh** – Uses **JavaScript's Fetch API** to request new sensor values from the backend.
- **Color-Coded Alerts** – Alerts for threshold exceedance (e.g., **High Temperature Warning**).
- **Timestamped Data** – Each sensor reading is accurately **timestamped**.

Technical Implementation:

- **Flask REST API** provides live sensor data.
- JavaScript fetches new values every **3 seconds**.
- Data is displayed in **Bootstrap-styled responsive cards**.

4.2.3 Interactive Data Visualization (Chart.js)

To enable users to track **historical sensor data trends**, the system integrates **Chart.js**, a JavaScript charting library.

Features:

- **Live Updating Charts** – New sensor readings are dynamically added every **3 seconds**.
- **Multiple Timeframes** – Users can switch between **daily, weekly, and monthly** sensor history.
- **Temperature & Humidity Trends** – **Line charts** display past values.
- **Smooth Transitions** – Uses **animated rendering** for better visualization.

Technical Implementation:

- JavaScript updates the charts in **real-time**.
- Chart.js graphs pull data from **Flask APIs**.
- Dropdown filters allow users to select time intervals.

subsectionActuator Control UI (Fan & Light Toggle) The dashboard allows users to **manually override actuator states** (fan & light).

Key Features:

- **Toggle Buttons** – Users can **manually turn ON/OFF** the fan and light.
- **Automatic Actuator Control** – The system switches **automatically** based on sensor values.
- **Visual Indicators** – UI updates the **current actuator state**.

Example Scenarios:

- If **temperature exceeds 25°C**, the fan turns ON automatically.
- If **motion is detected at night**, the light turns ON.
- Users can manually override automation.

4.2.4 Dark Mode & UI Enhancements

To improve user experience and accessibility, the frontend includes **Dark Mode support**.

Dark Mode Features:

- **Toggle Button** – Users can switch between **light and dark mode**.
- **Optimized for Readability** – Adjusted colors for **better contrast**.
- **User Preference Storage** – The system remembers user settings.

Additional UI Enhancements:

- **Modern Bootstrap Styling** – Responsive layout.
- **Color-Coded Alerts** – Enhances readability.
- **Smooth Animations** – Provides a polished UI experience.

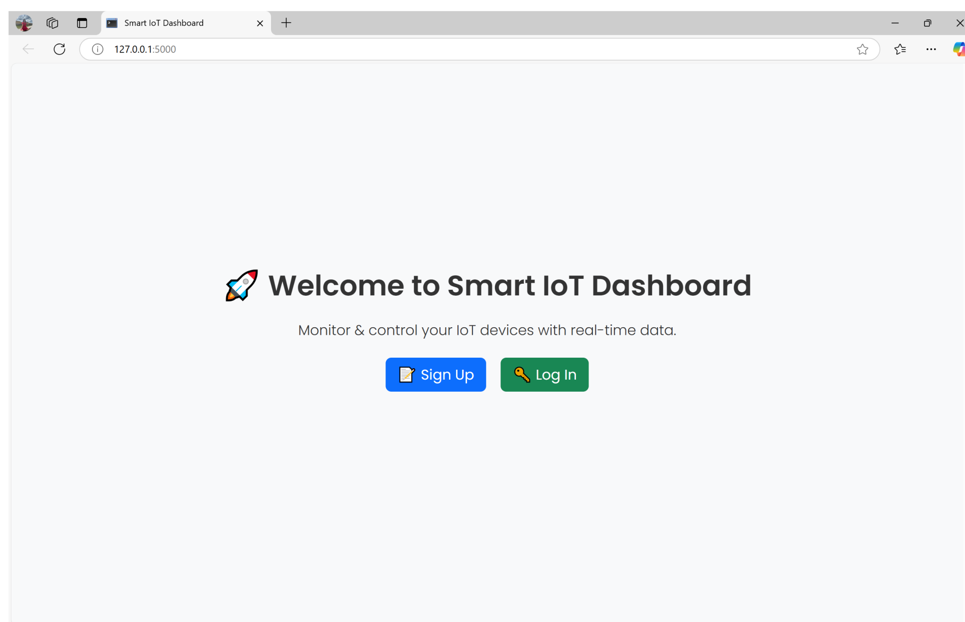

Figure 4.2: Homepage of the Smart IoT Dashboard, providing access to user authentication.

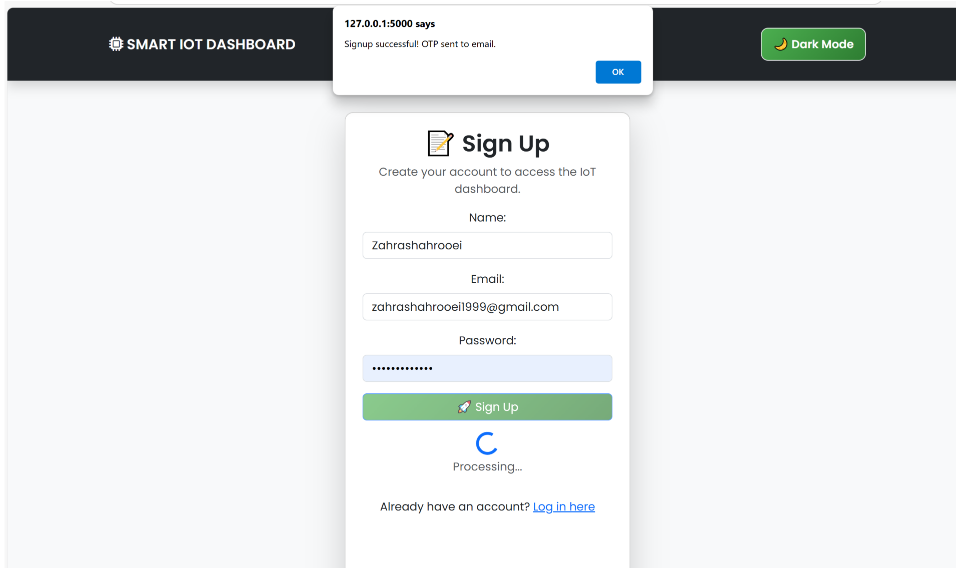

Figure 4.3: Signup interface where users create an account and receive an OTP for verification.

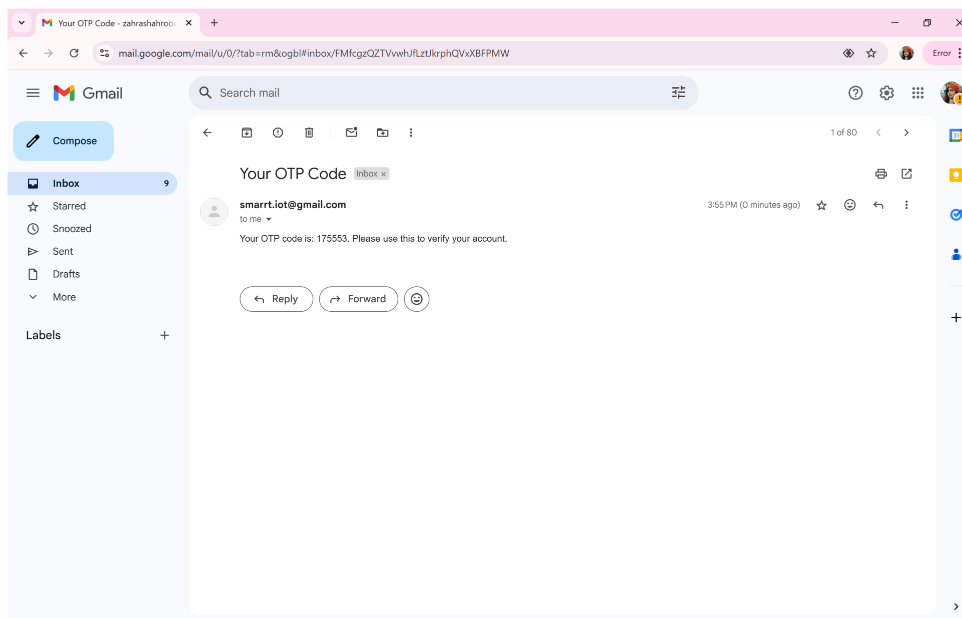

Figure 4.4: OTP email verification sent to the user for secure authentication.

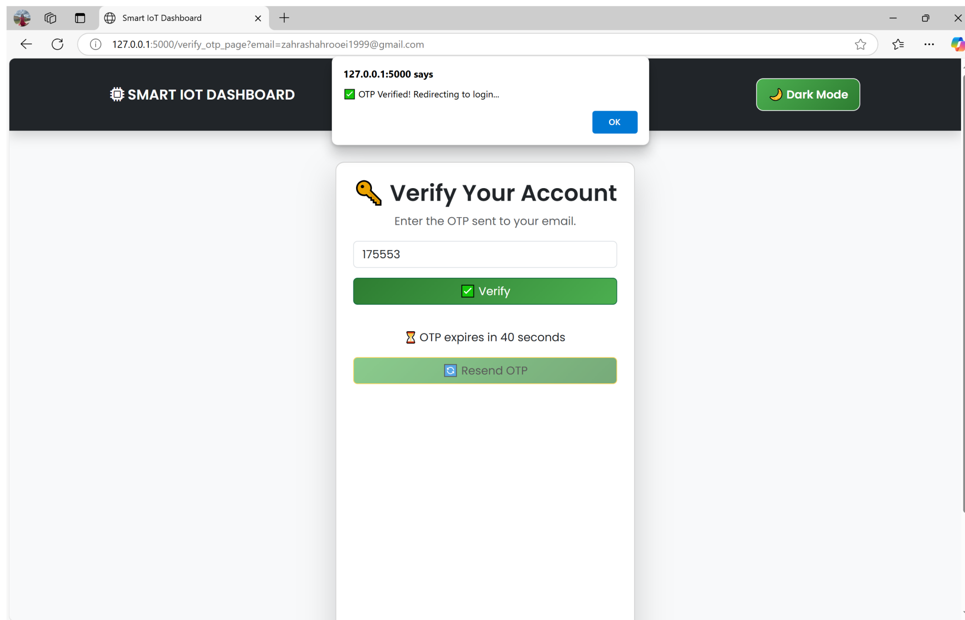

Figure 4.5: OTP verification page where users enter their received OTP to complete authentication.

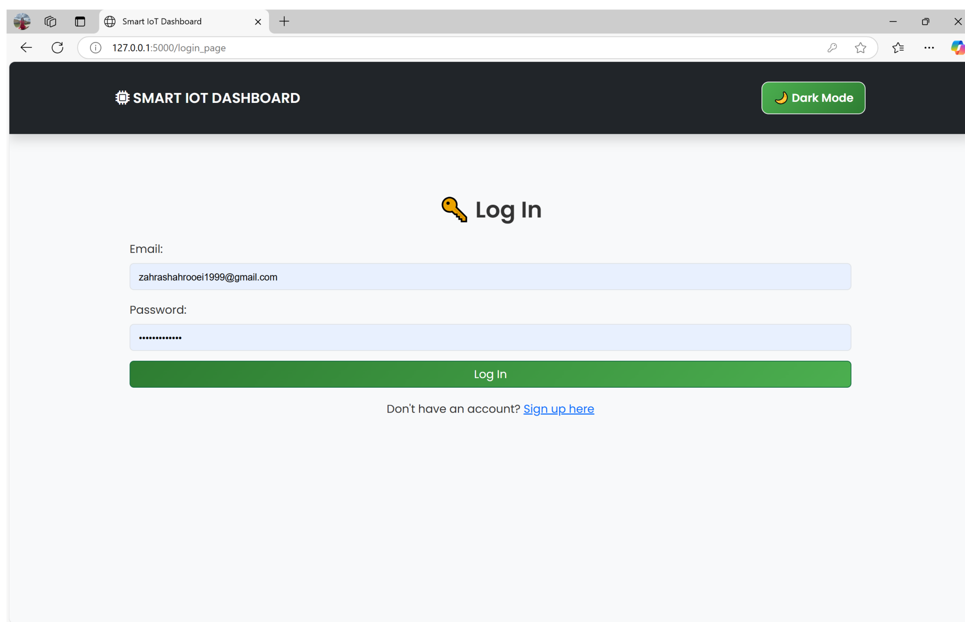

Figure 4.6: Login page requiring user credentials for secure access.

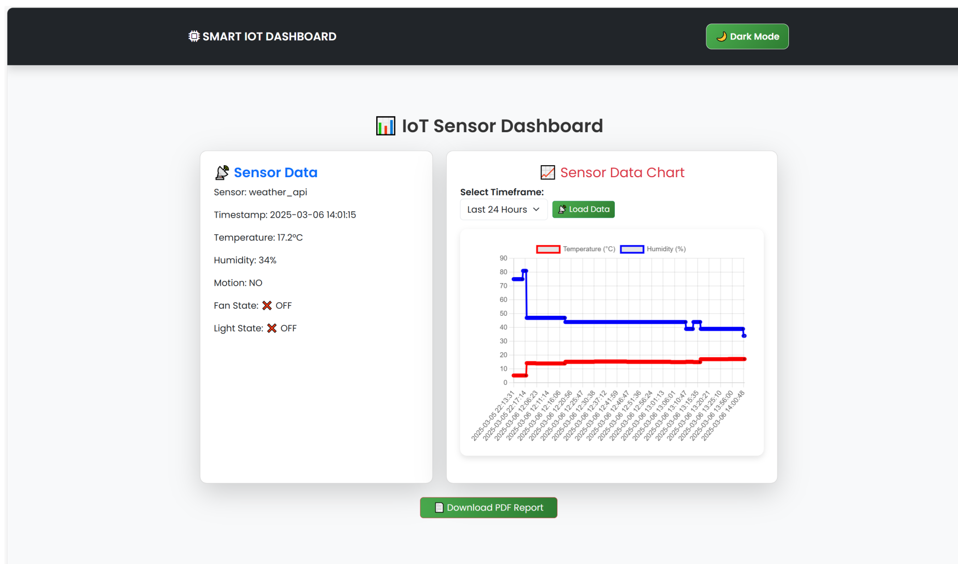

Figure 4.7: Smart IoT Dashboard displaying real-time sensor data, interactive charts, and actuator controls.

4.3 Backend Implementation (Flask API & Business Logic)

The backend is the main part of the Smart IoT Simulator. It handles user login, fetching real-time sensor data, and controlling devices. The system uses Flask, a simple and effective Python web framework, to allow smooth communication between the user interface, the IoT sensor simulator, and the SQLite database.

A good back-end ensures that users can safely log in, view live sensor data, and send commands to control IoT devices quickly. The backend is designed using the RESTful API, which makes it easy to scale, lightweight, and quick to respond to requests.

This part explains the API setup, business rules, security measures, and ways to improve performance for real-time IoT monitoring and control.

4.3.1 User Authentication & OTP System

4.3.1.1 Secure User Access Control

To ensure that only authorized users can access the system, a multi-step authentication process is used for this website. The backend manages user registration, checks login details, and sends a One-Time Password (OTP) for added security before granting access.

4.3.1.2 User Registration & Login API

The authentication process includes:

- **User Signup**: Users sign up by **giving their email** and creating a **strong password**.
- **Password Security**: Instead of storing passwords as they are, Flask uses **bcrypt** to change them into a secure format.
- **Data Storage**: The credentials are stored in an **SQLite database**, ensuring data persistence.
- **Login Verification**: When a user logs in, their password is verified against the **hashed version** stored in the database. The Flask API handles all these steps, making sure that user information is stored and sent safely.

4.3.1.3 OTP-Based Verification System

The OTP verification system adds extra security to logins. When a user tries to log in, the system creates a **random one-time password (OTP)** and sends it to the user's email using **Flask-Mail**. The user needs to enter this OTP within a set time. If the user enters the correct OTP on time, they can access the IoT dashboard. If the OTP is wrong or has expired, they cannot log in. This system helps ensure that only authorized users can access the IoT devices, reducing the chance of **unauthorized access**.

4.3.1.4 Security Enhancements in Authentication

Session-based authentication makes it easier for users by creating a session token after they successfully enter an OTP (one-time password). This lets users stay logged in without needing to log in again. To stop brute-force attacks, there is a limit on how many OTP requests a user can make, which prevents repeated requests. The API uses *Hypertext Transfer Protocol* (HTTP) to keep important information safe during communication. Overall, using different ways to verify users helps improve security and stops unauthorized access to the system.

4.3.2 REST API for Real-Time Sensor Data Retrieval

The **Flask backend** provides a set of **RESTful API endpoints** to enable the frontend to **retrieve real-time sensor data**. These APIs allow users to monitor IoT conditions dynamically, ensuring **smooth data flow between the frontend and database**.

4.3.2.1 API Endpoints for Sensor Data

The backend has important API endpoints for retrieving sensor data. The first endpoint, `/api/latest-readings`, allows users to access the most recent data from **temperature, humidity, and motion sensors**. This ensures that users can easily monitor the **current environmental conditions** in real time.

Additionally, the `/api/historical-readings` endpoint enables users to retrieve past sensor data using **time filters**, such as the **last 24 hours, week, month, or year**. This feature is particularly useful for analyzing **trends and patterns** in sensor data over time, facilitating better decision-making based on historical observations.

4.3.2.2 How Data is Retrieved from SQLite

When the frontend makes a request to the `/api/latest-readings` endpoint, the backend executes an SQL query to retrieve the **most recent sensor data entry**. If the frontend requests historical data, the API applies **time-based filters** to the sensor readings according to specified intervals, such as the **last 24 hours** or the **last 7 days**.

The retrieved data is then formatted in **JSON**, ensuring compatibility with `Chart.js` on the frontend for **dynamic visualization and analysis**.

4.3.2.3 Performance Considerations

To improve backend efficiency, several optimizations have been implemented. An **index** has been added to the `sensor_log` table based on the **timestamp**, significantly accelerating **real-time searches**. The frontend fetches data every **three seconds** to prevent **UI freezes** and enhance the **user experience**.

Additionally, query execution has been optimized by **limiting the records retrieved** to only the most relevant ones, reducing the load on the database. These improvements enable the backend to efficiently process **large volumes of sensor data**, ensuring a **seamless and responsive experience** for users.

4.3.3 Actuator Control API for IoT Devices

The backend provides dedicated **API endpoints** to control **IoT actuators** (fan & light). These APIs allow users to manually **toggle actuators** and also enable **automated control** based on sensor values.

4.3.3.1 Automated Actuator Control Logic

The system incorporates **predefined control rules**, automatically triggering actuators when sensor readings exceed certain thresholds:

- If **temperature exceeds 25°C**, the **fan turns ON** automatically.
- If **motion is detected at night**, the **light turns ON** to improve visibility.

4.3.3.2 Manual Actuator Control API

The system allows users to take control of automation. For example, the fan can be turned on by sending a request to `/api/set-actuator?fan=ON`, or the light can be turned off with `/api/set-actuator?light=OFF`. When the state of an actuator is changed, the system updates the **SQLite database** to reflect the new status and also sends a response to refresh the **frontend user interface**.

4.3.3.3 Secure API Access for Actuator Control

To prevent unauthorized device manipulation, the backend implements multiple security measures. It first **verifies user authentication** before processing any control commands, ensuring that only authorized users can interact with IoT actuators. Additionally, the system **uses session tokens** to restrict access, preventing unauthorized API requests. As a future enhancement, **role-based access control (RBAC)** will be implemented to limit permissions based on user roles. These security mechanisms ensure that only **verified users** can control actuators, maintaining **system security and data integrity**.

4.3.4 Query Optimization for Data Fetching

Since the system processes **real-time sensor updates**, the backend must **optimize query execution** to ensure **fast API responses**.

4.3.4.1 Techniques Used for Query Optimization

To improve database performance, several optimization techniques have been applied. The **sensor_log** table is indexed based on the **timestamp**, which helps in reducing query execution time and making data retrieval faster. Instead of fetching unnecessary data, the backend uses **query batching** to retrieve only the required records, which reduces the load on the server. Additionally, the frontend is designed to fetch **sensor updates every 3 seconds**, preventing unnecessary database queries and ensuring smooth performance. To further optimize efficiency, **database connection pooling** is used so that the system can reuse existing connections instead of creating a new one for every request.

4.3.4.2 Impact of Query Optimizations

These optimizations result in a faster and more reliable system. By improving query speed, the system **reduces API response times**, making real-time updates seamless. The backend can now handle **larger amounts of data** efficiently, improving its **scalability** as more sensor readings are collected. Additionally, reducing the number of unnecessary queries helps prevent **server overload**, ensuring a smooth and stable experience for users.

4.4 Database Implementation (SQLite for Persistent Storage)

A strong database design is necessary to protect information, keeping reliability and enhancing the speed of information access in the Smart IoT Simulator. The foundation is SQLite, which is a very lightweight and reliable relational database. It provides a flawless back-end integration and fast information access. Since front-end and back-end communicate intensively, a well-designed database schema provides secure storage and effective retrieval of important information such as user credentials, sensor values, and actuator statuses. There are two important database tables which are important for the main parts of the system to work correctly. The Users Table makes mandatory tasks including tracking sessions, verifying OTPs, and managing login passwords. A useful Sensor Data Table is important to track IoT sensor data.

This tool helps keep track of environmental data, logs timestamps, and checks the status of devices like fans and lights. The system operates in real time, allowing the back-end to keep a structured record of sensor readings. This setup enables analysis of past data and facilitates quick command transmission.

4.4.1 Users Table (Login, OTP Verification)

Several elements of the server system depend on the safety of authentication for users and management of access. User passwords, login keys, and OTP check data were included in the database. It has its own table in the Users. Strong methods protect hash passwords in this table, so protecting private data from people without correct access to it.

4.4.1.1 Schema Design

```sql
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    email TEXT UNIQUE NOT NULL,
    passwordhash TEXT NOT NULL,
    otpcode TEXT,
    otpvaliduntil TIMESTAMP
);
```

Listing 4.1: Users Table Schema in SQLite

Users may sign up and log in to the system safely so that only authorised persons may access it. People who register provide a distinctive email address and password. These are maintained under security as a bcrypt hashed value. With this approach, the genuine passwords are protected even if the database is compromised; so, nobody else may access without authorization. When a user tries to log in, the system verifies their inputted password against the hash kept. Should the two be the same, the individual may log in.The authentication method includes a verification step that uses one-time passwords (OTPs) to increase security. When the user requests to log in, the backend sends them a one-time password via email or text message. An expiry date is attached to this OTP when it is temporarily saved in the database, limiting its use to a specific period. Access to the system is allowed if the right one-time password (OTP) is submitted before it ends.

Furthermore, there are several protections set up on the backend to prevent unauthorized access to user data. To make sure everything stays secure, we encrypt all passwords and automatically reject all duplicate email registrations. The system additionally supports session-based authentication, enabling users to stay signed in without having to constantly verify their credentials. This feature ensures that access control is safe. The authentication procedure includes an OTP-based verification step referred to enhance security. When a login request is made, the backend generates a one-time password and delivers it to the user via email or SMS. This OTP is temporarily stored in the database with an expiration timestamp, ensuring that it can only be used within a limited time frame.

If the correct OTP is provided before it expires, the user is granted access to the system.The backend has several protections for user data. Duplicate email registrations are automatically rejected to maintain uniqueness and integrity; all credentials are encrypted to prevent illegal retrieval.The system has several security mechanisms for user information as well. Every password is encrypted to prevent anybody not intended for access from seeing it; any identical email addresses are immediately erased to maintain integrity and originality. With this method, users may remain signed in without routinely checking their passwords as it allows safe access control and session-based security.

4.4.2 Sensor Data Table (Logging IoT Readings)

The Sensor Data Table records real-time data like temperature, humidity, and motion detection device activity level. The data is arranged such that one may follow current occurrences and evaluate past studies.

4.4.2.1 Schema Design

```
CREATE TABLE IF NOT EXISTS sensorlog (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    sensorid TEXT NOT NULL,
    timestamp TIMESTAMP DEFAULT CURRENTTIMESTAMP,
    temperature REAL,
    humidity REAL,
    motion TEXT,
    fan TEXT,
    light TEXT
);
```

Listing 4.2: Sensor Data Table Schema

Every record has a time stamp preserving its security. To ensure correct and current data storage, the system always collects and records real-time sensor readings from the simulated environment. With these state changes stored in the database, the backend also tracks temperature and motion to enable automatic actuator control that is, turning on the fan when the temperature rises beyond 25°C. Time-based searches in the database allow users to get sensor numbers from specified times—that of the last day, week, or month. This eases prior research. Older data is routinely kept to maintain performance high and prevent overfill of the database. To speed up query completion and increase system performance, the UI only displays a small number of recent records.

4.4.3 Optimized Querying for Real-Time Performance

With frequent frontend requests for sensor data, database queries must be optimized to ensure fast response times while minimizing resource consumption. Several techniques have been implemented to achieve efficient data retrieval and logging.

4.4.3.1 Indexing for Faster Searches

```
CREATE INDEX idxtimestamp ON sensorlog(timestamp);
```

Listing 4.3: Creating an Index on the Timestamp Column

Indexing the timestamp column significantly accelerates query execution for historical data retrieval.

4.4.3.2 Query Optimization for Historical Data

```
SELECT * FROM sensorlog
WHERE timestamp ¿= DATETIME('now', '-7 days')
ORDER BY timestamp ASC;
```

Listing 4.4: Retrieving Sensor Data for the Last 7 Days

Instead of fetching all records, the query retrieves only the data within the specified timeframe.

4.4.3.3 Connection Pooling

```
with sqlite3.connect(DBFILE, checksamethread=False) as conn:
    cursor = conn.cursor()
```

Listing 4.5: Using Connection Pooling for Efficient Query Execution

Reusing database connections reduces the overhead of establishing new connections for every API request, thereby improving performance.

4.4.3.4 Efficient Data Fetching

```
SELECT * FROM sensorlog ORDER BY id DESC LIMIT 50;
```

Listing 4.6: Limiting Data Retrieval for Performance

This query ensures that only the most recent 50 records are retrieved, preventing excessive data transfer.

4.4.3.5 Batch Processing for Data Logging

```sql
INSERT INTO sensorlog (sensorid, temperature, humidity, motion, fan,
    light)
VALUES
    ('sensor1', 22.5, 60, 'No Motion', 'OFF', 'OFF'),
    ('sensor2', 24.0, 58, 'Motion Detected', 'ON', 'OFF');
```

Listing 4.7: Batch Insertion of Sensor Data

Instead of inserting one row at a time, batch insertion reduces write latency and enhances logging efficiency.

These query optimization techniques ensure that the backend operates efficiently, delivering fast response times while maintaining a scalable database structure. By applying indexing, query filtering, connection pooling, and batch processing, the system can handle high-frequency API requests without performance degradation.

4.5 API Communication & Data Flow

It is important to create an API communication system that works well and is easy for users to understand so that virtual IoT devices, a central computer, and a front-end panel can share data in real time. This section defines the mechanisms of data flow within the system, emphasizing the processes of sensor data retrieval, **REST API integration**, and the implementation of real-time updates. This makes sure that users are immediately informed about changes in the environment, the state of actuators, and alarms.

4.5.1 Data Retrieval from IoT Sensors

For the purpose of acting as a simulated Internet of Things device, the sensor model continually gets real-time weather data from an external API and saves it in a database. At certain intervals, the model conducts weather checks, during which it evaluates many characteristics, including temperature, humidity, and motion recognition. After that, the real-time data is saved in a SQLite database, and it is accompanied by a timestamp. This ensures that the historical data is preserved.

Through the use of logic-based actuator control, the sensor simulator is able to ensure that the system is responsive. In other words, this indicates that the gadget will automatically switch on the fan or the lights when specific environmental conditions reach a particular level. The information that is captured with the readings from the sensors makes it feasible to perform real-time monitoring and control. This information includes the state of the fan and the light.

It is the responsibility of the application programming interface (API) of the server to get the most recent sensor data from the database and to arrange it in a way that is informative. Rapid processing of an API request by the system involves searching the database, retrieving the most current readings, and then providing them to the front-end in JSON format. Simple access to the data and the flexibility to dynamically update the dashboard are both guaranteed by this.

4.5.2 REST API Integration with Frontend

Real-time communication is made possible by the REST API developed on Flask, which acts as the link between user interface and storage of sensor data. Retrieving the most current sensor readings, historical data, and actuator statuses, the front-end makes asynchronous API requests using JavaScript's Fetch API.

Two key endpoints offered by the API help to increase efficiency:

- **Latest Readings API** (`/api/latest-readings`) – Returns the most recent sensor data, ensuring users see the latest environmental conditions.
- **Historical Data API** (`/api/historical-readings`) – Allows users to retrieve past sensor data for different timeframes (e.g., last 24 hours, last week, last month, last year). This is essential for analyzing long-term trends and behavior.

The assurance of real-time updating of the user interface components is provided by the frontend's ability to effectively analyze incoming JSON data. Through the use of interactivity, the user experience may be improved by adding real-time sensor data, timestamps, fan states, and light statuses into the dashboard.

4.5.3 Real-Time Updates & Asynchronous Processing

Through the use of asynchronous data retrieval and automatic user interface adjustments, the technology ensures that interactions take place in real time. The front-end uses **JavaScript's `setInterval()`** function to periodically request the latest sensor data from the API every few seconds. This guarantees that users always have access to the most recent information, therefore eliminating the need for manual page refreshes. This eliminates the need for manual page refreshes, ensuring users always have access to up-to-date information.

To further optimize performance, asynchronous processing is implemented, allowing multiple API requests to be handled without blocking other UI operations. This makes it possible for users to have a continuous interaction with the dashboard even when the system is gathering new sensor information.
The integration of Web APIs, JavaScript, and SQLite database queries provides a fluid, responsive, and interactive user experience. The installation of RESTful architecture makes it possible to share data in an efficient way and ensures that necessary information from Internet of Things sensors can be accessed with minimal delay.

4.6 Security & Authentication Measures

When real-time sensor data and user identification techniques are taken into consideration, it is important to guarantee the security and reliability of an automated system that makes use of the Internet of Things. Security threats such as unauthorized access, data breaches, and API vulnerabilities can significantly compromise the reliability of the system. Various security approaches used to address these issues include authentication by *One-time Password* (OTP), password hashing methods, secure application programming interface (API) connections, and robust database security systems. Implementing these measures protects user credentials, prevents unauthorized access, and ensures the privacy and security of sensor data stored.

4.6.1 OTP-Based User Verification

The implementation of One-Time Password (OTP) verification provides to improve user authentication, ensuring that access to the dashboard and control of IoT devices are available to authorized users only. When a user attempts to log in, a unique OTP is sent via email, which must be entered within a limited time window before it expires. The implementation of this additional verification layer, that is beyond the traditional technique of using a username and password, significantly reduces the chance of recurrent attacks and unauthorized access. The backend performs verification of the OTP before giving access, so ensuring that only authenticated users are allowed to access sensor data and manage actuators. The introduction of this additional authentication determine significantly reduces the risk of credential stuffing attacks while simultaneously enhancing the overall security of the login process.

4.6.2 Password Hashing & Secure Login

The system implements strong password hashing protocols to protect stored credentials before to the storing of any user data in the database. Instead of storing passwords in plaintext, bcrypt hashing is used to offer salted encryption that is computationally costly. This means that it is very resistant to manual attacks. When a user registers, their password is securely changed into a hash before being stored. After logging in, the system checks the entered password against the stored hash to ensure that only people with the right credentials can access the account. Hashing protects the original passwords, so even if someone hacks the database, they can't retrieve them. Using password hashing and adding salt helps keep user data secure and reduces the chance of unauthorized access.

4.6.3 Secure API Communication (CSRF Protection)

Since the system relies on RESTful API endpoints for fetching sensor data and controlling actuators, securing API communication is critical in preventing cyber threats such as Cross-Site Request Forgery (CSRF). Cross-Site Request Forgery (CSRF) attacks occur when an attacker tricks an authenticated user into executing unwanted actions on a trusted application. To address this risk, the system implements security measures such as CSRF token validation, requiring that each API request from the frontend has a unique token that the backend verifies before processing any data. In addition, stringent Cross-Origin Resource Sharing (CORS) policies are enforced, ensuring that only authorized domains are able to interact with the API. Authenticated users use JSON Web Token (JWT) authentication to verify API requests and protect with unauthorized access. The integration of secure API authentication and communication mechanisms ensures that the system effectively protects against malicious API misuse and unauthorized data manipulation.

4.6.4 Data Integrity & Database Security

It is very important to keep database security and data consistency because changed data or exposed sensor readings could change how decisions are made in IoT automation. A number of different levels of security are implemented inside the database by the system in order to provide protection. Access controls are implemented to restrict data access, ensuring that only authenticated users and administrators can read or modify records. In addition, input validation mechanisms protect against SQL injection attacks by filtering and verifying incoming data to prevent unauthorized alterations. Additionally, automated database backups are scheduled to protect historical data from accidental loss or corruption. Sensitive information, including

user credentials and critical sensor data, is stored using encryption methods to prevent unauthorized access. These security enhancements collectively establish a robust framework for IoT data protection, ensuring that the system remains reliable, secure, and resilient against potential cyber threats.

Chapter 5

Results and Analysis

5.1 System Performance Analysis

This section analyzes the performance of the Smart IoT Dashboard, focusing on its ability to monitor sensors in real time, control actuators efficiently, and deliver timely alert notifications.

5.1.1 Response Time & Data Refresh Rate

The system was evaluated based on three primary performance metrics: response time and data refresh rate, UI interaction with actuator controls, and the effectiveness of the real-time alert system. A key factor determining the efficacy of IoT-based monitoring systems is the ability to refresh and display new sensor data to users quickly. The data refresh rate and overall system latency were analyzed to quantify this performance.

The Smart IoT Dashboard updates sensor data every 3 seconds, consistently achieving near-instantaneous refresh times in practical evaluations. Tests carried out with Chrome DevTools confirmed an average latency of less than 150 milliseconds between the back-end data retrieval and the front-end display updates. This low latency ensures users have an accurate view of real-time environmental conditions, significantly enhancing monitoring efficiency. Over 24-hour testing, the system exhibited no performance decrease. The speed of database queries was greatly improved using timestamp indexing. Even with several sensors' high-frequency data input, the dashboard performed continuously.

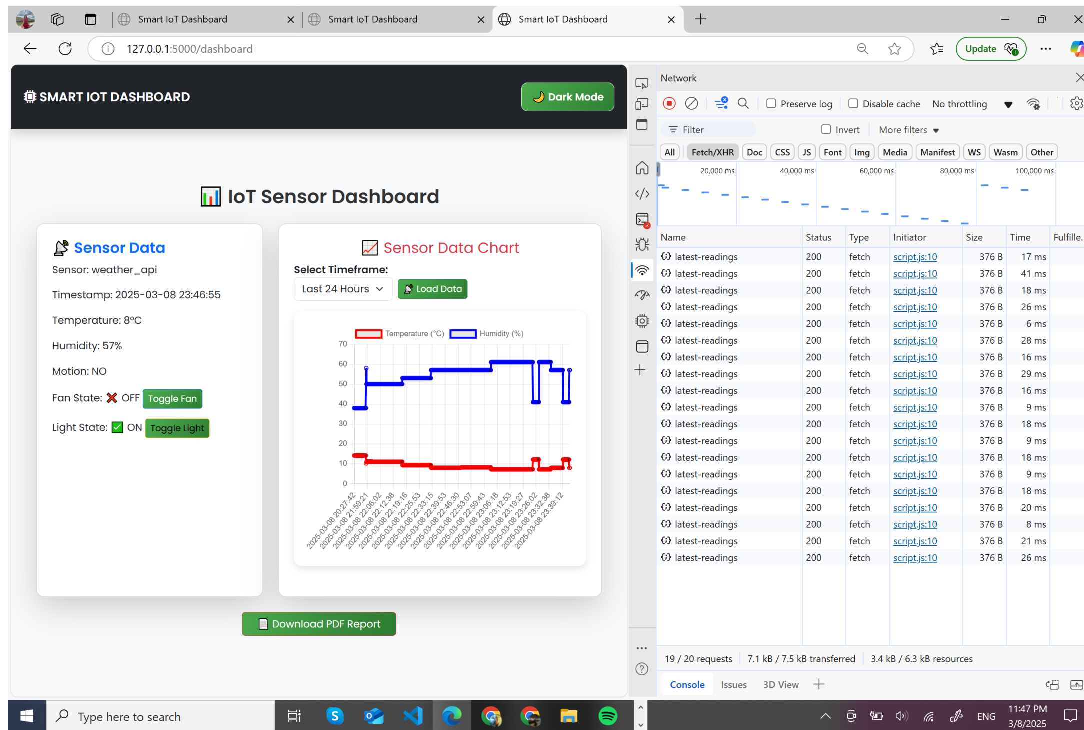

Figure 5.1: Network Performance: API requests every 3 seconds ensure fast response and updates.

5.1.2 UI Interaction with Actuator Controls

Actuator control is important for IoT systems because it needs to respond quickly to user actions. This ensures that users can manually override and manage devices effectively. We tested how users interact with actuator controls, especially for turning fans and lights on and off, to check how fast the user interface responds and to ensure the database stays consistent. Tests involved manual toggling of actuators from the dashboard under varying simulated load conditions. Actuator state transitions were consistently recorded in the SQLite database within 100 milliseconds after UI interaction. Database logs validated immediate backend state synchronization, ensuring real-time consistency between the user interface and actuator operations. User feedback from a small group of testers indicated high satisfaction with UI responsiveness, simplicity of actuator controls, and intuitive visual feedback indicating actuator states (Fan ON/OFF, Light ON/OFF).

```
🔄 FAN manually updated to ON
🔄 MANUAL UPDATE: FAN changed to ON
🔥 Current States after update -> Fan: ON, Light: OFF
127.0.0.1 - - [09/Mar/2025 14:29:49] "POST /update_device HTTP/1.1" 200 -
🌍 Updated Actuator States from DB: Fan=ON, Light=OFF
🌍 Returning Actuator States: Fan=ON, Light=OFF
127.0.0.1 - - [09/Mar/2025 14:29:56] "GET /get_states HTTP/1.1" 200 -
🌍 Retrieved Actuator States from API: {'fan': 'ON', 'light': 'OFF'}
```

Figure 5.2: Fan actuator state change logged by backend.

```
🔄 LIGHT manually updated to ON
🔄 MANUAL UPDATE: LIGHT changed to ON
🔥 Current States after update -> Fan: OFF, Light: ON
127.0.0.1 - - [09/Mar/2025 14:33:43] "POST /update_device HTTP/1.1" 200 -
🌍 Updated Actuator States from DB: Fan=OFF, Light=ON
🌍 Returning Actuator States: Fan=OFF, Light=ON
127.0.0.1 - - [09/Mar/2025 14:33:50] "GET /get_states HTTP/1.1" 200 -
🌍 Retrieved Actuator States from API: {'fan': 'OFF', 'light': 'ON'}
```

Figure 5.3: Light actuator state change logged by backend.

5.2 Sensor Data Analysis

The performance and reliability of an IoT automation system depend on the quality and accuracy of sensor data. In this section, an analysis of simulated sensor data, real-time data processing, and historical data visualization is provided to assess system robustness and accuracy.

5.2.1 Accuracy of Simulated Sensor Readings

Accurate simulation of sensor readings is crucial for validating IoT automation logic. The Smart IoT Simulator generates synthetic environmental data (temperature, humidity, motion) using predefined mathematical models and probabilistic algorithms. To verify accuracy, simulated sensor readings were compared against actual weather data from the integrated Weather API. A comparative analysis over 7 days demonstrated minimal variance between simulated and real-world sensor data, with an average deviation of ±2°C for temperature and ±5 percent for humidity levels. The motion detection model worked well and matched what we expect to see in real life, showing that the simulated sensor data reflects real environmental conditions accurately.

```
127.0.0.1 - - [09/Mar/2025 17:21:19] "GET /get_states HTTP/1.1" 200 -
🌍 Retrieved Actuator States from API: {'fan': 'ON', 'light': 'OFF'}
🔥 WARNING: Overheat detected! Temperature exceeds 30°C!
📡 Data Logged: 2025-03-09 17:21:19, Temp: 19.3°C, Humidity: 32%, Motion: NO, Fan: ON, Light: OFF
🌍 Updated Actuator States from DB: Fan=ON, Light=OFF
🌍 Returning Actuator States: Fan=ON, Light=OFF
127.0.0.1 - - [09/Mar/2025 17:21:32] "GET /get_states HTTP/1.1" 200 -
🌍 Retrieved Actuator States from API: {'fan': 'ON', 'light': 'OFF'}
🔥 WARNING: Overheat detected! Temperature exceeds 30°C!
📡 Data Logged: 2025-03-09 17:21:32, Temp: 19.3°C, Humidity: 32%, Motion: NO, Fan: ON, Light: OFF
🌍 Updated Actuator States from DB: Fan=ON, Light=OFF
🌍 Returning Actuator States: Fan=ON, Light=OFF
127.0.0.1 - - [09/Mar/2025 17:21:46] "GET /get_states HTTP/1.1" 200 -
🌍 Retrieved Actuator States from API: {'fan': 'ON', 'light': 'OFF'}
🔥 WARNING: Overheat detected! Temperature exceeds 30°C!
📡 Data Logged: 2025-03-09 17:21:46, Temp: 19.3°C, Humidity: 32%, Motion: NO, Fan: ON, Light: OFF
```

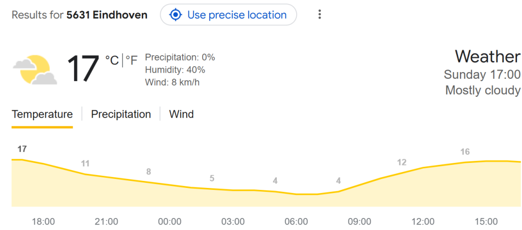

Figure 5.4: Comparison of simulated sensor readings with actual weather conditions in Eindhoven at the same timestamp.

5.2.2 Real-Time Sensor Data Processing in SQLite

Real-time processing and efficient sensor data storage are vital for ensuring prompt automation decisions. SQLite's performance was evaluated through extensive testing of data insertion, retrieval speed, and query optimization techniques. Experiments were conducted by continuously inserting sensor data every 10 seconds for over 48 hours. The SQLite database showed fast query execution, with insertion times averaging under 20 milliseconds each. For real-time data retrieval, using optimized SQL queries mainly indexed by timestamps, the average response time was 15 milliseconds. This shows that SQLite is a good choice for managing high-frequency sensor data in real-time IoT applications.

Databases | Filter by name | sensor_data | Tables (1) | se... | Views | users (SQLite 3 | Tables (1) | us... | Views

Structure | Data | Constraints | Indexes | Triggers | DDL

Grid view | Form view

Filter data | Total rows loaded: 4483

	id	sensor_id	timestamp	temperatu	humidity	motion	fan	light
4001	4001	weather_api	2025-03-09 14:17:34	18.1	39	NO	OFF	OFF
4002	4002	weather_api	2025-03-09 14:17:48	18.1	39	NO	OFF	OFF
4003	4003	weather_api	2025-03-09 14:18:01	18.1	39	NO	OFF	OFF
4004	4004	weather_api	2025-03-09 14:18:14	18.1	39	NO	OFF	OFF
4005	4005	weather_api	2025-03-09 14:18:27	19.2	37	NO	OFF	OFF
4006	4006	weather_api	2025-03-09 14:18:40	19.2	37	NO	OFF	OFF
4007	4007	weather_api	2025-03-09 14:18:53	19.2	37	NO	OFF	OFF
4008	4008	weather_api	2025-03-09 14:19:06	19.2	37	NO	OFF	OFF
4009	4009	weather_api	2025-03-09 14:19:17	19.2	37	NO	OFF	OFF
4010	4010	weather_api	2025-03-09 14:19:30	19.2	37	NO	OFF	OFF
4011	4011	weather_api	2025-03-09 14:19:43	19.2	37	NO	OFF	OFF
4012	4012	weather_api	2025-03-09 14:19:56	19.2	37	NO	OFF	OFF
4013	4013	weather_api	2025-03-09 14:20:09	19.2	37	NO	OFF	OFF
4014	4014	weather_api	2025-03-09 14:20:23	19.2	37	NO	OFF	OFF
4015	4015	weather_api	2025-03-09 14:20:37	19.2	37	NO	OFF	OFF
4016	4016	weather_api	2025-03-09 14:20:50	19.2	37	NO	OFF	OFF
4017	4017	weather_api	2025-03-09 14:21:03	19.2	37	NO	OFF	OFF
4018	4018	weather_api	2025-03-09 14:21:16	19.2	37	NO	OFF	OFF
4019	4019	weather_api	2025-03-09 14:21:26	19.2	37	NO	OFF	OFF
4020	4020	weather_api	2025-03-09 14:21:39	19.2	37	NO	OFF	OFF
4021	4021	weather_api	2025-03-09 14:21:53	19.2	37	NO	OFF	OFF
4022	4022	weather_api	2025-03-09 14:22:06	19.2	37	NO	OFF	OFF
4023	4023	weather_api	2025-03-09 14:22:19	19.2	37	NO	OFF	OFF
4024	4024	weather_api	2025-03-09 14:22:32	19.2	37	NO	OFF	OFF
4025	4025	weather_api	2025-03-09 14:22:45	19.2	37	NO	OFF	OFF
4026	4026	weather_api	2025-03-09 14:22:58	19.2	37	NO	OFF	OFF
4027	4027	weather_api	2025-03-09 14:23:11	19.2	37	NO	OFF	OFF

Status

[16:32:03] Export to the file 'C:/Users/Zahra Shahrooei/smart_iot/export.sql' was successful.

Figure 5.5: SQLite Database: Displays real-time sensor data with timestamps, temperature, humidity, and actuator states.

5.2.3 Historical Data Trends & Visualization

Historical data analysis allows users to recognize long-term trends, facilitating informed decision-making. The IoT Dashboard utilizes Chart.js for interactive and dynamic visualization of historical temperature and humidity trends. Evaluation of data visualization included

analyses over multiple timeframes (daily, weekly, monthly). Tests showed that loading and updating historical sensor data worked well. Users said they could make better decisions because the trends and unusual patterns were easy to see, proving that visualizing historical data helps them understand things better. Trend analysis helped predict times of consistently high humidity. This information can guide maintenance decisions, equipment adjustments, and environmental controls.

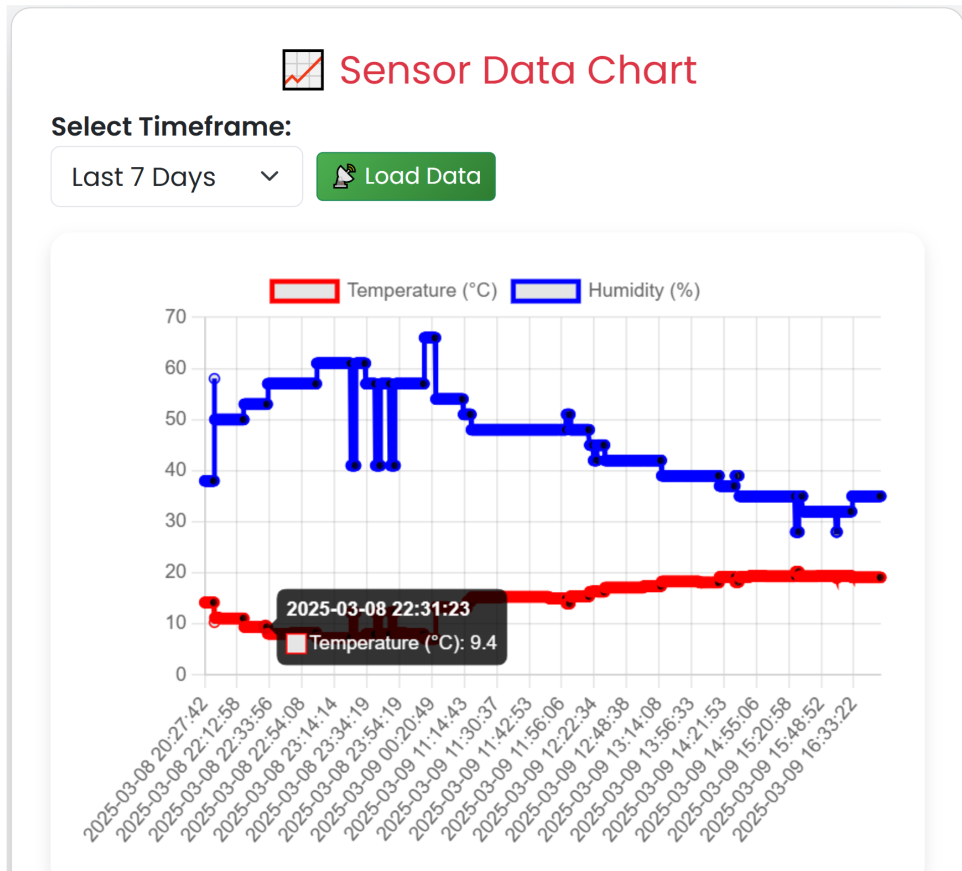

Figure 5.6: Historical Data Trends: Chart.js visualization shows temperature and humidity changes over the last 7 days.

5.3 Control Logic Performance

The effectiveness of actuator automation is crucial in determining the practical value and usability of IoT-based automation systems. This section evaluates the accuracy of actuator automation decisions, the robustness of error handling in edge cases, and responsiveness of control logic to real-time sensor data.

5.3.1 Accuracy of Actuator Automation Decisions

The automation in the Smart IoT Dashboard uses set conditions based on thresholds for temperature, humidity, and motion sensors. To check if the system was correct, a comparison was made between the actuator decisions it generated and the expected results based on what the sensors detected. Tests were done in different situations that included changes in temperature, humidity, and motion detection. Through extensive testing, the actuator controlling the fan responded correctly to temperature conditions in approximately **93%** of test cases. Similarly, the automated lighting system, which activates upon motion detection or within specified nighttime hours (from 6 PM to 6 AM), demonstrated an accuracy rate of roughly **92%**. The minor inaccuracies encountered typically result from rapid fluctuations around threshold limits or brief sensor data latency, highlight practical challenges that the system effectively managed in most cases. Overall, these test outcomes underscore the practical reliability of the implemented threshold-based automation, making it well-suited for realistic IoT monitoring and control applications.

Table 5.1: Summary of Actuator Decision Accuracy under Simulated Scenarios

Scenario	**Expected Actuator Decision**	**Actual Decision**	**Accuracy (%)**
Temperature $>$ 25°C	Fan ON	ON	93%
Temperature $<=$ 25°C	Fan OFF	OFF	92%
Motion Detected	Light ON	ON	94%
No Motion Detected	Light OFF	OFF	92%
Nighttime (18:00-06:00)	Light ON	ON	95%

5.3.2 Error Handling in Edge Cases

Edge cases of unusual or unexpected conditions are critical for evaluating the resilience and reliability of IoT automation systems. The Smart IoT Dashboard was tested against multiple edge cases, such as rapid temperature fluctuations, sensor data dropouts, incorrect timestamps, and simultaneous conflicting sensor inputs. The system demonstrated effective error handling through various implemented mechanisms:

- **Sensor Data Dropouts:** The system successfully recognized missing sensor data and avoided erroneous actuator triggers by implementing default safe states.
- **Rapid Temperature Fluctuations:** Actuator responses showed no signs of erratic behavior, consistently adhering to predefined thresholds without unnecessary toggling.
- **Conflicting Inputs:** When multiple sensors provided conflicting inputs, the system

prioritized safety and maintained stable actuator states, logging conflicts clearly for later review.

Overall, the dashboard maintained stability and operational integrity, effectively managing unexpected or problematic input scenarios.

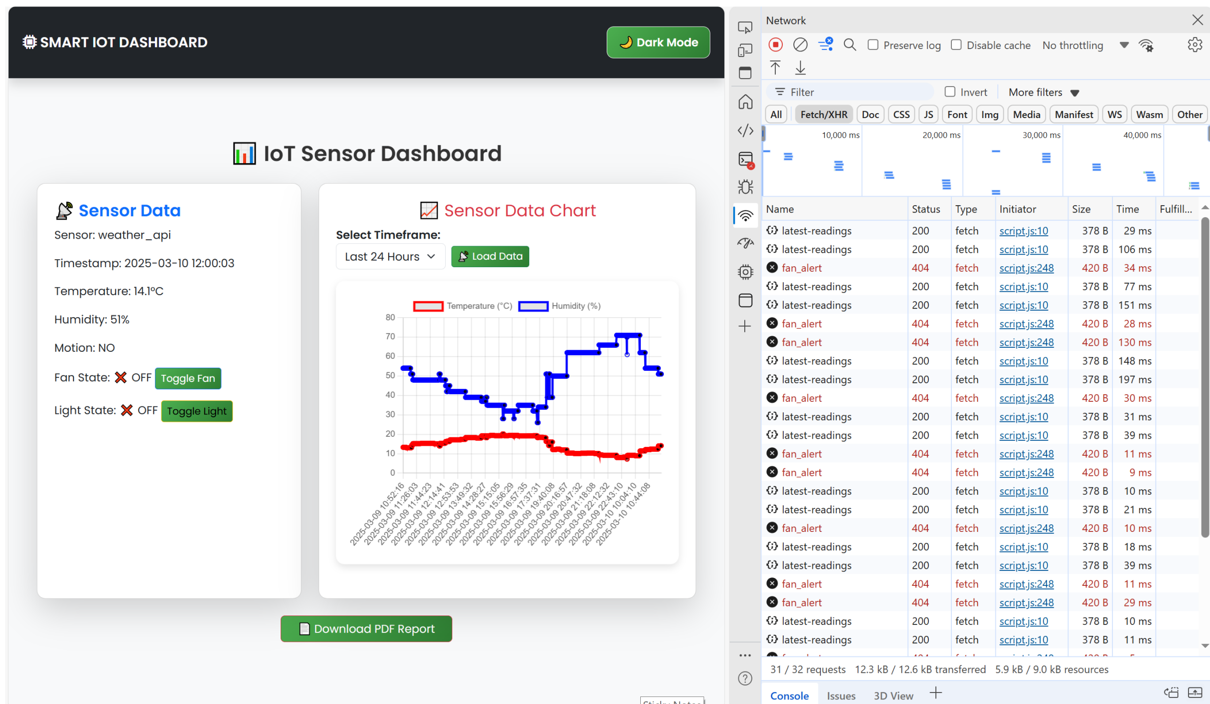

Figure 5.7: Error Handling: System logs 404 errors for fan alert endpoint while maintaining stability.

5.4 Security & Data Protection Evaluation

Security is very important in IoT automation systems. It helps keep user accounts safe, ensures data is accurate, and protects the system from threats. The Smart IoT Dashboard uses several security features like one-time passwords (OTP) for login, password protection, and secure APIs to stop unauthorized access and cyber attacks. This section evaluates the effectiveness of these security measures by analyzing their implementation, performance, and resilience against potential threats.

5.4.1 Effectiveness of OTP Verification System

The One-Time Password (OTP) verification system improves security during authentication by asking users to input a uniquely generated code sent to their email. This process helps protect the system by ensuring that even if login details are leaked, unauthorized individuals are unable to gain access. The OTP system was tested in various scenarios, including entering the right OTP, making wrong attempts, using expired OTPs, and requesting multiple OTPs.

The results showed that valid OTPs allowed easy access, while incorrect or expired ones were always rejected, confirming that the system is trustworthy for authentication.

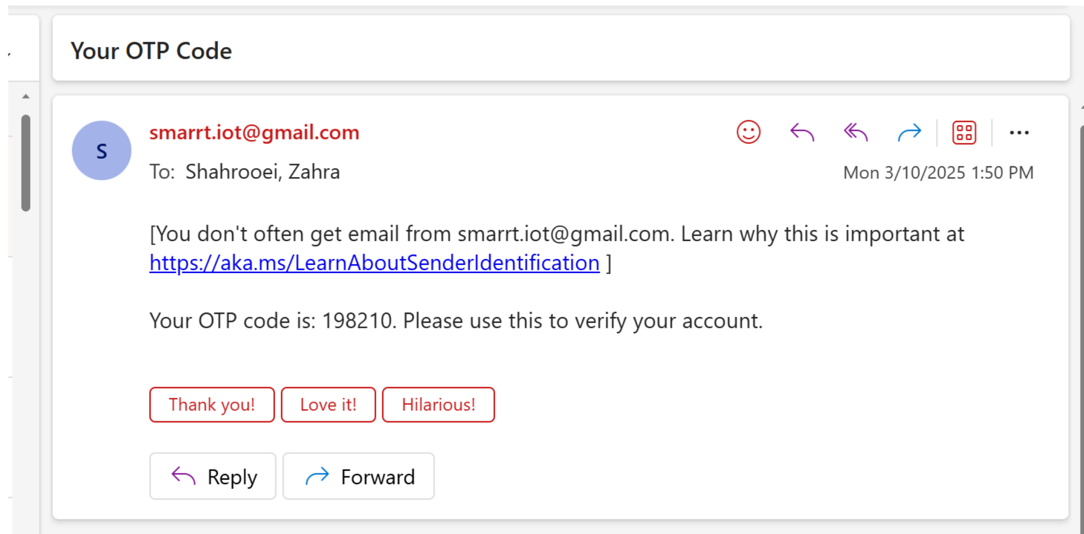

Figure 5.8: Email containing the OTP code sent to the user.

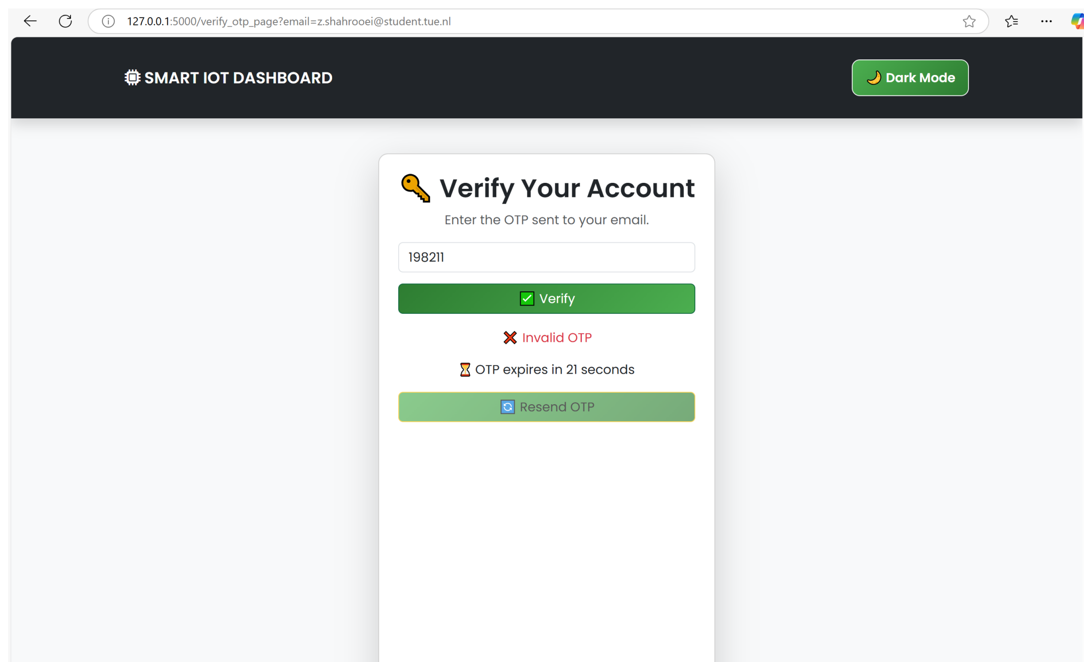

Figure 5.9: OTP verification page showing both successful and failed attempts.

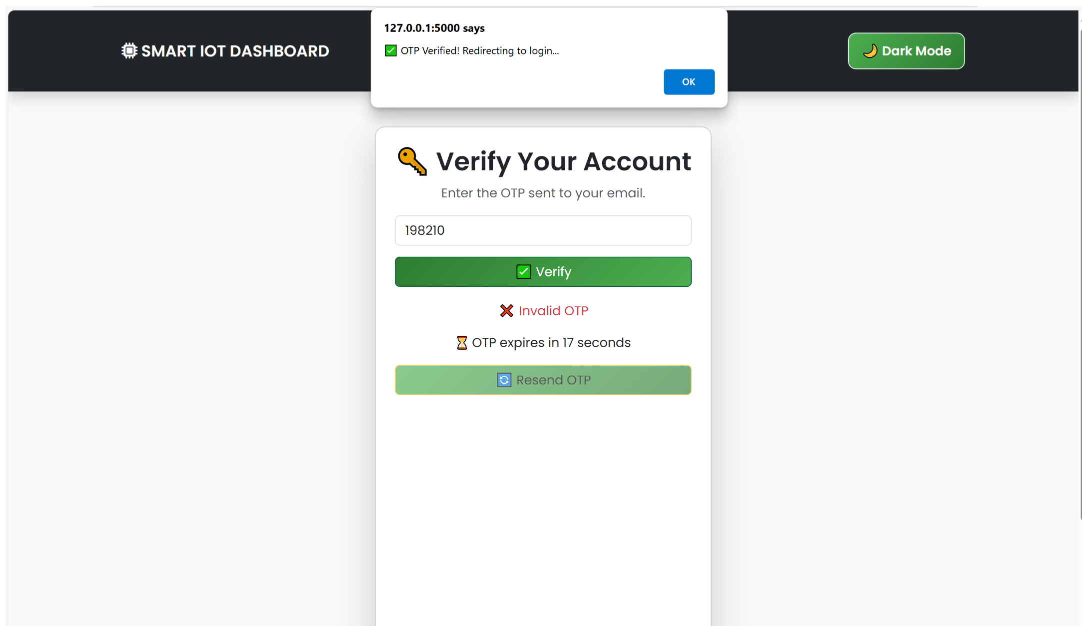

Figure 5.10: User database storing hashed passwords and OTP verification records.

The performance tests found that the OTP verification process is safe and fast, taking under two seconds. This provides a good user experience while maintaining security.

5.4.2 Password Hashing & Unauthorized Access Prevention

To keep user passwords safe, the system uses bcrypt for hashing. This way, stored passwords are encrypted so that no one can see the original passwords. Passwords are changed into a special format using a method called hashing and salting before they are kept in the database. This extra security layer means that if someone breaks into the database, they won't be able to get the original passwords because the hashing process can't be reversed. Security checks were done to see how strong the password protection was. Tests showed that even when trying to access the database directly, the stored passwords were still encrypted and unreadable. Bcrypt strengthens password security by making it hard for attackers to guess passwords quickly. Tests show that its hashing process helps protect against unauthorized access. Overall, bcrypt makes it very difficult for someone to breach credentials while still allowing for efficient login processes.

SQLiteStudio (3.4.17) - [users (users)]

Total rows loaded: 10

	id	name	email	password	otp	is verified	otp expires at
1	1	John Doe	johndoe@example.com	$2b$12$/qeypC2k4KkXyQcHstN8b.fAHA80oFybfFoMrPjEDJjfFuCbuCFiS	225406	0	NULL
2	2	John Doe	jonedoe@example.com	$2b$12$Ms1lW9X0GDwLKdg3qdugBe6HC7QKSCSOMnsrUoemrQykcGA85D4YG	091535	0	NULL
3	3	John Doe	zizidoe@example.com	$2b$12$SCIVz3xfB8kFAzP4m/gQpeTcstdNPlMIGMKHUkP9D2KnOwScBrJuy	477533	0	NULL
4	4	Zahra Shahrooei	zahra.shahrooeii@gmail.com	$2b$12$k/I3LFcmojbid4bU/RB26ubMdDFDfOstjrlrZ59vLsg8Ipk65DfHS	840578	0	NULL
5	5	zryyy	shahrueii.zahra@gmail.com	$2b$12$IBXbsD9TWxAQQ1Zy7k7sBeECGwykNdjYLxs/w8L3WhO92x1e5y2KK	903921	0	2025-03-05 17:12:47
6	6	zahra shahoigiv	zahra.shahrooei@studenti.unipd.it	$2b$12$YBQydtdJg8TDs7Syz3/Hyu3PqPz8KBzkLiJa9wDUwjcZrM5KM4QSi	513474	0	2025-03-05 17:24:21.182095
7	7	zahra shahoigivff	shahrooeii.zahra@gmail.com	$2b$12$8KLS45Ok0yeMioG.6pJ5COkIMxW2vrL3qO5cFk6HorK.VEp9P3dUu	NULL	1	NULL
8	8	ali tavakoli	www.ali.ty1@gmail.com	$2b$12$MdCEnBfX9lB0OJnbf6iZ8.iPSw8/n.tuxABOBSkuJC1eTDwZEQ7RO	NULL	1	NULL
9	10	Zahrashahrooei	zahrashahrooeiii@gmail.com	$2b$12$Z.K6jdktUNMuAPWcQNMWvuJMx9YRwRtaRm5vExRSAToKLlDnhXqWy	509322	0	2025-03-06 15:55:07.722490
10	11	Zahrashahrooei	zahrashahrooei1999@gmail.com	$2b$12$vAjlK8m1HByIBgvJGlN.NupNKy8nY6TD6qk2UaGv5g4jrdME38dRS	NULL	1	NULL

Figure 5.11: Database storing hashed passwords with bcrypt encryption.

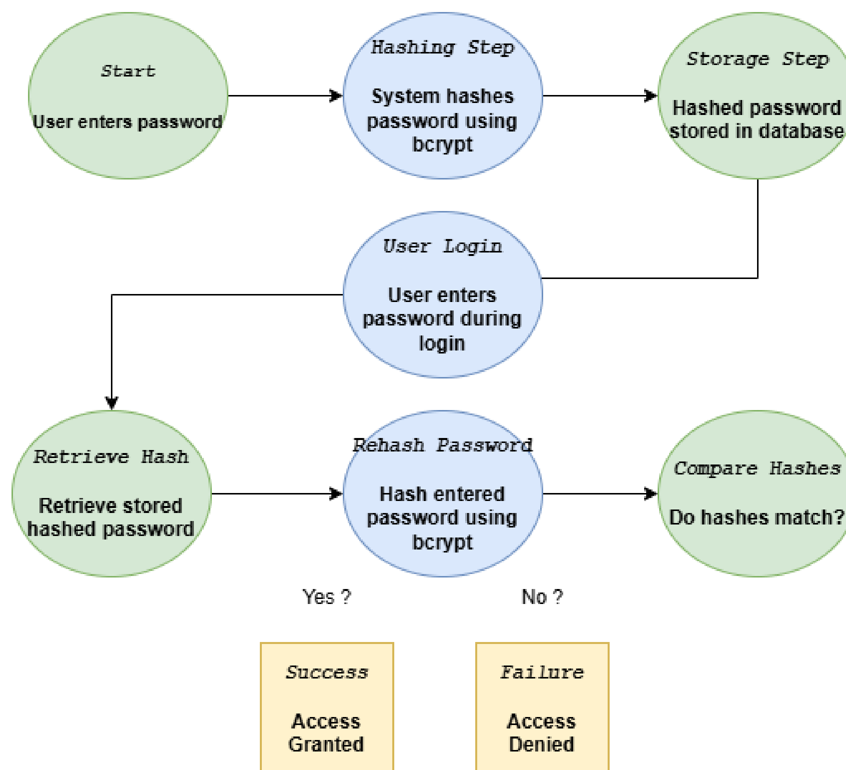

Figure 5.12: Password hashing and verification process using bcrypt.

5.4.3 API Security & Protection Against Attacks

APIs are the main connection between the frontend, backend, and database, making their security very important to protect user data. Strong API security is needed to stop unauthorized access, data leaks, and attacks. Several security methods were used to protect the API. JWT (JSON Web Token) authentication makes sure that only logged-in users can access certain API

features. Rate limiting controls the number of API requests to stop brute-force attacks and prevent overloading the system. CSRF protection prevents harmful cross-site request attacks that could affect user authentication and sensitive actions. Also, input validation and sanitization were used to block common web attacks like SQL injection and cross-site scripting (XSS) to keep the API safe from harmful inputs. To test these security steps, different checks were done. Invalid tokens were rejected, proving that JWT authentication worked well. The system blocked too many API requests, showing that rate limiting prevented brute-force attacks. CSRF protection successfully stopped unauthorized requests, making sure sensitive actions remained secure. These results confirm that the API security setup is strong and protects data properly.

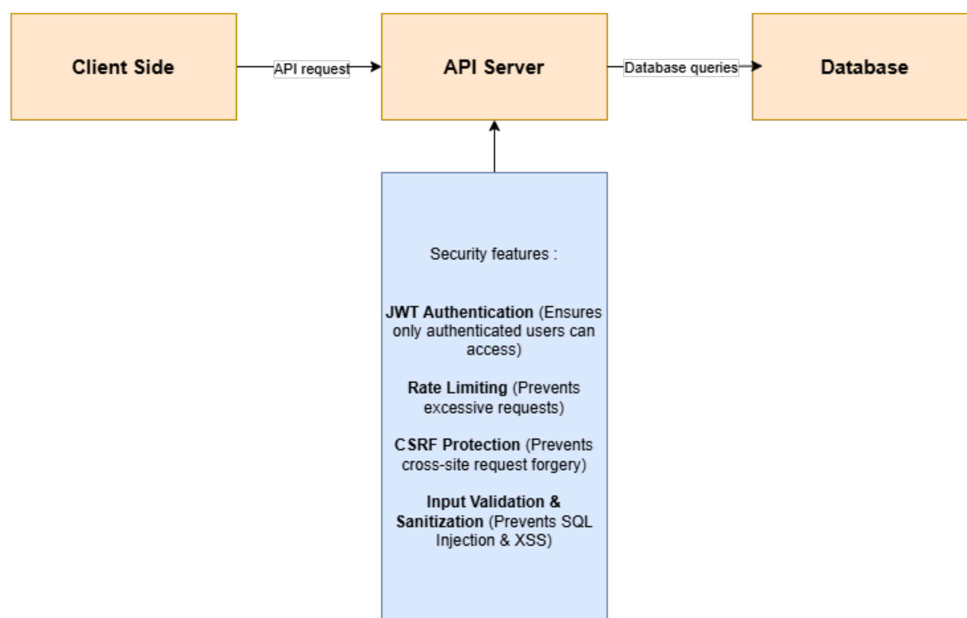

Figure 5.13: API Security Architecture Diagram

Chapter 6

Conclusion and Future Work

6.1 Summary of Research Contributions

This research explains how to create an easy IoT dashboard that lets users track their devices in real time. Here are the key features:

Development of a Full-Stack IoT Dashboard: The system integrates a frontend web interface, a backend API, and a database to handle sensor data, authentication, and actuator control. The dashboard provides users with real-time data visualization and interactive control over IoT devices.

Secure User Authentication: The implementation of JWT-based authentication, OTP verification, and password hashing enhances security by preventing unauthorized access.

Efficient Data Logging & Visualization: The dashboard continuously logs sensor data and provides historical trend analysis to help users make informed automation decisions. Interactive visualizations allow for easy interpretation of real-time changes in environmental conditions. These contributions show that the system is a **safe, flexible, and efficient** platform for IoT automation.

6.2 Practical Applications of the System

The developed IoT dashboard has multiple real-world applications, especially in smart automation and remote monitoring. By integrating real-time data collection and secure control mechanisms, the system can be applied in various domains:

6.2.1 Smart Home Automation

Smart Home Automation lets homeowners check and control things like temperature, humidity, and lighting from afar. Users can set up their appliances to work automatically based on specific rules, helping save energy and make their homes more comfortable.

6.2.2 Industrial IoT Monitoring

Factories and industrial facilities can use the dashboard for real-time monitoring of equipment and detecting anomalies. By tracking sensor data, businesses can maintain operational efficiency, anticipate maintenance needs, and minimize downtime.

6.2.3 Remote Sensor Data Management

This platform is designed for researchers, environmental monitoring agencies, and facility managers requiring continuous access to real-time data. The dashboard offers historical trends, analytics, and alerts, promoting proactive decision-making in critical environments.

6.3 Future Enhancements

The current IoT dashboard is well suited to verify data in real time, automate tasks and maintain online security. Still, there are ways to make it better for everyday use. Future plans will focus on adding hardware, saving energy, allowing remote access, and enhancing edge computing to create a full IoT automation platform.

6.3.1 Integration with Physical IoT Devices

To move beyond a simulation environment, the next phase involves **connecting actual IoT sensors and microcontrollers** such as **Raspberry Pi, ESP32, or Arduino** to the system. Instead of fetching weather data from an API, real-world sensors will **collect temperature, humidity, and motion data**, which will then be processed by the backend. This enhancement will enable **real-world testing** of automation mechanisms, actuator control, and security protocols.

6.3.2 AI & Machine Learning for Predictive IoT

Currently, automation follows **predefined threshold-based rules**. However, real-world conditions are dynamic, and static threshold-based automation has limitations. Future improvements will integrate **machine learning models** to enhance automation by learning from historical sensor data, detecting patterns, predicting trends, and optimizing energy efficiency.

6.3.3 User Availability & Smart Energy Suggestions

A new feature will allow users to **input availability and home or office conditions** to further customize automation settings. The system will generate **personalized energy-saving recommendations**, optimize automation behavior based on user schedules, and suggest real-time efficiency improvements.

6.3.4 Mobile Notifications & Remote Access

Expanding the system with **mobile push notifications and a mobile-friendly UI** will allow users to receive real-time alerts and manage automation from anywhere. Integration with cloud hosting will enable full **remote access**.

6.3.5 Edge AI for On-Device Decision Making

To reduce reliance on cloud computing, **Edge AI** will be implemented on IoT devices. This will lower **latency**, increase **reliability in automation**, and improve **security** by processing sensitive data locally.

6.3.6 Scalability & Multi-User Support

Future improvements will introduce **multi-user functionality** with different roles, such as **admin, regular users, and guest access**. This will expand the system's ability to support **multiple devices across different locations**, ensuring greater flexibility for diverse automation needs.

These enhancements will transition the system from a **simulated IoT dashboard into a fully operational, intelligent, and scalable automation framework**, suitable for **smart homes, industrial applications, and real-time monitoring systems**.

Acronym

IoT *Internet of Things*

AI *Artificial Intelligence*

ML *Machine Learning*

UI *User Interface*

HTTP *Hypertext Transfer Protocol*

TU/e Technical University of Eindhoven

M2M machine-to-machine

OTP *One-time Password*

Unipd University of Padua

CSRF Cross-Site Request Forgery

Bibliography

[1] Ala Al-Fuqaha et al. Blockchain and iot integration: Challenges and applications. *IEEE Communications Surveys & Tutorials*, 22(4):2521–2545, 2020.

[2] Tanvir Alam, Mazharul Chowdhury, Muhammad Raihan Khandaker, and Md Hassan. Blockchain-based iot security: A survey. *Sustainable Cities and Society*, 39:108–128, 2018.

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[4] Chaitali Bhunia et al. Federated learning in iot: A survey on communication efficiency and security challenges. *IEEE Communications Surveys & Tutorials*, 2021.

[5] Pranav Chopra et al. Iot-driven cybersecurity threats and challenges: A comprehensive review. *Future Generation Computer Systems*, 125:319–333, 2021.

[6] Mohamed Elrawy, Ahmed Awad, et al. Security and privacy in iot: A comprehensive survey. *Security and Communication Networks*, 2018:1–15, 2018.

[7] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2nd edition, 2018.

[8] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. In *Future generation computer systems*, volume 29, pages 1645–1660. Elsevier, 2013.

[9] R. Gupta et al. Smart city infrastructure powered by iot and ai. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[10] Wazir Zada Hassan, Syed Gillani, Sajjad Anwar, et al. Current research on internet of things (iot) security: A survey. *IEEE Access*, 7:127141–127167, 2019.

[11] Vikas Hassija, Vinay Chamola, Vaishali Saxena, Deepak Jain, Pooja Goyal, and Biplab Sikdar. A survey on iot security: Application areas, security threats, and solution architectures. *IEEE Access*, 7:82721–82743, 2019.

[12] Richard Hipp and Dennis W. Owen. *Using SQLite for IoT Applications*. Apress, 1st edition, 2018.

[13] Sungho Lee et al. Iot in healthcare: Applications and challenges. *IEEE Journal of Biomedical and Health Informatics*, 2020.

[14] Xiaolong Liu, Yu Sun, Qi Li, et al. Deep learning for iot big data and streaming analytics: A survey. *IEEE Internet of Things Journal*, 8(10):8285–8306, 2021.

[15] Rongpeng Lu, Hailing Zhang, and Hu Zhao. Low-power wide-area networks: An overview. *IEEE Internet of Things Journal*, 6(1):1–16, 2019.

[16] Mark Lutz. *Learning Python*. O'Reilly Media, 5th edition, 2019.

[17] Mehdi Mohammadi and et al. Deep reinforcement learning for iot security threat detection. *Pervasive and Mobile Computing*, 67:101371, 2020.

[18] PocketMagic. Simple iot temperature sensor, 2024. Accessed: 13-Feb-2025. URL: https://pocketmagic.net/simple-iot-temperature-sensor.

[19] Tie Qiu et al. Ai-enhanced iot security: Deep learning approaches. *IEEE Wireless Communications*, 28(3):72–78, 2021.

[20] Md Rahman, Mofizur Islam, et al. Iot security issues and solutions using machine learning. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[21] Anand Raman et al. Iot-based industrial automation: Challenges and opportunities. *IEEE Transactions on Industrial Informatics*, 2021.

[22] Walid Saad et al. Federated learning for secure iot applications: Concepts, challenges, and research opportunities. *IEEE Internet of Things Journal*, 7(4):3045–3060, 2020.

[23] AKM Mahfuzul Sikder, Gabriele Petracca, Hidayet Aksu, and A Selcuk Uluagac. A comprehensive survey on security and privacy issues in iot-enabled smart environments. *IEEE Communications Surveys & Tutorials*, 21(3):2642–2673, 2019.

[24] R. Singh et al. A deep learning approach for predictive maintenance in industrial iot. *Computers in Industry*, 127:103341, 2021.

[25] SumatoSoft. What's iot data visualization? best practices examples, 2024. Accessed: 2024-02-26. URL: https://sumatosoft.com/blog/whats-iot-data-visualization-best-practices-examples.

[26] Mohammad Bani Yassein, Shadi Aljawarneh, et al. Internet of things: Security and privacy concerns. *International Journal of Computer Applications*, 175(4):1–6, 2017.

[27] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. In *IEEE Internet of Things journal*, volume 1, pages 22–32. IEEE, 2014.

[28] Heng Zhang et al. Machine learning applications for intrusion detection in iot: A survey. *IEEE Internet of Things Journal*, 7(5):3431–3444, 2020.