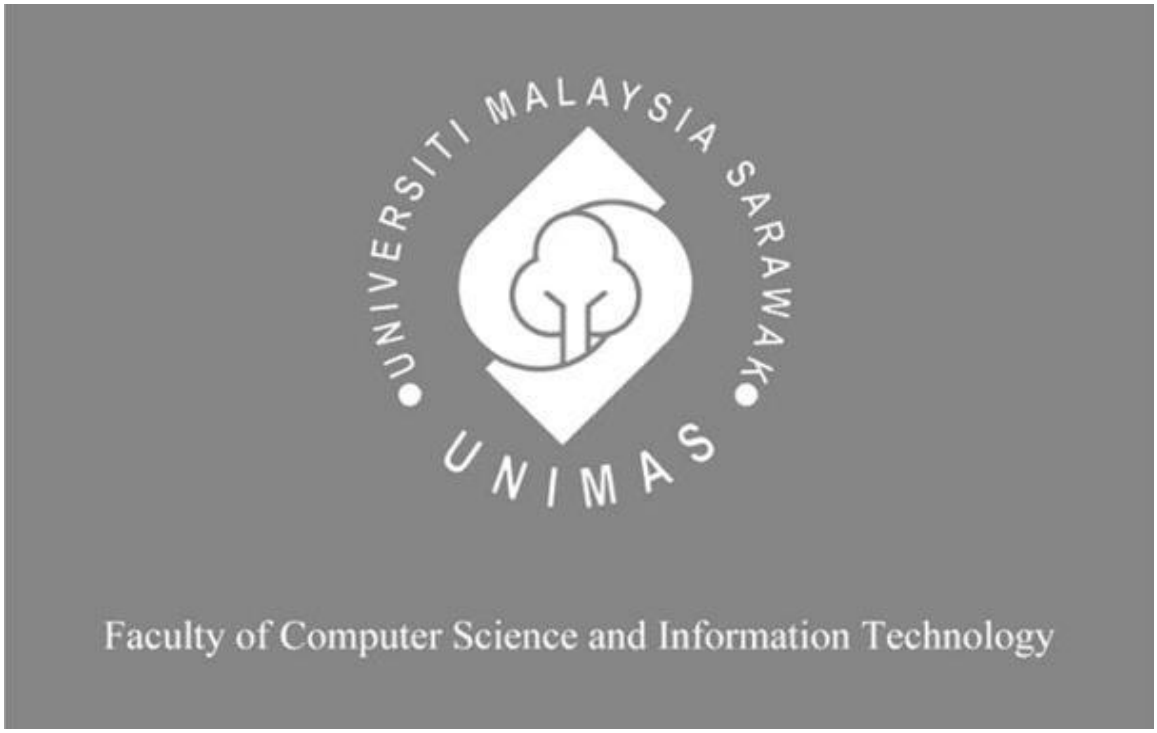



# BATTERY HEALTH AND LIFESPAN PREDICTION DIAGNOSTIC AND MONITORING SYSTEM

**VESHAL A/L K SANMUGAM**

**Bachelor of Software Engineering with Honours**

**2025**

# BATTERY HEALTH AND LIFESPAN PREDICTION DIAGNOSTIC AND MONITORING SYSTEM

VESHAL A/L K SANMUGAM

This project is submitted in partial fulfilment of

the requirements for the degree of Bachelor of Software Engineering with Honours

Faculty of Computer Science and Information Technology

UNIVERSITI MALAYSIA SARAWAK

2025

# SISTEM DIAGNOSTIK DAN PEMANTAUAN KESIHATAN SERTA RAMALAN JANGKA HAYAT BATERI

VESHAL A/L K SANMUGAM

Projek ini merupakan salah satu keperluan untuk Ijazah Sarjana Muda Kejuruteraan Perisian dengan Kepujian

Fakuliti Sains Komputer dan Teknologi Maklumat

UNIVERSITI MALAYSIA SARAWAK

2025

**UNIVERSITI MALAYSIA SARAWAK**

**THESIS STATUS ENDORSEMENT FORM**

**TITLE** BATTERY HEALTH AND LIFESPAN PREDICTION DIAGNOSTIC AND MONITORING SYSTEM

**ACADEMIC SESSION:** 2024/2025

VESHAL A/L K SANMUGAM

**(CAPITAL LETTERS)**

hereby agree that this Thesis* shall be kept at the Centre for Academic Information Services, Universiti Malaysia Sarawak, subject to the following terms and conditions:

1. The Thesis is solely owned by Universiti Malaysia Sarawak
2. The Centre for Academic Information Services is given full rights to produce copies for educational purposes only
3. The Centre for Academic Information Services is given full rights to do digitization in order to develop local content database
4. The Centre for Academic Information Services is given full rights to produce copies of this Thesis as part of its exchange item program between Higher Learning Institutions [ or for the purpose of interlibrary loan between HLI ]
5. ** Please tick ( √ )

| | | |
|---|---|---|
| ☐ | CONFIDENTIAL | (Contains classified information bounded by the OFFICIAL SECRETS ACT 1972) |
| ☐ | RESTRICTED | (Contains restricted information as dictated by the body or organization where the research was conducted) |
| ☑ | UNRESTRICTED | |

(AUTHOR'S SIGNATURE)

Validated by


(SUPERVISOR'S SIGNATURE)

Permanent Address

39, JALAN NURI 12,
BANDAR PUCHONG JAYA
47170 PUCHONG SELANGOR

Date: 25/07/2025

Date: 25/07/2025

Note * Thesis refers to PhD, Master, and Bachelor Degree

** For Confidential or Restricted materials, please attach relevant documents from relevant organizations / authorities

**DECLARATION**

I hereby declare that this research together with all its content is none other than that of my own work, with consideration of the exception of research-based information and relative materials that were adapted and extracted from other resources, which have evidently been quoted or stated respectively as references.

Signed,

VESHAL A/L K SANMUGAM

82219

Faculty of Computer Science and Information Technology 16th January 2025

University Malaysia Sarawak

**ACKNOWLEDGEMENT**

I would like to extend my sincere gratitude to those who have contributed to the successful completion of my individual final year project. First and foremost, I wish to express my heartfelt appreciation to my project supervisor, Dr. Mohamad Imran bin Hj.Bandan, for his guidance, support, and invaluable feedback throughout this journey. His expertise and encouragement were instrumental in shaping the quality and direction of my work. I would also like to express my deep gratitude to my project examiner, Dr. Amelia Jati Anak Robert Jupit, for her thorough evaluation and constructive comments, which have greatly contributed to ensuring the quality and comprehensiveness of my project.

I am thankful to the Faculty of Computer Science and Information Technology for providing a supportive academic environment and the necessary resources to facilitate my research and development.

I am also grateful to my peers and classmates for their encouragement and collaborative insights, which have enriched my learning experience during this project. Special thanks go to my family and friends for their unwavering support, understanding, and motivation throughout this academic endeavour.

Lastly, I appreciate the opportunity to undertake this individual project, which has allowed me to explore, apply, and expand my knowledge in meaningful and practical ways. To everyone who has been part of this journey, your contributions have been invaluable, and I am deeply thankful for your role in the successful completion of my final year project.

**Table of Contents**

## List of Figures

## List of Tables

**ABSTRACT**

Battery health degradation is a significant challenge that impacts the efficiency, safety, and longevity of battery-powered devices. Traditional monitoring methods often rely on reactive maintenance, leading to unexpected failures, reduced performance, and increased operational costs. This Final Year Project introduces an innovative solution for real-time battery health diagnostics and lifespan prediction through a comprehensive monitoring system. Leveraging advanced data analysis and predictive modeling techniques, the system processes key battery metrics such as voltage and temperature to evaluate performance and estimate the remaining lifespan. The proposed solution addresses the limitations of conventional battery monitoring by providing automated, real-time insights into battery status, helping users take proactive maintenance actions before failures occur. Core functionalities include data collection from sensors, processing of battery metrics, and predictive modeling to assess battery health. Notifications are generated based on analysis results to inform users of necessary interventions. By enhancing battery management practices, this system minimizes unexpected downtimes, optimizes performance, and contributes to more sustainable energy solutions. Through its innovative design, this project not only improves existing battery monitoring practices but also lays the groundwork for advancements in predictive maintenance and energy management technologies.

## ABSTRAK

Kemerosotan kesihatan bateri merupakan cabaran utama yang memberi kesan kepada kecekapan, keselamatan, dan jangka hayat peranti berkuasa bateri. Kaedah pemantauan tradisional sering bergantung pada penyelenggaraan reaktif, yang boleh menyebabkan kegagalan yang tidak dijangka, prestasi yang menurun, dan peningkatan kos operasi. Projek Tahun Akhir ini memperkenalkan penyelesaian inovatif untuk diagnostik kesihatan bateri secara masa nyata dan ramalan jangka hayat melalui sistem pemantauan yang komprehensif. Dengan memanfaatkan teknik analisis data dan pemodelan ramalan yang canggih, sistem ini memproses metrik utama bateri seperti voltan dan suhu untuk menilai prestasi serta menganggarkan jangka hayat yang tinggal. Penyelesaian yang dicadangkan menangani keterbatasan pemantauan bateri konvensional dengan menyediakan maklumat automatik dan masa nyata mengenai status bateri, membantu pengguna mengambil tindakan penyelenggaraan secara proaktif sebelum kegagalan berlaku. Fungsi utama sistem ini merangkumi pengumpulan data daripada sensor, pemprosesan metrik bateri, dan pemodelan ramalan untuk menilai kesihatan bateri. Pemberitahuan dijana berdasarkan hasil analisis untuk memaklumkan pengguna mengenai intervensi yang diperlukan. Dengan meningkatkan amalan pengurusan bateri, sistem ini mengurangkan masa henti yang tidak dijangka, mengoptimumkan prestasi, dan menyumbang kepada penyelesaian tenaga yang lebih lestari. Melalui reka bentuk inovatifnya, projek ini bukan sahaja memperbaiki amalan pemantauan bateri sedia ada tetapi juga meletakkan asas bagi kemajuan dalam penyelenggaraan ramalan dan teknologi pengurusan tenaga.

# CHAPTER 1: INTRODUCTION

## 1.1 Introduction/Background

A battery module is an assembly of multiple battery cells configured to deliver a specified voltage and capacity. These modules are widely used in applications requiring a substantial energy supply, such as electric vehicles, renewable energy storage, and high-demand electronic devices. Each cell contributes to the total power output and overall health of the module, making monitoring critical to ensure longevity and reliability.

Global reliance on battery-powered devices is rapidly increasing, with a notable shift toward sustainable energy applications, including electric vehicles (EVs) and renewable energy storage systems. By 2030, the global battery market is expected to exceed USD 322.2 billion, driven by the rising adoption of EVs and the demand for portable power in consumer electronics (Markets, 2025). This growth accentuates the need for efficient, predictive battery monitoring systems to maintain performance and safety in various usage scenarios.

While demand for batteries grows, so do the challenges in managing their health. Current monitoring methods often fail to provide comprehensive, real-time insights into factors like degradation, temperature impact, and voltage inconsistencies. These limitations lead to unexpected battery failures, increasing maintenance costs and posing risks in critical applications. Recent studies highlight that failures in battery cells can generate internal heat, potentially leading to combustion and severe operational hazards, particularly in electric vehicles (Kirana et al., 2023).

To address these challenges, machine learning models have emerged as a powerful tool for battery health diagnostics and lifespan prediction. By analysing historical performance data, machine learning algorithms can identify degradation patterns and

provide accurate predictions of battery lifespan. This predictive capability enables proactive maintenance, enhancing safety, reducing costs, and minimizing environmental impact by extending battery usage before disposal (Li et al., 2024). Integrating machine learning into battery monitoring systems represents a significant advancement in ensuring reliability, sustainability, and efficiency in battery-dependent applications.

### 1.2 Problem Statement

Batteries play a crucial role in numerous applications, from portable devices to electric vehicles. However, despite ongoing advancements in battery technology, real-time, highly accurate battery health prediction systems remain limited. Most existing methods struggle with challenges related to applicability, scalability, and accuracy under real-world conditions (Hong et al., 2023). The absence of robust monitoring systems often leads to undetected issues such as overheating, voltage drops, and rapid degradation, which can cause device failures or safety hazards (Zhao et al., 2024). As battery demand continues to increase, there is an urgent need for systems capable of real-time diagnostics and accurate lifespan prediction. Proactive monitoring would allow users to make timely repairs or replacements, significantly reducing the risk of failure.

### 1.3 Scope

This work focuses on developing a proof of concept for a battery health monitoring system applicable to a wide range of battery-powered devices. While the system is designed to support applications such as electric vehicles, energy storage systems, and portable electronics, testing will primarily be conducted on lithium-ion batteries due to their affordability and accessibility. The system will be designed as a web-based platform to ensure accessibility across multiple devices, enabling real-time monitoring of critical metrics such as battery percentage, voltage, temperature, and estimated remaining lifespan. By providing predictive insights and proactive alerts, the system will help users manage battery health more effectively, minimising failure risks and extending battery lifespan across various industrial and personal applications.

### 1.4 Aims and Objective

The aim of this project is to develop a battery health monitoring and lifespan prediction system that provides real-time insights, enabling users to optimise battery performance and longevity.

The overall objectives of this project are:

- To design a system that enables prediction towards battery lifetime.
- To develop a monitoring system for inspecting battery health and performance.
- To analyse the system using an Arrhenius-based technique, incorporating predefined simulation profiles—Healthy New, High-Temperature Stress, Aged High-Cycle, Low State-of-Charge Snapshot, and Near End-of-Life—along with real-time data.

## 1.5 Methodology

The development of this battery health monitoring system follows a structured software engineering approach to ensure a systematic progression from requirements gathering to implementation and testing. The Iterative Waterfall Model has been selected to allow for necessary refinements at different stages, particularly in system design and testing. Unlike the traditional Waterfall Model, the Iterative Waterfall Model incorporates feedback loops, enabling modifications to be made in earlier phases when errors or improvements are identified (GeeksforGeeks, 2024). This approach reduces costly rework and ensures that errors are contained within their respective phases rather than being discovered only at the final testing stage.

Figure 1.1 illustrates the development workflow, outlining the sequential phases from requirements analysis and data collection to system deployment. The methodology consists of four primary phases: Requirements Analysis and Data Collection, System Design, Implementation, and Testing. Each phase is described in detail below.

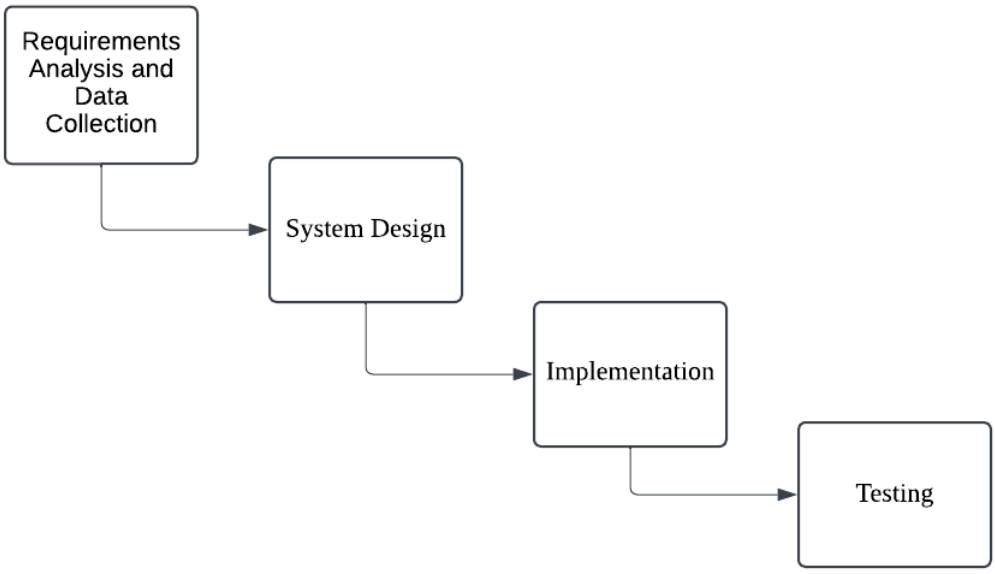



*Figure 1.1: Flow Chart of Iterative Waterfall Model*

**Phase 1: Requirements Analysis and Data Collection**

- Requirements Gathering: Begin by identifying all system requirements to monitor battery health effectively, defining essential metrics such as battery percentage, voltage, temperature, and degradation rate. Conduct a stakeholder analysis to ensure that all end-user needs, including those of businesses and individual users, are accurately addressed.
- Data Collection: Collect relevant datasets that represent battery health metrics and lifecycle stages. A physical prototype will be implemented using sensors to capture real-time data, ensuring accurate monitoring of battery health. Document how data will be filtered and stored for analysis, and select pre-existing datasets or develop simulations to support initial system testing.

**Phase 2: System Design**

- Architecture Design: Design the overall architecture of the web-based platform, specifying components for data acquisition, storage, processing, and display. Establish the integration of prediction models within this architecture, choosing a backend that supports scalability and data security, such as Firebase or AWS.
- User Interface (UI) Design: Develop a detailed UI layout to visualize battery metrics and lifespan predictions. Ensure the design is user-friendly and accessible on various devices. The interface should allow users to view current battery health and receive real-time notifications of any critical changes, such as impending degradation or performance issues.
- System Diagrams:
    - Context Diagram/ DFD Level 0: The Context Diagram (or DFD Level 0) represents the system at the highest level of abstraction. It shows the entire system as a single process and its interactions with external entities such

as users, administrators, and external databases. The data flows between these entities and the system are depicted to outline the inputs and outputs of the system.

- DFD Level 1 and Level 2: DFD Level 1 breaks down the single high-level process from the Context Diagram into major sub-processes. These sub-processes illustrate the key functionalities of the system, such as Data Collection, Battery Health Prediction, and Report Generation, along with their interactions with data stores and external entities. DFD Level 2 provides further detail by decomposing individual sub-processes from Level 1. For example, the Battery Health Prediction process could be broken into tasks like retrieving processed data, running prediction algorithms, and saving the results. This level shows the most granular interactions and ensures a comprehensive understanding of system workflows.
- Entity-Relationship Diagram (ERD): The ERD defines the data structure and relationships within the system. Key entities such as Administrator, User, Battery, Processed Battery Data, Prediction, and Notification are identified, with their attributes and relationships visualized. For instance:
  - An Administrator can manage multiple Users.
  - A User owns multiple Batteries, and each Battery has associated Processed Data and Predictions.
  - A User may receive multiple Notifications, representing system interactions or alerts.

These diagrams provide a visual representation of the system's functionality, data management, and relationships, ensuring clarity in design and communication across development stages.

**Phase 3: Implementation**

- Backend Development: A FastAPI service handles data ingestion, storage and processing. The prediction module, written in Python with scikit-learn utilities, estimates remaining battery life from live voltage, temperature and cycle data. Database operations use SQLite for local persistence, and REST endpoints expose the data securely to the front-end.
- Frontend Development: A React interface built with TypeScript, Chakra UI and Recharts displays real-time battery metrics. The dashboard streams telemetry every second, renders state-of-health and remaining-life charts, and provides intuitive controls for navigation, filtering and notification management.

**Phase 4: Testing**

- Scenario Execution: An automated TypeScript harness runs each of the five battery profiles through one hundred simulation steps, captures one API response per second and saves both a JSON log of responses and a CSV export of the telemetry table for later audit.
- Predictive-Accuracy Verification: After the final step the harness compares the model's state-of-health and remaining-life values with deterministic reference calculations and accepts results that differ by no more than five percentage points for SoH and five hours for RUL.
- Status-Classification Check: The harness confirms that each battery's final SoH triggers the correct dashboard label (Healthy, Warning or Critical) and that notifications appear when thresholds are crossed.

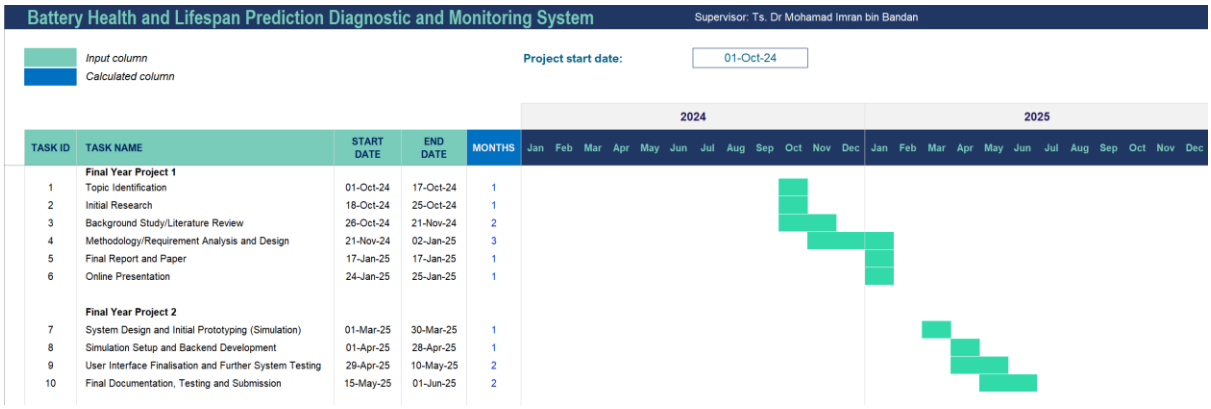
## 1.6 Significance of Project

The developed battery health and lifetime monitoring system has significant practical applications in everyday life and industry. By continuously tracking metrics like battery percentage, voltage, and temperature, the system offers users critical insights that enable proactive maintenance. This real-time monitoring can prevent unexpected battery failures, which is especially crucial in applications requiring high reliability, such as electric vehicles and renewable energy storage systems.

Furthermore, the system's predictive capabilities allow users to anticipate and address battery degradation early, minimizing the risks of performance loss and extending the functional lifespan of battery-powered devices. This proactive approach not only improves safety but also contributes to cost savings, reduces electronic waste, and supports sustainable energy practices by maximizing the usable life of batteries across diverse settings.

## 1.7 Project Schedule

Figure 1.2 displays the Gantt chart that defines the task for the project that will be completed in a period of two academic semesters.




| TASK ID | TASK NAME | START DATE | END DATE | MONTHS |
|---|---|---|---|---|
| | **Final Year Project 1** | | | |
| 1 | Topic Identification | 01-Oct-24 | 17-Oct-24 | 1 |
| 2 | Initial Research | 18-Oct-24 | 25-Oct-24 | 1 |
| 3 | Background Study/Literature Review | 26-Oct-24 | 21-Nov-24 | 2 |
| 4 | Methodology/Requirement Analysis and Design | 21-Nov-24 | 02-Jan-25 | 3 |
| 5 | Final Report and Paper | 17-Jan-25 | 17-Jan-25 | 1 |
| 6 | Online Presentation | 24-Jan-25 | 25-Jan-25 | 1 |
| | **Final Year Project 2** | | | |
| 7 | System Design and Initial Prototyping (Simulation) | 01-Mar-25 | 30-Mar-25 | 1 |
| 8 | Simulation Setup and Backend Development | 01-Apr-25 | 28-Apr-25 | 1 |
| 9 | User Interface Finalisation and Further System Testing | 29-Apr-25 | 10-May-25 | 2 |
| 10 | Final Documentation, Testing and Submission | 15-May-25 | 01-Jun-25 | 2 |

*Figure 1.2: Gantt Chart of Final Year Project 1 and Final Year Project 2*

## 1.8 Expected Outcome

By the end of this project, a comprehensive battery health and lifetime monitoring system will be developed. This system will continuously track essential metrics, including battery percentage, voltage, and temperature, to provide real-time insights into battery health. Additionally, it will predict the remaining lifespan of batteries, enabling proactive alerts for users before significant degradation occurs, thereby improving safety, reliability, and efficient battery usage.

## 1.9 Project Outline

This project report comprises six chapters that systematically describe the proposed Battery Health and Lifespan Prediction Diagnostic and Monitoring System.

### 1.9.1 Chapter 1: Introduction

This chapter introduces the project, outlining its objectives, scope, and significance. It provides a detailed problem statement, justification for the project, a summary of the proposed methodology, and an anticipated project timeline. The expected outcomes and the overall purpose of the system are also presented.

### 1.9.2 Chapter 2: Background and Literature Review

This chapter examines existing battery monitoring systems and predictive models through a thorough literature review. It explores methodologies, technologies, and solutions currently adopted in the industry, highlighting their advantages and limitations. The findings guide the development of an enhanced system capable of real-time monitoring and lifespan prediction.

### 1.9.3 Chapter 3: Requirement Analysis and Design

This chapter delves into the analysis of system requirements, both functional and non-functional, and details the specifications for hardware and software. It includes system design documentation such as:

- Context diagram
- Data flow diagram
- Entity-relationship diagram (ERD)
- System architecture
- User interface (UI) design

The design ensures scalability and accessibility for users on web-based platforms.

### 1.9.4 Chapter 4: Implementation

This chapter details the system's development, focusing on the integration of prediction models, data processing methods, and data visualization features. It includes technical aspects of system implementation, architecture, and component design. Screenshots and diagrams illustrate key system components and their functionality.

### 1.9.5 Chapter 5: Results and Analysis

This chapter presents the outcomes of all simulation scenarios and explains their implications for battery health monitoring. It details the data captured for each profile, displays the resulting graphs and tables, and analyses how temperature, state of charge and cycle history affect state of health and remaining-life predictions. The chapter also summarises validation results for predictive accuracy, status classification and system performance, providing evidence that the prototype meets its functional objectives.

### 1.9.6 Chapter 6: Conclusion and Future Work

The final chapter evaluates the system based on testing results and user feedback. It summarizes the project's achievements, acknowledges any limitations, and suggests potential enhancements for future work. Recommendations for deploying the system in real-world scenarios are also provided.

## 1.10 Summary

This project introduces the Battery Health and Lifespan Prediction Diagnostic and Monitoring System, a web-based solution that addresses the challenge of tracking and maintaining battery health across diverse applications. The system blends real-time data acquisition with predictive analytics to estimate remaining battery life, giving users actionable insights that support proactive maintenance and help prevent unexpected failures. Metrics such as state-of-charge, voltage, temperature and cycle count are streamed to the back end, processed into state-of-health and remaining-life estimates, and displayed through an intuitive dashboard.

The prototype relies on a FastAPI back end with a local SQLite store and a React front-end built-in JavaScript. Users can view critical battery parameters in real time and receive visual alerts when conditions indicate overheating, accelerated degradation or an imminent end-of-life state. By simulating battery behaviour under five scripted scenarios, Healthy New, High-Temperature Stress, Aged High-Cycle, Low State-of-Charge Snapshot and Near End-of-Life, the system demonstrates how predictive monitoring surpasses traditional, reactive approaches.

Through its combination of continuous monitoring, clear visualisation and scenario-tested forecasting, the system offers a cost-effective foundation for smarter maintenance decisions in both industrial and consumer contexts. By enabling informed choices about battery replacement and care, the platform can improve reliability, support sustainability goals and enhance safety in energy-dependent operations.

# CHAPTER 2: BACKGROUND AND LITERATURE REVIEW

This chapter explores the foundational aspects and existing methods related to the *Battery Health and Lifespan Prediction Diagnostic and Monitoring System*. It delves into the challenges of battery health monitoring and prediction, reviews current techniques, and highlights the importance of a comprehensive diagnostic and predictive approach.

## 2.1 Background of the Problem

Modern technologies across various domains heavily depend on battery systems for energy storage. From electric vehicles and portable electronic devices to renewable energy systems, batteries are integral to daily life. However, as batteries age, they experience a decline in performance, leading to reduced efficiency, lower capacity, and eventual failure. This degradation process, influenced by factors like charge-discharge cycles, temperature variations, and improper handling, poses significant challenges to industries and consumers alike (Liu et al., 2023).

For example, lithium-ion batteries, known for their high energy density, are particularly vulnerable to overcharging, deep discharges, and exposure to extreme temperatures. Without effective monitoring and predictive systems, unexpected battery failures can lead to operational downtimes, safety risks, increased maintenance costs, and inefficient resource utilization.

Predicting battery lifespan is a complex endeavour due to the nonlinear nature of degradation processes. Environmental conditions, usage patterns, and chemical dynamics add layers of complexity to accurate predictions. Traditional methods, such as manual inspections or periodic voltage checks, are no longer sufficient to meet the demands of modern applications.

Recent advancements in technology have introduced innovative solutions, such as IoT-enabled monitoring systems and machine learning models, which aim to address these challenges. IoT systems provide real-time data on battery performance metrics, while machine learning algorithms analyse historical data to predict the remaining useful life (RUL) with improved accuracy. These approaches not only enhance operational efficiency but also contribute to sustainability by optimizing battery usage and reducing waste.

This project focuses on developing a Battery Health and Lifespan Prediction Diagnostic and Monitoring System that integrates real-time data acquisition with machine learning models. The system aims to monitor critical battery metrics such as voltage, temperature, and state of charge while providing predictive insights to enable proactive maintenance and improve battery lifespan estimation. This aligns with the project's objectives of functionality, accuracy, and real-world applicability, ensuring a cost-effective and scalable solution for battery health monitoring.

## 2.2 Current Solution of Battery Health and Lifespan Prediction

This section explores the existing solutions and approaches used for diagnosing and monitoring battery health and predicting battery lifespan. It highlights advancements in technology, tools, and methodologies implemented in current systems to ensure accurate and reliable predictions.

### 2.2.1 Monitoring Techniques for Battery Modules

This subsection focuses on the various techniques and technologies employed to monitor battery modules. It discusses the mechanisms used to collect, process, and analyse battery data, enabling effective management and lifespan prediction.

**Ampere-hour (Ah) Counting Technique (Direct Measurement)**

The Ampere-hour (Ah) counting technique is one of the most widely used methods for estimating the State of Health (SOH) of batteries due to its simplicity (Pradhan & Chakraborty, 2022). This technique measures the quantity of Coulombs (Ah) charged or discharged during the charging and discharging process. The maximum available capacity of the battery is determined by fully charging and subsequently fully discharging it. The SOH is calculated using the following Equation 2.1:

$$\text{SOH} = \frac{Qmax}{Qnominal} \; X \; 100\% \qquad (2.1)$$

where Qmax represents the maximum available capacity of the aged battery, and Qnominal is the nominal capacity of a new battery (Pradhan & Chakraborty, 2022).

To ensure accurate capacity measurement, a complete charge-discharge cycle is required, which is both time-consuming and energy-intensive (Pradhan & Chakraborty, 2022). Additionally, minimizing cumulative error demands highly precise electrical probes. This charging-discharging process is typically conducted in controlled laboratory environments, using low current values at room temperature, though these conditions do not always reflect real-world applications (Pradhan & Chakraborty, 2022).

Despite its limitations, Ah counting remains essential for validating capacity measurements obtained through other methods. It plays a significant role in equivalent circuit models (ECM) for state-of-charge and capacity estimation. Unlike traditional Ah counting, ECM does not require a fully charged battery, making it more suitable for onboard applications (Pradhan & Chakraborty, 2022). Furthermore, this technique is less influenced by external factors such as depth of discharge, temperature, and C-rate compared to other approaches.

To improve the accuracy of Ah counting, Zhang et al. (2022) introduced an enhanced ampere-hour integral method incorporating a Long Short-Term Memory (LSTM) model for State-of-Charge (SOC) estimation. Their approach effectively mitigates cumulative errors and compensates for inaccuracies due to varying operating conditions. By integrating machine learning-based correction, this method provides more precise and reliable battery health estimations without requiring complete charge-discharge cycles (Zhang et al., 2022). Such advancements significantly enhance real-world battery monitoring and management, making them highly relevant for modern diagnostic systems.

**Electrochemical impedance spectroscopy (EIS) (Direct Measurement)**

Electrochemical Impedance Spectroscopy (EIS) is a widely used technique for analysing battery health by measuring impedance changes over different frequency ranges. The impedance response of a battery varies due to factors such as charge transfer resistance, solid electrolyte interphase (SEI) resistance, and double-layer capacitance (Pradhan & Chakraborty, 2022). EIS is particularly useful in estimating the State of Health (SOH) of batteries by analysing impedance spectra at different aging stages and environmental conditions (Liu et al., 2023).

The impedance spectrum generated through EIS reflects the interplay of various electrochemical phenomena within the battery, as illustrated in Figure 2.1, which presents a typical impedance spectrum test waveform for lithium-ion batteries (LIB). The EIS method operates by applying a small-amplitude sinusoidal excitation signal across a range of frequencies, allowing the impedance spectrum to be measured. This spectrum helps determine equivalent circuit model (ECM) parameters, such as solution resistance, charge transfer resistance, and Warburg impedance, which change as the battery ages (Liu et al., 2023). At high frequencies, inductive effects due to battery wiring and electrode

porosity dominate, while at lower frequencies, capacitive effects and diffusion processes become more significant (Pradhan & Chakraborty, 2022).

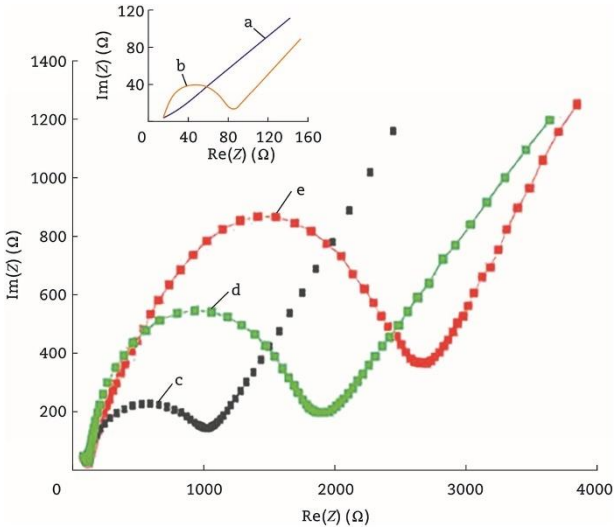



*Figure 2.1: Impedance spectrum test waveform of Lithium-Ion Battery (ref. from Liu et al., 2023).*

The temperature and state of charge (SOC) significantly affect the impedance spectrum. A study on prismatic LiFePO4 cells demonstrated that impedance increased at lower temperatures, and charge transfer resistance was highly influenced by SOC levels (Wang et al., 2019). As batteries age, impedance spectra exhibit distinct characteristics: the overall impedance shifts rightward, the high-frequency arc decreases, and the intermediate-frequency arc becomes more pronounced, indicating increased resistance in the SEI layer and charge transfer process (Wang et al., 2019).

Despite its advantages in providing detailed battery health diagnostics, EIS faces challenges in real-world applications. Factors such as SOC and temperature introduce variability in impedance-based SOH estimation, necessitating advanced identification algorithms and calibration methods (Pradhan & Chakraborty, 2022). To address these challenges, multi-ECM methods have been proposed to account for temperature

influences, and simplified multi-point impedance techniques have been developed to improve calculation efficiency (Liu et al., 2023).

Although EIS provides accurate and rich parameter data, it requires specialised instruments and complex measurement setups, limiting its practicality for online battery monitoring. Recent advancements aim to enhance online EIS measurement by employing low-frequency step waves and pulse impedance measurement methods, which can offer quicker and more practical assessments of battery impedance while maintaining reasonable accuracy (Liu et al., 2023; Zhou et al., 2018).

**Machine Learning Technique (Data – Driven Model)**

Machine learning (ML) algorithms have become a key approach for estimating the State of Health (SOH) of batteries. These techniques focus on constructing predictive models from training data, enabling accurate SOH approximations without explicit programming (Pradhan & Chakraborty, 2022). As a fundamental artificial intelligence method, ML has been widely applied in fields such as healthcare, finance, and image processing, and its adoption for battery health monitoring continues to grow.

Pradhan and Chakraborty (2022) identified several ML methods commonly used for SOH estimation:

- **Gaussian Process Regression (GPR):** Known for providing probabilistic predictions and quantifying uncertainties, GPR is crucial for robust battery health monitoring.
- **Neural Networks (NNs):** These models excel at capturing complex and nonlinear relationships in battery parameters, making them highly effective for detailed SOH analysis.
- **Support Vector Machines (SVMs):** Since battery datasets often exhibit high-dimensional characteristics, SVMs are particularly useful for processing such data efficiently.
- **Fuzzy Logic:** By employing rule-based reasoning, fuzzy logic helps manage uncertainty and imprecision, leading to reliable SOH predictions.
- **Markov Chain Models:** These models analyse sequential state transitions in battery health over time, offering a dynamic perspective on SOH estimation.
- **Monte Carlo Methods:** Used for probabilistic simulations, these methods assess uncertainties and variabilities in battery performance.

Despite their effectiveness, ML-based SOH estimation methods face notable challenges. The collection of training data is both time-intensive and costly, presenting a significant barrier to widespread implementation (Pradhan & Chakraborty, 2022). Additionally, the accuracy of these models heavily depends on optimal feature selection and parameter tuning poor choices can result in substantial prediction errors or algorithm convergence issues.

The advent of big data platforms offers a promising solution, enabling the collection of vast datasets that can improve model training while reducing computational demands on hardware. However, as Pradhan and Chakraborty (2022) noted, handling

large datasets introduces new challenges, including the need for effective data screening to minimize errors from wrongly selected parameters.

In conclusion, ML techniques provide powerful, scalable solutions for battery health monitoring, enhancing SOH prediction accuracy. However, continued advancements in data collection, parameter optimisation, and algorithm design are necessary to maximise their real-world applicability.

### 2.2.2 Techniques for Battery Lifespan Prediction

Battery lifespan prediction is a critical aspect of modern battery research, driven by the need for reliable and efficient energy storage solutions. Accurate prediction techniques enable the assessment of battery health and performance over time, facilitating timely maintenance and reducing the risk of unexpected failures. This section explores the various techniques employed in battery lifespan prediction, focusing on their methodologies, strengths, and challenges in providing actionable insights into battery degradation and longevity. According to Qu et al. (2024), when it comes to modelling approaches, battery models are generally divided into three main categories: physics-based models, empirical models, and equivalent circuit models (ECMs), as illustrated in Figure 2.2.

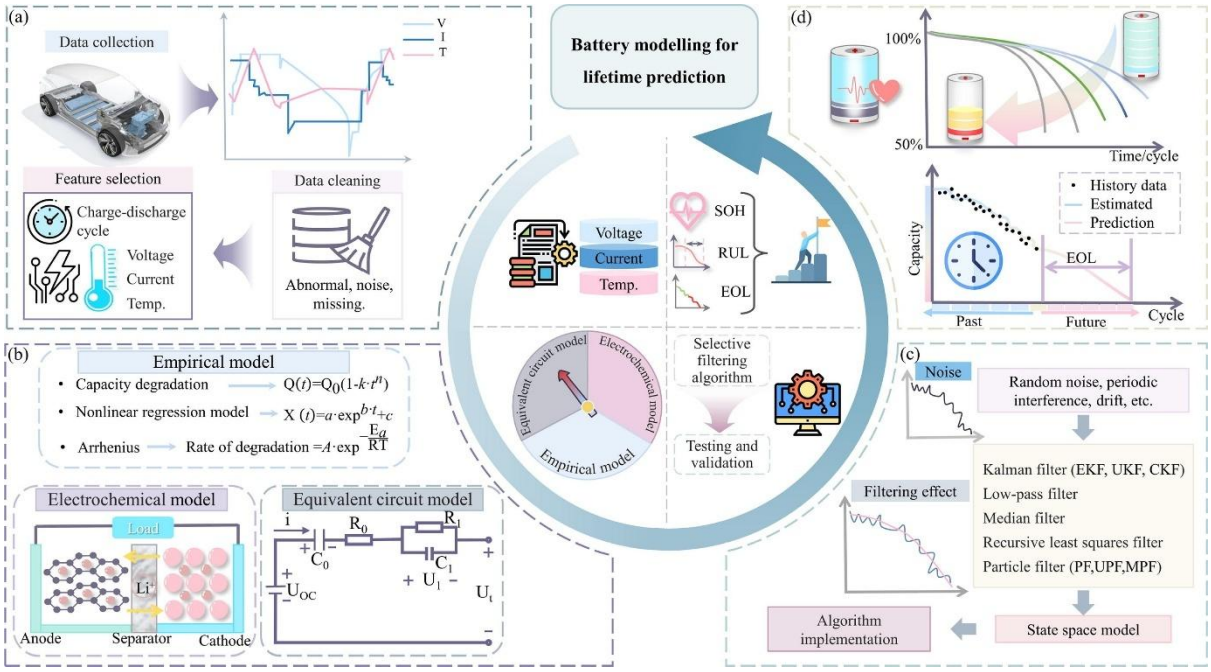



*Figure 2.2: Modelling for battery health assessment and lifespan prediction (adapted from Qu et al, 2024)*

**Physics-based Models**

Physics-based models are widely used for predicting battery lifespan as they capture the electrochemical processes and degradation mechanisms influencing battery health. Qu et al. (2024) highlight that these models rely on fundamental physical principles to estimate the remaining useful life (RUL) of a battery by analysing operating conditions and historical performance. Unlike purely data-driven methods, they provide insights into degradation modes such as loss of active material and lithium inventory, which are directly linked to mechanisms like SEI growth and lithium plating (Qu et al., 2024).

There are different types of physics-based models, each with varying complexity and applicability. Guo et al. (2022) categorise these into several approaches, including the Pseudo-Two-Dimensional (P2D) model, Single Particle Model (SPM), Equivalent Circuit Model (ECM), and Semi-Empirical Models. The P2D model provides detailed simulations by incorporating partial differential equations (PDEs) to describe electrochemical

reactions but demands significant computational power, making it difficult to implement in real-time battery management systems (Guo et al., 2022). In contrast, the SPM simplifies these reactions using ordinary differential equations (ODEs), offering reduced computational complexity while remaining accurate for low-to-medium C-rates (Guo et al., 2022).

On the other hand, ECMs model battery behavior based on impedance characteristics, providing a computationally efficient approach suitable for real-time applications. However, they struggle with accuracy at low states of charge (SOC) or under high-current conditions (Guo et al., 2022). Semi-empirical models rely on experimental data and power-law relationships to estimate degradation, making them easy to implement but prone to errors unless combined with physics-based methods (Guo et al., 2022).

Physics-based models offer structured and interpretable predictions but face challenges in parameterization and computational feasibility. Even the most detailed models may not capture every degradation scenario, and inverse modelling for parameter estimation remains computationally intensive (Qu et al., 2024). Recent efforts focus on integrating physics-based models with machine learning to improve efficiency and adaptability (Qu et al., 2024). This hybrid approach could enhance real-time battery health monitoring while maintaining interpretability, making it a promising direction for practical deployment.

**Empirical Models**

Battery degradation is commonly assessed using the capacity fade curve, which tracks capacity change over time, charge throughput, or equivalent cycle number (Pradhan & Chakraborty, 2022; Qu et al., 2024). An intuitive approach to predicting battery lifetime is to develop empirical or semi-empirical capacity fade models that incorporate parameters such as time and charge throughput (Pradhan & Chakraborty, 2022; Qu et al., 2024). These models often follow a square-root-of-time dependence due to diffusion-limited SEI formation, while temperature dependence is modeled using Arrhenius kinetics (Qu et al., 2024).

Enhancements to these models have introduced additional factors such as C-rate, average state of charge (SOC), depth-of-discharge (DoD), and voltage (Pradhan & Chakraborty, 2022; Qu et al., 2024). Modern empirical models distinguish between calendar aging, which is time-dependent, and cycle aging, which relies on charge throughput (Qu et al., 2024). However, these models require extensive datasets, struggle with cell-to-cell variations, and often fail to capture complex interactions between aging mechanisms, leading to inaccuracies in predicting degradation rates (Qu et al., 2024).

To improve accuracy, empirical models analyze raw time series data, such as voltage, current, and temperature, collected during battery cycling (Pradhan & Chakraborty, 2022; Qu et al., 2024). By identifying correlations within large datasets, regression models or empirical formulas can be applied to predict degradation trends (Qu et al., 2024). Specific aging models, including the weighted Ah aging model and the event-oriented aging model, have been developed for remaining useful life (RUL) estimation (Qu et al., 2024). Instead of relying on predefined invariant degradation models, recent studies have employed empirical degradation models based on time series analysis, incorporating autoregressive models and survival regression techniques (Qu et al., 2024). The use of

parametric bootstrap methods further improves confidence interval estimation, enhancing prediction reliability (Qu et al., 2024).

Under low-current conditions, quadratic polynomial regression models effectively capture degradation trends, particularly for nickel-metal hydride (NiMH) batteries (Qu et al., 2024). Additionally, empirical internal resistance models have been developed to predict both calendar aging and cycle-life degradation across various temperature ranges (Qu et al., 2024). Some models also incorporate health indicators (HIs), such as fully discharged voltage, internal resistance, rate of temperature change, and battery discharge curves, to refine predictions (Qu et al., 2024).

Although empirical models effectively establish correlations between RUL and battery characteristics, these correlations evolve as batteries age and operating conditions change (Qu et al., 2024). To address this, filtering algorithms such as Kalman filters (KF) and particle filters (PF) are often integrated to allow real-time updates of model parameters (Qu et al., 2024). PFs are particularly useful for handling nonlinear and non-Gaussian noise in state estimation, improving prediction accuracy (Qu et al., 2024). Recent research has focused on refining PF algorithms, addressing computational efficiency issues, and exploring novel techniques to enhance prediction accuracy (Qu et al., 2024).

In conclusion, empirical models provide an efficient method for battery lifetime prediction by capturing capacity fade behaviour under different operating conditions. The incorporation of raw time series data, autoregressive models, bootstrap methods, and advanced filtering algorithms has significantly improved the accuracy and reliability of predictions. As research continues to refine these models, their role in battery prognostics and maintenance optimization will become even more critical.

**Random Forests (Common Machine Learning Technique)**

In the study by Qu et al. (2024), the application of Random Forests (RF) for predicting battery health is examined, particularly in estimating the state of health (SOH) of batteries. RF, as an ensemble learning method, constructs multiple decision trees from various sub-samples of the data, which are then averaged to reduce overfitting. This makes RF highly effective in handling large datasets with numerous input variables, capturing complex, nonlinear relationships within the data. The ability of RF to handle missing values while maintaining high accuracy further reinforces its suitability for battery health prediction (Qu et al., 2024).

The RF model for battery health estimation utilizes features extracted from battery performance data, such as voltage, current, and temperature readings. A significant advantage of RF is its ability to make predictions directly from raw data, eliminating the need for extensive data preprocessing. However, the trade-off is the reduced interpretability of the model, as the outputs are derived from aggregating multiple decision trees, making it more challenging to trace the rationale behind specific predictions (Qu et al., 2024).

Qu et al. (2024) employed incremental capacity analysis for feature selection, extracting key health indicators (HIs) like charge and discharge voltage as inputs for the RF model. Their experiments demonstrated that RF models trained on these selected features could predict battery health with high accuracy, achieving a predictive error rate below 1.3%. Sultan et al. (2025) further compared RF with other machine learning models, such as k-Nearest Neighbors (kNN), Gradient Boosting, and Dense Neural Networks (DenseNN), in predicting the remaining useful life (RUL) of batteries. Their study found that RF achieved one of the highest R-squared (R2) values, indicating a strong model fit, and maintained a low Mean Absolute Error (MAE), reflecting high prediction accuracy.

However, kNN slightly outperformed RF in terms of predictive accuracy, while RF remained a close second with robust performance (Sultan et al., 2025).

The process begins with the collection of raw data, from which relevant features are extracted. RF then trains a series of decision trees using these features, with each tree contributing to the final prediction. This parallel operation of decision trees ensures robustness against noisy data. During the testing phase, the model's performance is evaluated using error metrics such as root mean square error (RMSE) and mean absolute error (MAE), which measure the accuracy of the predictions. Sultan et al. (2025) highlighted that RF models performed optimally when multiple features were used, reinforcing the importance of feature selection for improving prediction reliability.

One of the strengths of RF is its robustness in handling missing data. Even when significant portions of the dataset are incomplete, RF maintains high predictive accuracy. However, one limitation of RF models is that they may not always provide highly precise estimates for continuous numerical predictions, such as the exact remaining useful life (RUL) of a battery (Qu et al., 2024). Sultan et al. (2025) noted that while RF performed well in predicting RUL, it required more computational resources compared to Gradient Boosting, which was identified as a viable alternative for resource-constrained environments.

In conclusion, as demonstrated by Qu et al. (2024) and Sultan et al. (2025), Random Forests offer a powerful and efficient method for predicting battery health. While they are highly accurate and robust against missing data, the model's lack of interpretability and potential limitations in continuous prediction precision pose challenges. Nevertheless, RF proves to be a promising tool for real-time battery health estimation, contributing significantly to predictive maintenance strategies.

### 2.3 Comparison of the Reviewed Systems

The Ampere-hour (Ah) Counting method provides a straightforward and cost-effective approach to battery monitoring. As shown in Table 2.1, it measures charge and discharge currents in real-time to estimate capacity, with moderate accuracy and low complexity. However, it lacks predictive capabilities, making it more suitable for short-term monitoring.

Electrochemical Impedance Spectroscopy (EIS) offers high accuracy and provides a deeper understanding of battery health by analyzing impedance across different frequencies. While Table 2.1 highlights its effectiveness in long-term health analysis, it requires specialized hardware and has high complexity. Like Ah Counting, it does not offer predictive capabilities.

On the other hand, Machine Learning (ML) models leverage historical data to predict battery health and lifespan. As indicated in Table 2.1, ML models provide high accuracy when trained on quality data. They require significant computational resources, large datasets, and complex model tuning. However, unlike the previous methods, ML enables predictive capabilities, making it ideal for long-term battery lifespan estimation and predictive maintenance.

*Table 2.1: Comparison of Battery Health Monitoring Techniques: Ampere-hour Counting, Electrochemical Impedance Spectroscopy (EIS), and Machine Learning.*

| **Comparison Factor** | **Ampere-hour (Ah) Counting (Direct Measurement)** | **Electrochemical Impedance Spectroscopy (EIS) (Direct Measurement)** | **Machine Learning (Data-Driven Model)** |
|---|---|---|---|
| **Technique Type** | Direct measurement | Direct measurement | Data-driven model |
| **Method** | Measures the charge/discharge current and integrates it over time to estimate capacity. | Measures the impedance of the battery at different frequencies to analyse its health. | Uses historical data to train models that predict battery health or performance. |
| **Real-Time Monitoring** | Yes | Yes | Depends on implementation |
| **Accuracy** | Moderate | High | High (depending on data and model) |
| **Hardware Requirements** | Current sensors, data loggers | Impedance analyser, specialized equipment | Computational resources, sensors for battery parameters |
| **Real-Time Feedback** | Yes | Yes | No (requires model inference) |
| **Data Storage** | Local or centralized storage for current data | Centralized data storage for impedance data | Centralized or cloud-based for training data |
| **Complexity** | Low | High | High (requires large datasets and model tuning) |
| **Predictive Capability** | No | No | Yes (after training phase) |
| **Cost** | Low | High (due to specialized equipment) | Moderate to High (depending on computational resources) |
| **Use Case** | Short-term monitoring of battery usage | Long-term health and degradation analysis | Predictive maintenance, long-term lifespan prediction |

Comparing model-based techniques, Table 2.2 outlines the key differences between Physics-based Models, Empirical Models, and Machine Learning (Random Forests). Physics-based Models rely on detailed electrochemical modeling, offering high accuracy when properly calibrated, but they come with high computational costs and are not suitable for real-time prediction.

Empirical Models, while moderately accurate, use statistical correlations derived from historical data. As noted in Table 2.2, they are simpler to implement than physics-based models and provide short-term predictions. However, their reliability depends on data quality, making them less effective for long-term forecasting.

Lastly, Random Forests, a machine learning technique, offer flexibility and adaptability, with high accuracy depending on data quality and feature selection. Table 2.2 shows that while this method is computationally demanding, it supports real-time predictions and lifespan forecasting, offering a balance between interpretability and complexity.

*Table 2.2: Comparison of Modelling Approaches for Battery Health Estimation and Lifetime Prediction: Physics-based Models, Empirical Models, and Machine Learning (Random Forests).*

| **Comparison Factor** | **Physics-based Models** | **Empirical Models** | **Random Forests (Common Machine Learning Techniques)** |
|---|---|---|---|
| **Technique Type** | Model-based | Data-driven | Machine learning |
| **Method** | Uses battery physics, such as electrochemical reactions, to model degradation and predict lifespan. | Relies on experimental data and statistical methods to establish correlations between battery usage and lifespan. | Uses ensemble learning from data, such as charge/discharge cycles, to predict lifespan. |
| **Real-Time Prediction** | No | No | Yes |
| **Accuracy** | High (if well-calibrated) | Moderate (depends on data quality) | High (depending on data and feature selection) |
| **Data Requirements** | Extensive experimental data and modelling parameters | Historical usage data | Large datasets with multiple features (e.g., temperature, charge cycles) |
| **Complexity** | High (requires detailed modelling) | Moderate (depends on statistical methods) | High (requires data preparation and model training) |
| **Hardware Requirements** | Computational resources for modelling | Data storage for historical data | Computational resources, sensor data for training |
| **Interpretability** | Low (complex models) | High (simple correlations) | Moderate (decision trees are interpretable but complex ensemble models are harder to explain) |
| **Cost** | High (requires extensive research and simulations) | Moderate | Moderate to High (depending on model complexity) |
| **Use Case** | Long-term, highly accurate lifespan prediction | Short-term predictions with lower accuracy | Flexible and adaptable predictions using machine learning techniques |

## 2.4 Direction of the Proposed System

Based on the techniques reviewed in Section 2.2, several challenges were identified in the existing methods for monitoring battery health and lifespan prediction. These challenges highlight areas where improvements could be made, which the proposed system intends to address.

The first technique, Ampere-hour (Ah) counting, is a direct measurement approach that is simple and widely used. However, it is limited by its inability to account for complex behaviours such as temperature variations, aging effects, and non-linear discharge patterns. This results in inaccuracies when predicting battery health and lifespan over time. Moreover, Ah counting requires frequent recalibration and can become inefficient for batteries that undergo frequent charge-discharge cycles, such as in electric vehicles or mobile devices. The proposed system will address these limitations by incorporating more advanced techniques that can dynamically adjust to changing environmental factors and battery conditions.

The second technique, Electrochemical Impedance Spectroscopy (EIS), is also a direct measurement method, offering high accuracy in estimating battery health by evaluating internal resistance and capacitance. However, EIS requires specialized equipment and is computationally intensive, which makes it less feasible for real-time applications or widespread consumer use. The proposed system will integrate data-driven models, such as machine learning techniques, to offer a more accessible and scalable solution. These models can process historical data and make accurate predictions without the need for expensive equipment or complex measurements.

Finally, Machine Learning (ML) techniques, particularly data-driven models, offer a promising approach for predicting battery lifespan based on historical data and real-

time monitoring. While these models have shown great potential, they require large datasets for training and may be susceptible to overfitting if not properly tuned. The proposed system will mitigate this by using robust cross-validation methods and incorporating a variety of features, including environmental factors, usage patterns, and battery-specific parameters, to ensure more accurate predictions.

In the realm of battery lifespan prediction, Physics-based models and Empirical models have been the cornerstone for estimating battery behaviour. Physics-based models are theoretically sound but require detailed information about the battery's construction and operating conditions, which are not always available. On the other hand, empirical models rely on historical data but may lack generalizability across different battery types and applications. The proposed system will leverage both approaches, combining the strengths of physics-based modelling with the flexibility and scalability of empirical models.

Lastly, techniques like Random Forests, a common machine learning algorithm, offer a promising way to predict battery lifespan by analysing complex patterns in data. However, these models can sometimes struggle with interpretability, making it difficult to understand the underlying reasons behind a specific prediction. The proposed system will address this challenge by implementing explainable AI techniques, ensuring that predictions are not only accurate but also understandable to users and stakeholders.

In summary, while the existing techniques for monitoring battery health and predicting lifespan offer valuable insights, they each have limitations in terms of scalability, accuracy, and feasibility for real-world applications. The proposed system will combine the strengths of direct measurement techniques, machine learning models, and hybrid approaches to overcome these challenges, providing a more efficient, scalable, and accurate solution for battery health and lifespan monitoring. By utilizing real-time data,

advanced algorithms, and accessible technology, the system will enhance the ability to predict and manage battery performance across a wide range of applications, from consumer electronics to electric vehicles.

## 2.5 Summary

This chapter reviewed various techniques for monitoring battery health and predicting battery lifespan, including Ampere-hour (Ah) counting, Electrochemical Impedance Spectroscopy (EIS), and Machine Learning models for monitoring, as well as Physics-based Models, Empirical Models, and Random Forests for lifespan prediction. A comparative analysis was conducted based on accuracy, scalability, feasibility, and computational complexity, highlighting key limitations such as limited accuracy, high computational costs, hardware dependency, and scalability challenges for real-time applications. To address these issues, the chapter proposed an integrated and scalable system that combines direct measurement techniques with machine learning models, offering a more accurate and accessible solution for battery health monitoring and lifespan prediction.

# CHAPTER 3: REQUIREMENT ANALYSIS AND SYSTEM DESIGN

## 3.1 Introduction

This chapter presents the project management methodology employed in the development of the Battery Health and Lifespan Prediction Diagnostic and Monitoring System. The methodology follows the Iterative Waterfall Model, which retains the structured phases of the traditional Waterfall approach but allows for iterative refinement at each stage. Instead of strictly completing one phase before proceeding to the next, feedback loops enable modifications and improvements throughout the development process. Each phase Requirement Analysis, System Design, Implementation, and Testing is revisited as needed based on insights gained from subsequent phases. This approach enhances flexibility while maintaining a systematic and well-documented development process. The chapter begins by outlining the requirement analysis phase, including data collection and system specifications. It then describes the system design phase, covering architecture, database schema, data flow diagrams, and user interface design. Finally, the implementation and testing phases are discussed, emphasizing iterative refinement and validation of the system.

## 3.2 Methodology

The development of the Battery Health and Lifespan Prediction Diagnostic and Monitoring System follows the Iterative Waterfall Model, a structured yet flexible methodology that allows for incremental improvements at each stage. While the model maintains a sequential approach progressing through Requirement Analysis, System Design, Implementation, and Testing it also incorporates iterative refinements, enabling modifications based on feedback from later phases.

The development process begins with Requirement Analysis and Data Collection, where system requirements are identified, essential metrics for battery health monitoring are defined, and relevant datasets are collected or simulated. The System Design phase follows, involving architectural planning, UI/UX design, and system diagrams that illustrate data flow and interactions.

Next, during the Implementation phase, the backend, predictive models, and frontend components are developed and integrated iteratively, ensuring adaptability based on early findings. The Testing phase uses an automated, scenario-based harness to confirm that the finished prototype meets its functional and predictive requirements, once all acceptance thresholds are satisfied, the results are documented and no further iteration is performed.

The structured yet iterative nature of this approach is visually represented in Figure 3.1, outlining the key phases and their iterative progression within the development lifecycle.

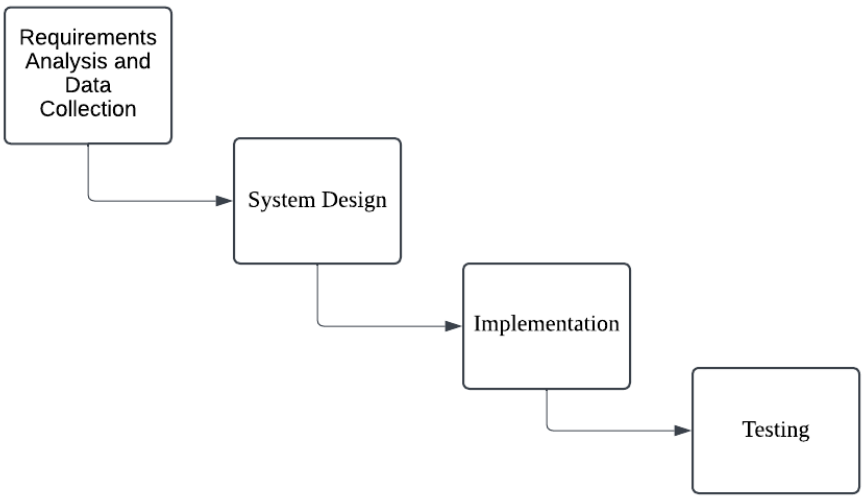



*Figure 3.1: Development Phases of the Battery Health and Lifespan Prediction Diagnostic and Monitoring System Using the Iterative Waterfall Model (ref. from GeeksforGeeks, 2024)*

### 3.2.1 Phase 1: Requirement Analysis and Data Collection

In the first phase of the project, we focus on understanding the requirements for the battery health monitoring system, particularly for lithium-ion (Li-ion) batteries. Lithium-ion batteries are commonly used in various applications, including energy storage systems and electric vehicles, due to their high energy density, long lifespan, and lower self-discharge rate. The data collected during this phase will form the foundation for designing the monitoring system.

**Battery Inverter Data Collection**

The data for this analysis was collected from a battery inverter system, which interfaces with the lithium-ion batteries to manage and monitor their operation. The system provides real-time data on several parameters that are critical for evaluating the battery's health and performance.

The data recorded for each inverter includes key performance indicators (KPIs) for both the Battery Inverter Extension 1 (E1) and Battery Inverter Master (M). These KPIs help to monitor the health of the battery, its efficiency, and its ability to deliver the required power. Below are the key parameters used for analysis:

- Battery Current (BATA): This parameter measures the current flowing through the battery. A high or fluctuating current may indicate issues with the battery's charging or discharging cycles.
- Battery Apparent Power (BATAP): This indicates the total power the battery system is capable of handling. It helps in understanding the energy capacity and potential performance under different load conditions.
- Battery Voltage (BATV): Monitoring the voltage helps identify whether the battery is operating within its optimal voltage range. Deviation from this range may indicate degradation or issues with the battery cells.
- Battery Temperature (BTEMP): Temperature is a critical factor in the performance of lithium-ion batteries. Elevated temperatures can accelerate the degradation process, while too low temperatures can affect battery efficiency.
- Cycle Count (CYCLE): The number of charge-discharge cycles the battery has undergone. This is a key factor in assessing the lifespan of the battery, as lithium-ion batteries typically degrade after a certain number of cycles.

- State of Charge (SOC): This indicates the current charge level of the battery. A consistent drop in SOC over time could signal issues with battery capacity.
- SOC Error (SOCERR): The discrepancy between the actual SOC and the measured SOC. This is important to detect any potential inaccuracies in the battery's state-of-charge measurement system.
- Condition Status: This provides information on the overall condition of the battery, helping to detect any anomalies in the battery's performance.

**Detailed Data Description**

The dataset consists of multiple entries recorded at specific timestamps. For example, the following data is recorded at 2023-01-01 00:05:

- P6AIBJandah33KW.BATTINV.E1.BATA: 33.275 A (Battery Inverter Extension 1, Battery Current)
- P6AIBJandah33KW.BATTINV.E1.BATAP: 33.275 VA (Battery Inverter Extension 1, Battery Apparent Power)
- P6AIBJandah33KW.BATTINV.E1.BATV: 50.04 V (Battery Inverter Extension 1, Battery Voltage)
- P6AIBJandah33KW.BATTINV.E1.BTEMP: 25.1 °C (Battery Inverter Extension 1, Battery Temperature)
- P6AIBJandah33KW.BATTINV.E1.SOC: 82% (Battery Inverter Extension 1, State of Charge)
- P6AIBJandah33KW.BATTINV.E1.SOCERR: 7.5% (Battery Inverter Extension 1, SOC Error)

The data includes several other parameters for both the Battery Inverter Extension 1 and Battery Inverter Master, offering a comprehensive view of the battery's performance across multiple measurement points. Figure 3.2 depicts the snapshot of dataset.

```
P6AIBJandah33KW.BATTINV.E1.BATA|2023-01-01 00:05|33275.000000
P6AIBJandah33KW.BATTINV.E1.BATAP|2023-01-01 00:05|33.275000
P6AIBJandah33KW.BATTINV.E1.BATV|2023-01-01 00:05|50.040000
P6AIBJandah33KW.BATTINV.E1.BTEMP|2023-01-01 00:05|25.100000
P6AIBJandah33KW.BATTINV.E1.Condition|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.E1.CYCLE|2023-01-01 00:05|52.000000
P6AIBJandah33KW.BATTINV.E1.FAC|2023-01-01 00:05|50.000000
P6AIBJandah33KW.BATTINV.E1.L|2023-01-01 00:05|240.270000
P6AIBJandah33KW.BATTINV.E1.L2|2023-01-01 00:05|239.970000
P6AIBJandah33KW.BATTINV.E1.L3|2023-01-01 00:05|239.760000
P6AIBJandah33KW.BATTINV.E1.PAC|2023-01-01 00:05|1583.000000
P6AIBJandah33KW.BATTINV.E1.PACP|2023-01-01 00:05|1583.000000
P6AIBJandah33KW.BATTINV.E1.Slave1|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.E1.Slave2|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.E1.SOC|2023-01-01 00:05|82.000000
P6AIBJandah33KW.BATTINV.E1.SOCERR|2023-01-01 00:05|7.500000
P6AIBJandah33KW.BATTINV.M.BATA|2023-01-01 00:05|26628.000000
P6AIBJandah33KW.BATTINV.M.BATAP|2023-01-01 00:05|26.628000
P6AIBJandah33KW.BATTINV.M.BATV|2023-01-01 00:05|50.400000
P6AIBJandah33KW.BATTINV.M.BTEMP|2023-01-01 00:05|27.200000
P6AIBJandah33KW.BATTINV.M.Condition|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.M.CYCLE|2023-01-01 00:05|40.000000
P6AIBJandah33KW.BATTINV.M.FAC|2023-01-01 00:05|50.000000
P6AIBJandah33KW.BATTINV.M.L|2023-01-01 00:05|239.990000
P6AIBJandah33KW.BATTINV.M.L2|2023-01-01 00:05|239.970000
P6AIBJandah33KW.BATTINV.M.L3|2023-01-01 00:05|239.980000
P6AIBJandah33KW.BATTINV.M.PAC|2023-01-01 00:05|1265.000000
P6AIBJandah33KW.BATTINV.M.PACP|2023-01-01 00:05|1265.000000
P6AIBJandah33KW.BATTINV.M.Slave1|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.M.Slave2|2023-01-01 00:05|307.000000
P6AIBJandah33KW.BATTINV.M.SOC|2023-01-01 00:05|82.000000
P6AIBJandah33KW.BATTINV.M.SOCERR|2023-01-01 00:05|2.200000
P6AIBJandah33KW.DPM.AAC|2023-01-01 00:05|4.859902
P6AIBJandah33KW.DPM.FAC|2023-01-01 00:05|49.999805
P6AIBJandah33KW.DPM.PAC|2023-01-01 00:05|2791.775703
P6AIBJandah33KW.DPM.PF|2023-01-01 00:05|0.797189
P6AIBJandah33KW.DPM.VAC|2023-01-01 00:05|415.980225
P6AIBJandah33KW.DPM.VL1N|2023-01-01 00:05|240.496292
P6AIBJandah33KW.DPM.VL2N|2023-01-01 00:05|240.183517
P6AIBJandah33KW.DPM.VL3N|2023-01-01 00:05|240.008682
P6AIBJandah33KW.PV.PV1.AADC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV1.AVDC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV1.BADC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV1.BVDC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV1.Condition|2023-01-01 00:05|307.000000
P6AIBJandah33KW.PV.PV1.GridRelay|2023-01-01 00:05|65533.000000
P6AIBJandah33KW.PV.PV1.PAC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV2.AADC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV2.AVDC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV2.BADC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV2.BVDC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV2.Condition|2023-01-01 00:05|307.000000
P6AIBJandah33KW.PV.PV2.GridRelay|2023-01-01 00:05|65533.000000
P6AIBJandah33KW.PV.PV2.PAC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV3.AADC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV3.AVDC|2023-01-01 00:05|0.000000
P6AIBJandah33KW.PV.PV3.BADC|2023-01-01 00:05|0.000000
```

*Figure 3.2: Snapshot of dataset*

**Parameters Specific to Lithium-Ion Batteries**

For the purpose of the battery health monitoring system, we specifically focus on the parameters relevant to lithium-ion batteries. These include:

- SOC and SOC Error: Lithium-ion batteries often suffer from inaccuracies in SOC estimation, especially as they age. Monitoring SOC and SOC Error allows for early detection of issues, such as incorrect charge levels or capacity degradation.
- Cycle Count: Lithium-ion batteries are typically rated for a specific number of charge-discharge cycles. A higher cycle count generally corresponds to a reduction in battery capacity, leading to shorter runtimes.
- Temperature: The thermal management of lithium-ion batteries is crucial for their longevity. Batteries that operate at higher temperatures tend to have a shorter lifespan, and thus, temperature monitoring is critical for predictive maintenance.
- Voltage: Lithium-ion batteries are sensitive to overvoltage and undervoltage conditions, which can lead to damage or reduced efficiency. Therefore, tracking battery voltage is a key factor in assessing battery health.

**Data Analysis for System Design**

The collected data provides a detailed snapshot of the battery's performance, which can be used to design the monitoring system. The monitoring system will leverage this data to:

- Evaluate Battery Health: By analysing the voltage, temperature, SOC, and cycle count, the system can assess the overall health of the lithium-ion battery and predict its remaining useful life.
- Identify Performance Degradation: A steady decline in voltage, an increase in temperature, or discrepancies in SOC can signal degradation or failure in the battery cells.
- Predict Failure Points: Using the data trends, the system can predict when the battery will likely fail or when it will require maintenance, helping to extend the battery's lifespan.

**Requirement Specifications for the Monitoring System**

Based on the data analysis, the following system requirements are identified for the battery health monitoring system:

- Real-Time Data Collection: The system must continuously collect data from the battery inverter parameters.
- Data Storage and Retrieval: The collected data needs to be stored in a database with easy access for analysis.
- Battery Health Prediction: The system should use the collected data to predict the health status of the lithium-ion batteries and estimate their remaining useful life.
- Alerting System: If critical thresholds such as temperature, voltage, or SOC are breached, the system should trigger alerts to inform users of potential battery issues.
- Reporting and Visualisation: The system should generate easy-to-understand reports and visualizations to help users interpret the battery's health and performance over time.

### 3.2.2 Phase 2: System Design

In this phase, the focus is on designing the architecture and components of the battery health monitoring system. Based on the data collected in Phase 1, the system is structured to effectively monitor, assess, and predict the health and lifespan of lithium-ion batteries. The design will include the system's hardware and software components, data flow, user interface, and integration with real-time data sources. The goal is to create a scalable, efficient, and user-friendly system capable of processing battery performance data, providing insights, and generating alerts for maintenance or replacement needs.

**System Architecture**

The Battery Health and Lifespan Prediction Diagnostic and Monitoring System is designed to track, monitor, and predict the health and lifespan of lithium-ion batteries. The architecture consists of multiple components working together seamlessly, including the hardware (sensors for battery parameters), a website for data visualization, and a prediction model for estimating battery lifespan. Below is a detailed description of the system architecture.

**Battery Monitoring System (BMS)**

The core component of the system involves monitoring the crucial parameters of the lithium-ion battery, such as voltage, current, temperature, and State of Charge (SOC). The BMS uses these parameters to evaluate the health of the battery in real-time and flag any potential issues (such as overcharging, deep discharge, or overheating).

Voltage Monitoring:

- The BMS continuously tracks the voltage levels across the battery cells to ensure they are within the safe operating range (typically 3.0V to 4.2V per cell for lithium-ion batteries). If the voltage falls below 3.0V or exceeds 4.2V, it triggers an alert.

Current Monitoring:

- A current sensor is used to measure the battery's charging and discharging current. Monitoring the current helps identify when the battery is being charged or discharged at an unsafe rate, potentially leading to capacity loss or overheating.

Temperature Monitoring:

- Temperature sensors are essential for tracking the thermal behavior of the battery. Lithium-ion batteries degrade faster at high temperatures, so the system will raise an alert if the temperature exceeds a predefined threshold which is 45°C.

State of Charge (SOC):

- The State of Charge (SOC) represents the current charge level of the battery as a percentage of its total capacity. This is calculated using the Coulomb Counting method, which tracks the flow of charge in and out of the battery over time. SOC is used to predict how much charge is left in the battery and helps in lifecycle estimation.

**Website for Data Visualisation**

A web-based interface allows users to visualize the real-time battery data and track its health. The website displays the following information:

- Real-time Voltage, Current, and Temperature: These values are updated continuously and presented in easy-to-read formats, such as graphs or gauges.
- State of Charge (SOC): The SOC is represented visually, showing the remaining charge as a percentage.
- Battery Health: Indicators such as "Healthy," "Warning," or "Critical" are displayed to inform users of any potential issues with the battery.
- Warning Alerts: If the battery reaches dangerous thresholds (e.g., overvoltage, undervoltage, overheating), a warning or error message is shown, providing users with actionable steps.

The website is built to be user-friendly and allows for both technical and non-technical users to interpret battery health data effectively. The website uses real-time data from the monitoring system, making it easy to track changes and trends over time.

**Battery Lifespan Prediction Model**

The system also includes a lifespan prediction feature, designed to estimate how long the battery will last before significant degradation occurs. This is done using advanced predictive models based on real-time data collected from the battery.

Cycle Counting:

- Lithium-ion batteries typically degrade with the number of charge/discharge cycles. The system counts each full cycle and uses this count to predict the remaining lifespan of the battery. A full cycle is counted when the battery discharges and recharges from 100% to 0%.

Arrhenius Model:

- Temperature has a direct impact on the battery's rate of degradation. The Arrhenius equation is used to model how temperature affects the lifespan of the lithium-ion battery. The Equation 3.1 according to Kucinskis et al. (2022) is as follows:

$$k = Ae^{\frac{-Ea}{RT}} \tag{3.1}$$

where:

- k is the rate constant for degradation.
- A is the pre-exponential factor (specific to the battery chemistry).
- Ea is the activation energy (in Joules per mole), which dictates how sensitive the battery is to temperature.
- R is the gas constant (8.314 J/mol·K).
- T is the absolute temperature in Kelvin.

Capacity Loss Prediction:

- The system tracks capacity loss over time and predicts when the battery will lose a significant portion of its original capacity. This is calculated using degradation models based on charge/discharge cycles and temperature impacts.

State of Health (SOH):

- The State of Health (SOH) is calculated by comparing the current capacity to the original capacity of the battery. The SOH will degrade over time, and the system uses this information, along with the battery's current voltage and temperature, to predict the remaining useful life of the battery.

**System Communication**

The BMS communicates with the web interface through a server. Data from the sensors is sent to the server, which then updates the website in real-time. The server may be hosted on a cloud platform to allow remote access and storage of historical data. The system uses standard web protocols to ensure secure and reliable communication between the battery monitoring hardware and the user interface.

**Prediction Formulae**

The system uses the following prediction formulas to estimate the battery's remaining lifespan:

1. Below is Equation 3.2, Cycle Life Prediction (Based on Charge/Discharge Cycles):

   Remaining Life (Cycles) = Total Life (Cycles) – Cycles Used (3.2)

   Where the total life in cycles is based on manufacturer specifications, and cycles used are tracked by the system.

2. Below is Equation 3.3, Capacity Degradation:

   $$C(t) = C_0 . e^{-at} \quad (3.3)$$

   Where:

   - C(t) is the battery capacity at time t.
   - $C_0$ is the initial capacity.
   - α is the degradation constant, based on temperature and usage patterns.
   - t is the time in years.

**Flow Chart of System**

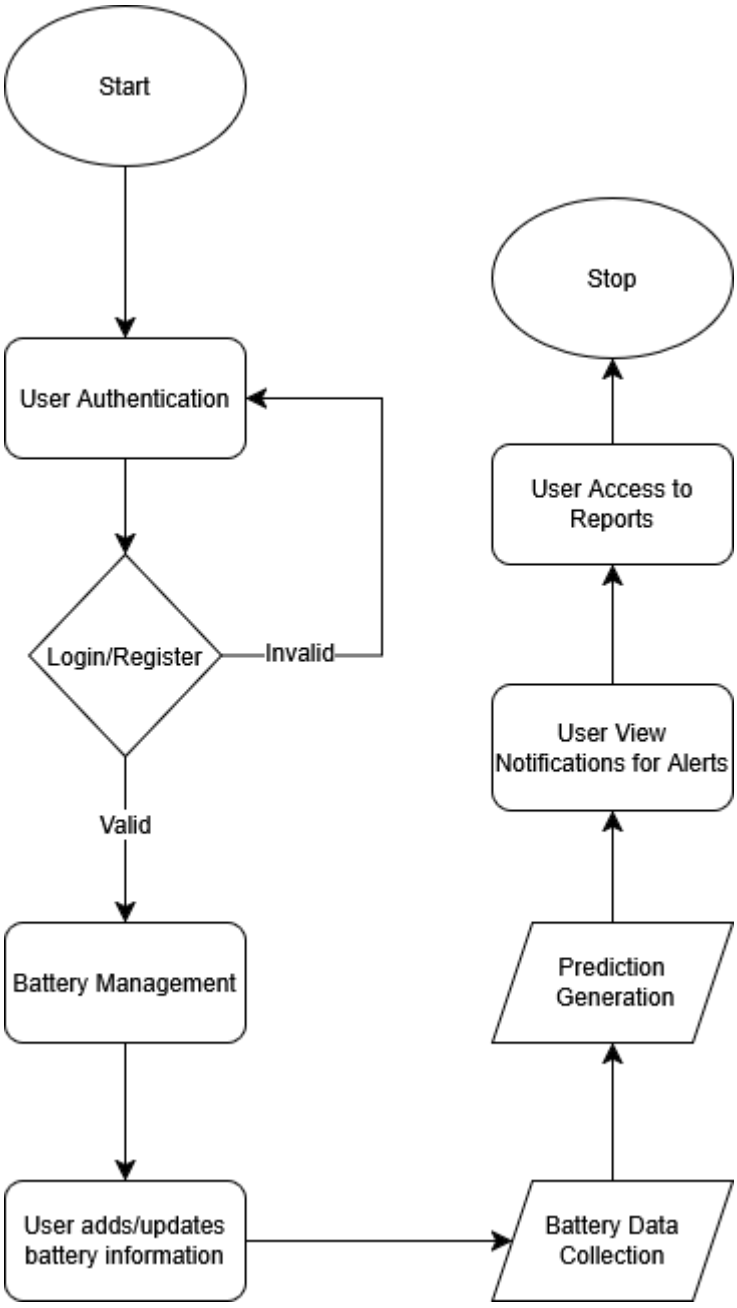



*Figure 3.3: Flow Chart of Battery Health and Lifespan Prediction*

Figure 3.3 illustrates the overall system workflow in a flowchart, providing a structured representation of how users interact with the Battery Health and Lifespan Prediction Diagnostic and Monitoring System. The process begins with the Start node, leading to User Authentication, where users must either log in or register an account. If authentication fails, the system redirects the user back to the login process. Once authenticated, the Battery Management phase allows users to add or update battery

information, which is then stored in the system. The next stage, Battery Data Collection, continuously retrieves and processes real-time battery metrics, such as voltage, current, and temperature, ensuring that the system maintains up-to-date records.

Following this, the Prediction Generation process analyses the collected battery data to estimate its lifespan and health status. Users are then notified of critical updates through the Notification Handling process, where alerts are generated if necessary. Additionally, users can access reports to review historical and predictive insights into their battery's performance. Finally, the process concludes at the Stop node, marking the completion of the system's workflow.

**Database Design**

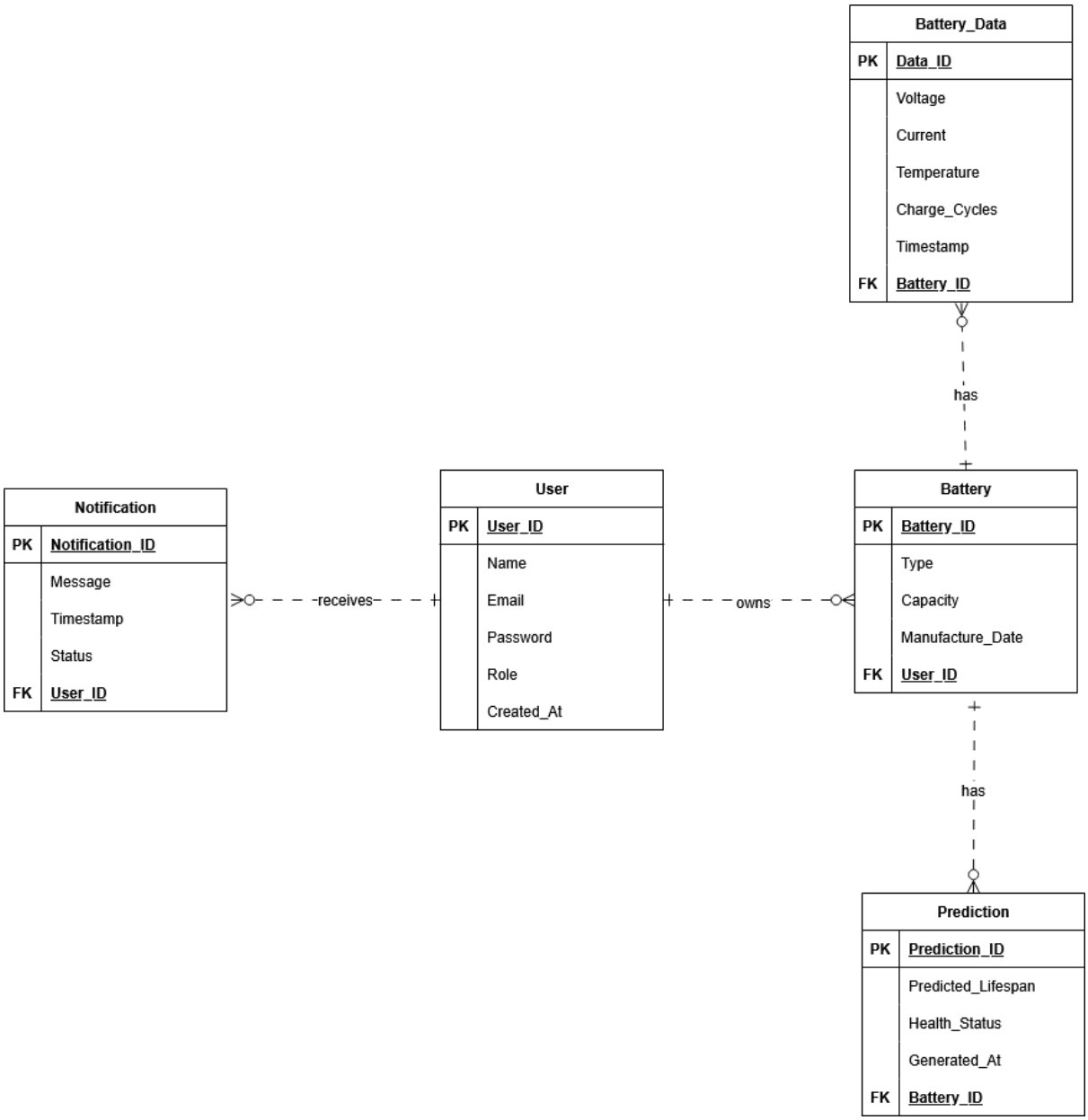



*Figure 3.4: Entity Relationship Diagram for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Entity-Relationship Diagram (ERD) in Figure 3.4 provides a structured representation of the system's database design, illustrating the relationships between key entities involved in monitoring and predicting battery health. This diagram ensures a well-organised and efficient database schema that supports seamless data storage, retrieval, and processing.

The User entity represents individuals interacting with the system, including administrators and end-users. Each user is uniquely identified by User_ID, and attributes such as Name, Email, and Role define their identity and permissions. Users own one or more Battery records, which store essential details such as Type, Capacity, and Manufacture_Date.

To monitor real-time battery conditions, the system maintains the Battery_Data entity, which captures key parameters such as Voltage, Current, Temperature, and Charge_Cycles. These records are crucial for assessing battery performance and predicting its lifespan. The Prediction entity leverages this data to forecast Predicted_Lifespan and determine the Health_Status of each battery.

Additionally, the Notification entity plays a vital role in alerting users about their battery status. Each notification is linked to a specific user and contains important messages regarding battery health, lifespan predictions, or system alerts.

The relationships between these entities establish a structured data flow, where:

- A User can own multiple Batteries.
- Each Battery generates multiple Battery_Data records over time.
- Predictions are derived from historical Battery_Data.
- Notifications are issued to users based on predictions and system events.

By defining these entities and relationships, the ERD in Figure 3.4 serves as the foundation for implementing a scalable and efficient battery monitoring and prediction system.

**Data Flow Diagram**

A Data Flow Diagram (DFD) visually represents the flow of information within a system. It uses symbols like rectangles, circles, and arrows to map out data inputs, outputs, storage points, and their interconnections. DFDs provide an efficient way to illustrate how data is processed, making it easier to understand system functions at various levels of detail. This approach is valuable for both analysing existing systems and modelling new ones, effectively communicating the data flow to both technical and non-technical audiences (What Is a Data Flow Diagram, n.d.).

**Context Diagram/Data Flow Diagram Level 0**

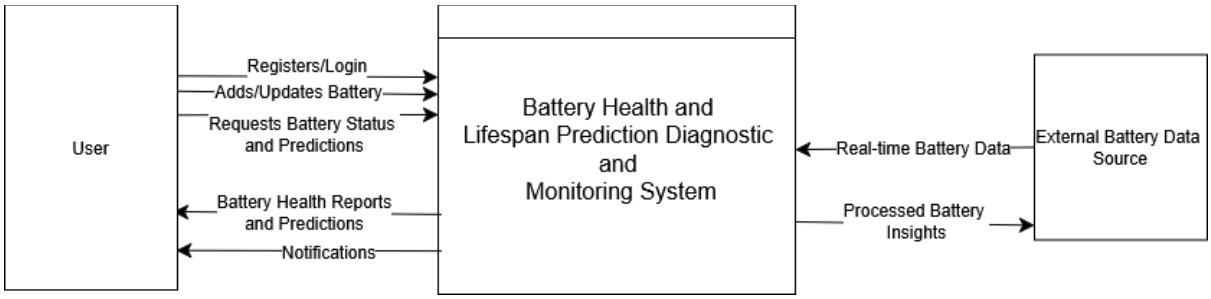



*Figure 3.5: Context Diagram for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The DFD Level 0 (Context Diagram), as shown in Figure 3.5, provides a high-level overview of the data flow within the Battery Health and Lifespan Prediction Diagnostic and Monitoring System. It illustrates the interactions between the system and external entities, ensuring that data is systematically processed and stored.

The User is a primary external entity that interacts with the system. Users can register and log in by providing their credentials, including name, email, and password. Once authenticated, they can add or update battery information, specifying details such as the battery type, capacity, and manufacture date. Additionally, users can request battery health status and lifespan predictions. In response, the system provides detailed

reports on battery condition, predictions on remaining lifespan, and necessary notifications regarding battery performance.

The External Battery Data Source supplies real-time battery data to the system. This data includes key parameters such as voltage, current, temperature, and charge cycles, which are essential for monitoring battery health and predicting its lifespan. The system processes and stores this data, integrating it into the prediction model to generate accurate assessments of battery performance.

Internally, the system manages several key processes to ensure accurate monitoring and prediction. It securely stores user data, maintains battery information, processes real-time battery metrics, generates lifespan predictions, and delivers timely notifications to users. The structured flow of data aligns with the ERD, ensuring consistency between database design and system operations.

**Data Flow Diagram Level 1**

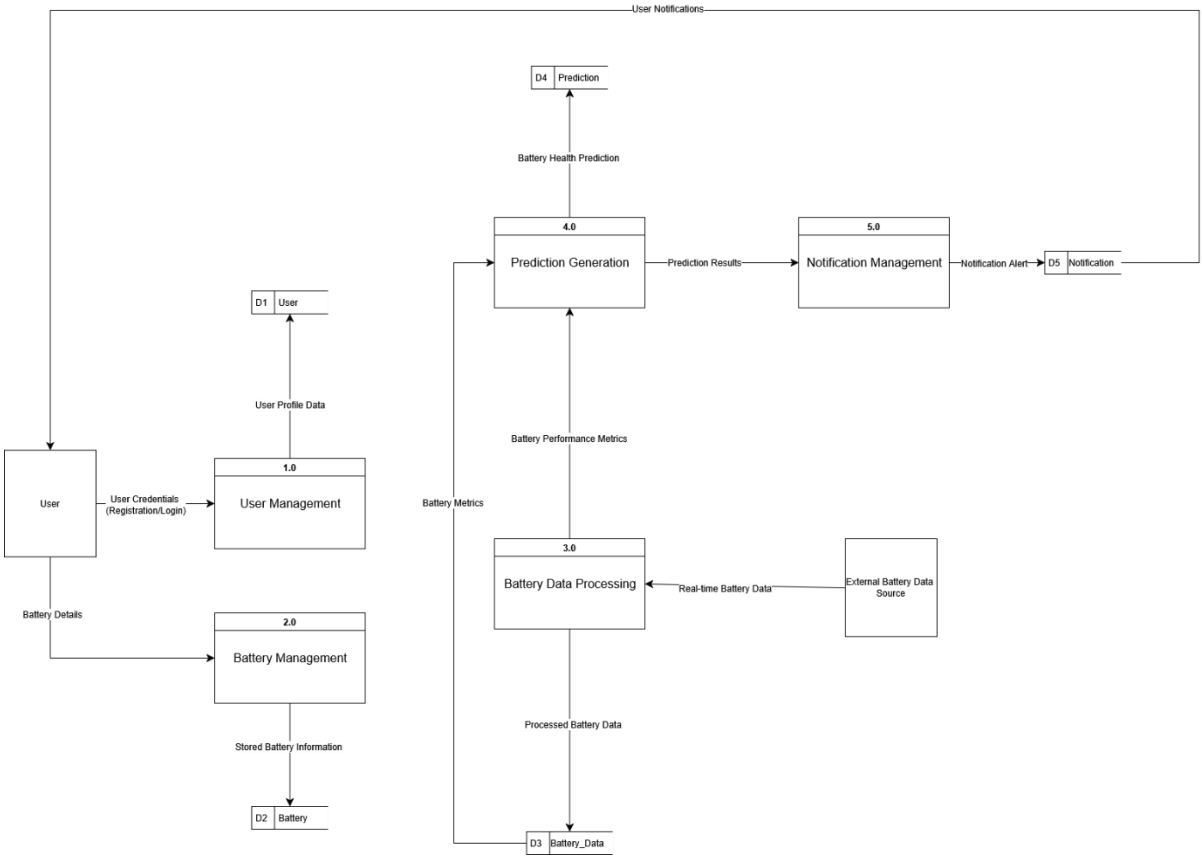



*Figure 3.6: Level 1 Data Flow Diagram for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

DFD Level 1 expands on the context diagram by detailing the internal processes and data flow within the Battery Health and Lifespan Prediction Diagnostic and Monitoring System.

The User interacts with the system through Process 1.0 (User Management), where credentials are verified and user information is stored in the User Data Store. Users can also manage battery details via Process 2.0 (Battery Management), which records battery specifications in the Battery Data Store.

Real-time battery data from an External Battery Data Source flows into Process 3.0 (Battery Data Processing), where metrics such as voltage, current, temperature, and charge cycles are analysed and stored in the Battery_Data Data Store. This processed

data is then used by Process 4.0 (Prediction Generation) to estimate battery lifespan and health status, with predictions stored in the Prediction Data Store.

Finally, Process 5.0 (Notification Management) retrieves prediction results and generates relevant alerts, storing them in the Notification Data Store before notifying the User. This ensures users receive timely updates regarding battery health and performance.

The breakdown in Figure 3.6 ensures alignment with the system's core objectives, providing structured data handling for battery monitoring and prediction.

**Data Flow Diagram Level 2 for Process 3.0**

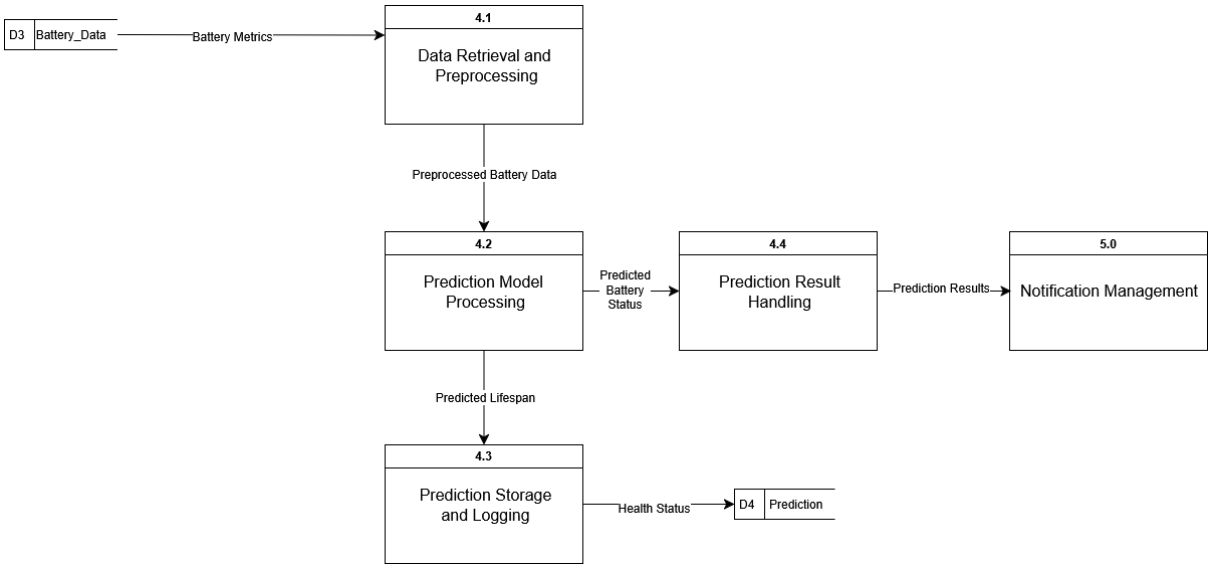



*Figure 3.7: Level 2 Data Flow Diagram for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

Figure 3.7 illustrates the DFD Level 2 for Process 4.0 (Prediction Generation), providing a detailed breakdown of how battery data is processed to generate lifespan and health predictions. This level further decomposes the prediction process into multiple sub-processes, ensuring an efficient and structured approach to forecasting battery health.

The process begins with Process 4.1 (Data Retrieval & Preprocessing), where real-time and historical battery metrics such as Voltage, Current, Temperature, Charge Cycles, and Timestamp—are retrieved from the Battery_Data Data Store. This raw data is cleaned and pre-processed to remove inconsistencies and ensure it is in a structured format for further analysis.

Next, the pre-processed data flows into Process 4.2 (Prediction Model Processing), where predictive algorithms are applied to assess battery degradation trends. The system analyses the cleaned data, estimating key parameters such as Predicted Lifespan and Health Status using machine learning models or statistical techniques.

Once the predictions are generated, they are stored through Process 4.3 (Prediction Storage & Logging), which records the computed results in the Prediction Data Store.

This ensures that past predictions can be retrieved for system monitoring, trend analysis, and user reference.

Simultaneously, the prediction results are passed to Process 4.4 (Prediction Result Handling), where they are evaluated to determine if user notifications should be triggered. If certain conditions such as a critically low battery health status or an imminent failure prediction are met, an alert is generated and forwarded to Process 5.0 (Notification Management). This ensures that users receive timely notifications about their battery's status, allowing them to take preventive actions as needed.

Through this structured sequence, Figure 3.7 effectively details the prediction generation workflow, demonstrating how the system processes raw battery data into meaningful insights that support proactive battery health monitoring and lifespan prediction.

**User Interface Design**

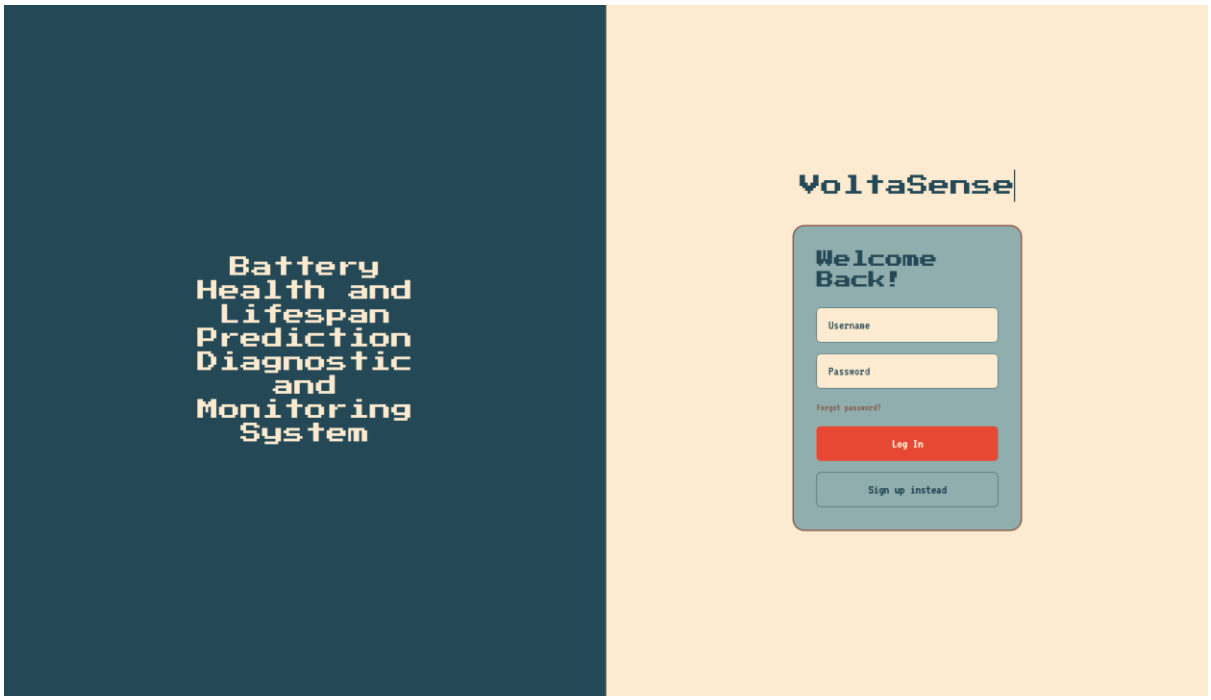



*Figure 3.8: Log in Page for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Login Page (refer to Figure 3.8) serves as the entry point for users to access the system. It is designed to be user-friendly, intuitive, and secure. The page typically includes fields for Username (or Email) and Password, both of which are essential for authenticating users. Below the login fields, there may be options such as Forgot Password for recovering account access and a Sign-Up link for new users to register. A clear Login button allows users to submit their credentials and proceed to the main dashboard.

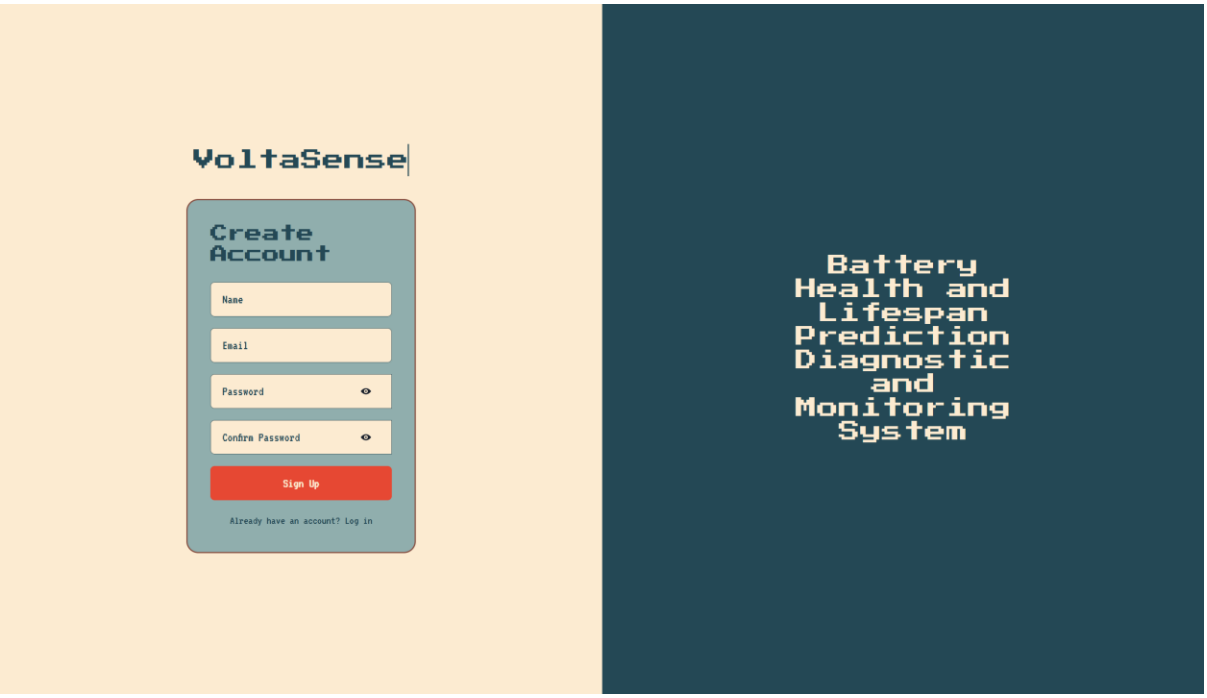



*Figure 3.9: Sign up Page for Battery Health and Lifespand Prediction Diagnostic and Monitoring System*

The Sign-Up page (refer to Figure 3.9) provides a straightforward registration interface for new users. The brand name *VoltaSense* is displayed prominently at the top, reinforcing the system identity. The form is centred in a card that contrasts with the background, guiding the user's focus. Four input fields are provided such as, Name, Email, Password and Confirm Password, ensuring that basic profile information and secure credentials are collected during account creation. The password fields employ masked input to protect sensitive data while typing. A red Sign Up button submits the form data for validation and account provisioning. Beneath the button, a text link invites existing users to switch to the login view, keeping navigation seamless within the authentication flow. The overall layout follows usability conventions by grouping related fields, maintaining clear labels and providing immediate visual feedback, thereby making the onboarding process both intuitive and secure.

The layout is simple and clean, ensuring that users can easily identify and interact with the login elements. Security features like password masking and validation checks for both fields are incorporated to protect user data. Additionally, the design is responsive, ensuring accessibility across various devices and screen sizes.

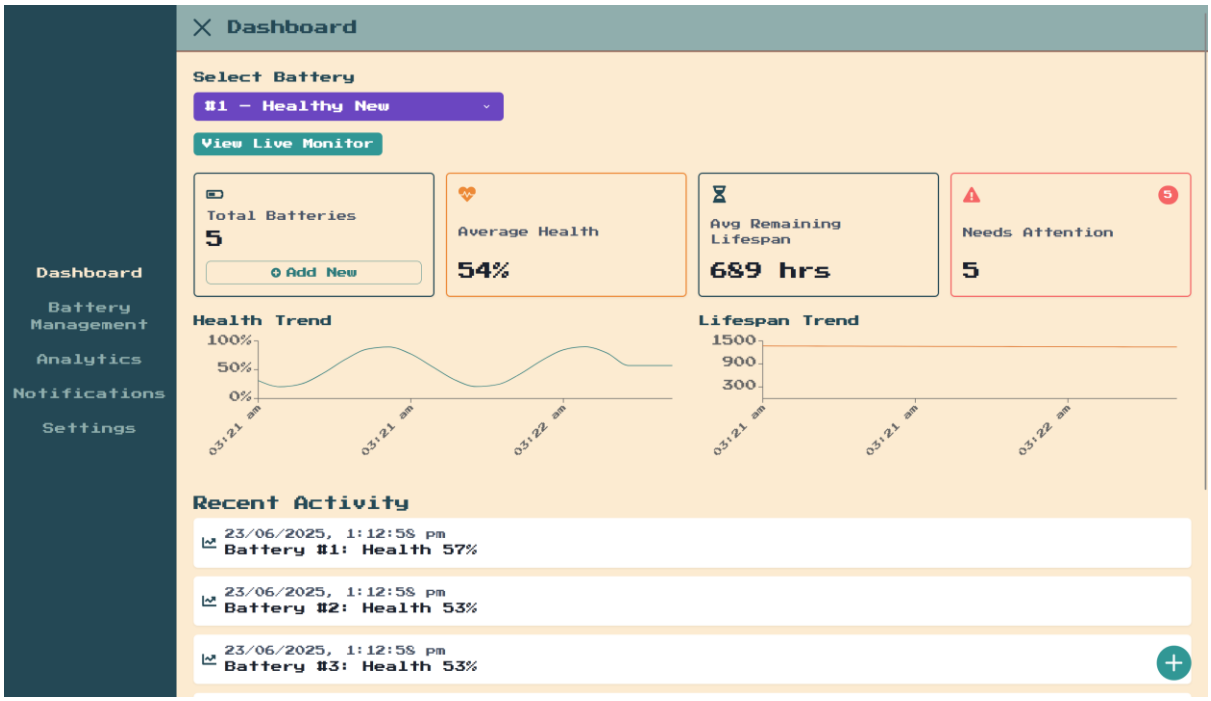



*Figure 3.10: Dashboard for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Dashboard (refer to Figure 3.10) serves as the system's main landing page, summarising fleet-level and individual-battery information in a single view. A Select Battery dropdown at the top allows the user to switch quickly between batteries; the View Live Monitor button opens a detailed, real-time panel for the chosen unit. Four summary cards follow: *Total Batteries* shows the fleet count with an inline *Add New* action, *Average Health* reports mean state-of-health as a percentage, *Avg Remaining Lifespan* converts that health into predicted service hours, and *Needs Attention* lists batteries currently flagged as Warning or Critical.

Two compact line charts track trends over time. *Health Trend* plots the average state-of-health for the selected battery, while *Lifespan Trend* charts the corresponding

remaining-life estimate, giving an at-a-glance indication of whether performance is stable or degrading. Below the charts, a *Recent Activity* feed logs timestamped health updates, making it easy to trace significant changes.

A left-hand navigation bar provides quick access to Dashboard, Battery Management, Analytics, Notifications and Settings, keeping all major sections within one click. The layout groups related data, uses clear typography and colour coding to highlight critical metrics, and maintains consistent spacing for readability, ensuring that users can assess overall battery status and drill into specifics without leaving the main screen.


*Figure 3.11: Battery Management for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Battery Management page (refer to Figure 3.11) provides a tabular view of every battery registered in the system and offers controls for bulk administration. A toolbar across the top includes buttons to Add Battery, Export CSV, Delete Selected, and Stop Simulation. A search bar lets users filter the list by ID or type.

The table itself displays key columns: ID, Type, Capacity, Last Health, Manufacture Date, Status, a miniature Sparkline that visualises recent health changes, and an Actions column with Edit and Delete buttons for each row. A master checkbox in the header and individual checkboxes in each row allow multi-select operations such as bulk deletion or export.

This layout enables quick auditing of fleet-wide health, immediate editing of metadata, and the ability to start or stop live-data simulations without navigating away from the management view.

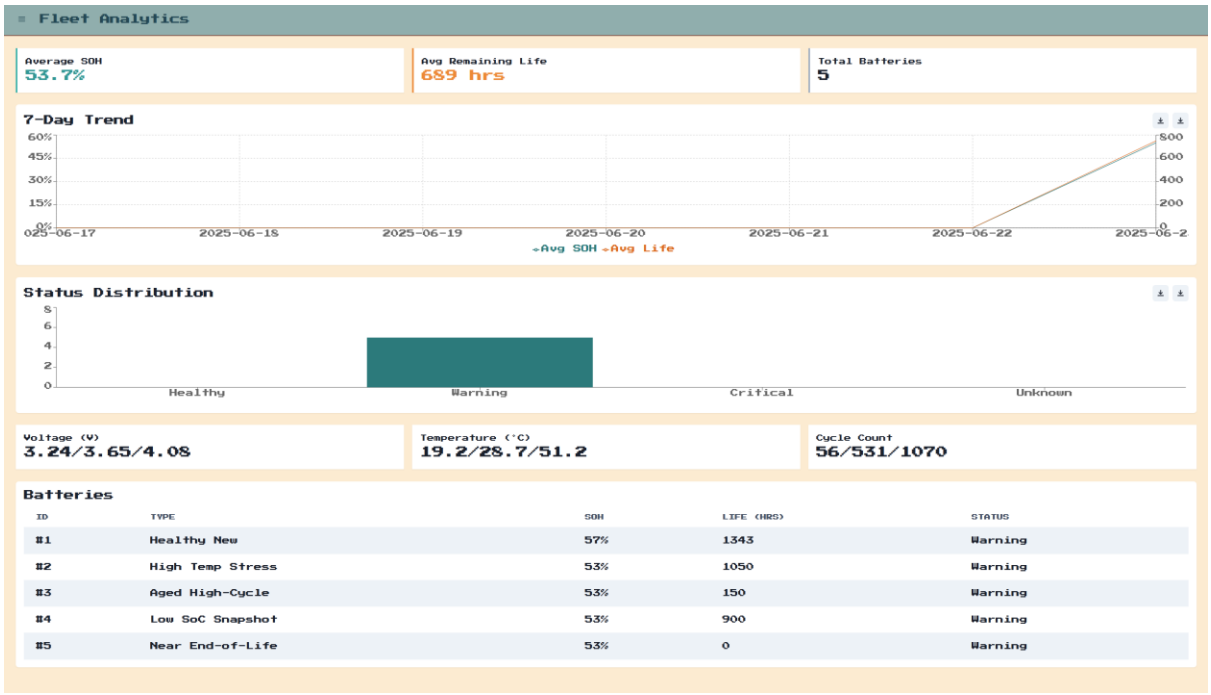



*Figure 3.12: Analytics for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Fleet Analytics page (refer to Figure 3.12) aggregates real-time health information for the entire battery fleet. A row of summary cards at the top highlights three headline metrics: Average SOH for the fleet, Avg Remaining Life in hours, and Total Batteries currently online. Directly beneath these cards a dual-axis line chart titled 7-Day Trend plots average state-of-health and average remaining life over the past week, giving a quick sense of whether the fleet is improving or degrading.

A Status Distribution bar chart follows, grouping batteries into Healthy, Warning, Critical and Unknown categories so that maintenance staff can see at a glance how many units need attention. Below this visual, three compact stat tiles report fleet-wide minima, averages and maxima for voltage, temperature and cycle count, helping operators verify that operating conditions stay within safe limits.

At the bottom of the view a Batteries table lists each unit by ID and type, alongside its current state-of-health, predicted life and status badge. Rows are colour-coded to match the distribution chart above, making problem batteries easy to spot. With these combined elements, the Fleet Analytics page offers a concise yet comprehensive overview of fleet performance and emerging issues in a single dashboard.


*Figure 3.13: Notification for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Notifications page (refer to Figure 3.13) lists every alert generated by the system so that operators can track events chronologically and identify batteries requiring

prompt action. A top-bar search field labelled Filter by Battery lets users narrow the table to a specific battery ID with a few keystrokes.

The table itself is sorted by timestamp, most recent first, and displays five columns: Time, Battery, Type, Message and Severity. "Time" records the exact moment an alert is issued, while "Battery" shows the affected unit's ID. "Type" indicates the class of event—for example *SoH Warning* or *SoH Critical*. The "Message" column provides a concise description, such as *SoH warning at 31 %*. "Severity" is colour-coded, with yellow *WARNING* tags and red *CRITICAL* tags that draw attention to the most urgent issues.

By combining filtering with colour emphasis and clear labelling, the Notifications page delivers a quick audit trail of system-detected anomalies, enabling maintenance teams to respond to health degradations before they lead to unexpected battery failures.

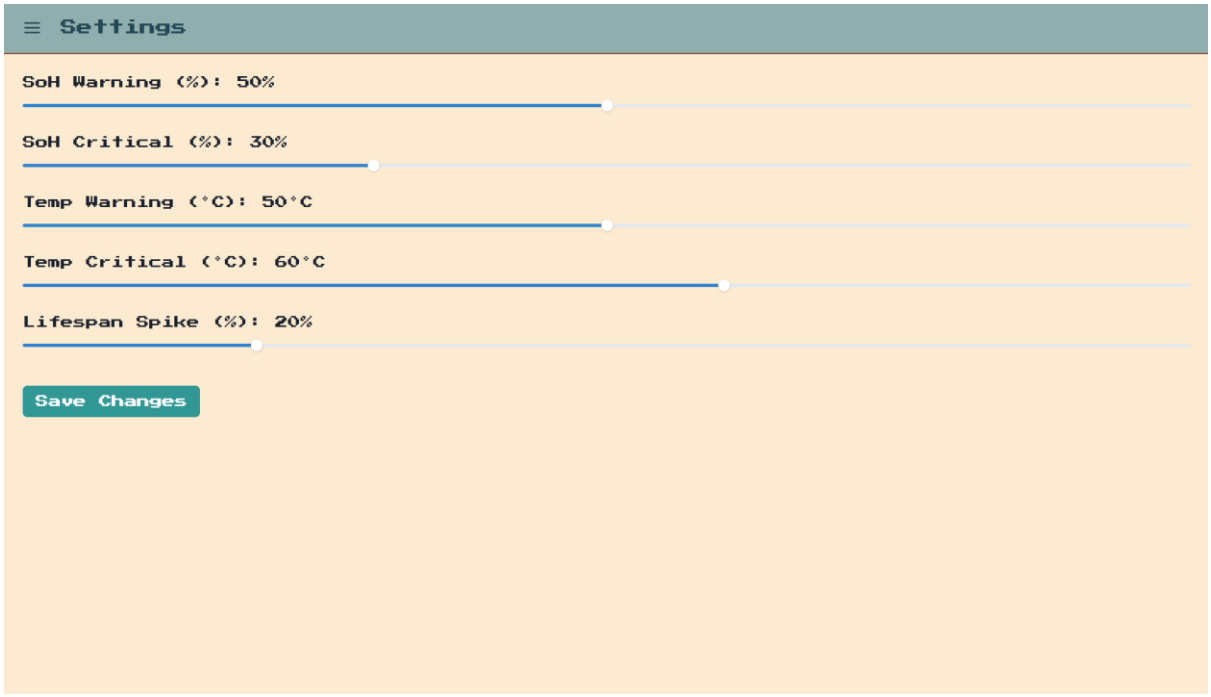



*Figure 3.14: Settings for Battery Health and Lifespan Prediction Diagnostic and Monitoring System*

The Settings page (refer to Figure 3.14) lets users tailor alert thresholds and prediction sensitivities to match their operational requirements. Four horizontal sliders appear in sequence. The first two set the SoH Warning and SoH Critical percentages,

defining when batteries change status colours on the dashboard. The next pair adjusts Temperature Warning and Temperature Critical limits in degrees Celsius, allowing the system to trigger notifications when cells overheat. A fifth slider labelled Lifespan Spike (%) governs how large a sudden jump in remaining-life estimation must be before the system flags it for review, helping to filter out noise from genuine anomalies.

Each slider displays its current value inline for immediate feedback, and changes are persisted with a single Save Changes button at the bottom of the page. This simple layout enables quick, precise control over key parameters while keeping the interface uncluttered and easy to understand.

### 3.2.3 Phase 3: Implementation

In Phase 3, the Battery Health and Lifespan Prediction Diagnostic and Monitoring System will be developed and deployed. This phase will involve coding the user interface, implementing backend processes for data collection, processing, and prediction generation, and integrating the necessary predictive models. The system will be tested for functionality, performance, and security during this phase. Once the front-end and back-end components exchange data reliably, the integrated system is smoke-tested for basic functionality and local security. Implementation tasks and technical details are presented in Chapter 4.

### 3.2.4 Phase 4: Testing

Phase 4 verifies that the completed prototype meets its functional and predictive requirements. An automated harness drives five scripted scenarios, captures one API response per second and compares each prediction with reference values. Acceptance checks cover prediction error, status-label correctness and API latency. All results are summarised in validation tables and analysed to confirm that the system performs within predefined thresholds. The full testing procedure and outcomes are documented in Chapter 5.

### 3.3 Summary

This chapter has outlined the development approach for the Battery-Health Monitoring and Lifespan-Prediction System. An iterative waterfall model guides the work, progressing through four structured phases. Phase 1 defines requirements and identifies the key metrics like voltage, temperature, state-of-charge and cycle count, while assembling simulated or real telemetry for initial trials. Phase 2 translates those requirements into a detailed system design, specifying data flow, component responsibilities and user-interface layouts. Phase 3 covers implementation, where the FastAPI back end, the prediction module and the React front end are coded and integrated into a working prototype. Phase 4 validates the finished system with an automated, scenario-based test harness to confirm functional correctness, prediction accuracy and performance targets. The technical details of implementation and the full testing results will be presented in Chapters 4 and 5 respectively.

# CHAPTER 4: IMPLEMENTATION

Chapter 4 presents the technical realization of the Battery Health and Lifespan Prediction Diagnostic and Monitoring System by translating the requirement analysis and design framework from Chapter 3 into executable software. Section 4.1 details the development environment and toolchain configuration for the front-end (React, Chakra UI, Recharts) and the back-end (FastAPI, SQLModel with SQLite, Python-based prediction module). Section 4.2 illustrates the implementation of each user interface: Dashboard, Live View Monitor, Battery Management, Analytics, Notifications and Settings, and Authentication, with annotated code excerpts that explain core logic. Section 4.3 describes the back-end architecture, including API endpoint implementation, data model definitions, prediction engine integration, and the alerting mechanism. Section 4.4 examines client-server interaction patterns to demonstrate full end-to-end functionality.

### 4.1 Development Environment Setup

A reproducible development environment was defined for both front-end and back-end components to ensure consistent builds and reliable integration.

#### 4.1.1 Front-end Environment

The front-end application was built using Node.js (v22.15.1) and npm to manage dependencies and scripts. The project was initialized with the Vite React template, and the exact commands for creating the repository, installing libraries and launching the local server are shown in Figure 4.1. The first command is `npm create vite@latest frontend -- --template react`, then we will proceed to change directory to the created folder using `cd frontend`. Once in the new folder, we can run `npm install` to install all required packages, following with `npm run dev` which will start the local development server. After the initial scaffold, React, React DOM, Chakra UI, Recharts and the Socket.IO client were added via npm to support component-based rendering, theming, interactive charts and real-time data updates. Once dependencies were installed, the development server was started with a simple npm script, and the resulting folder layout with the public directory at the root and the src directory housing App.jsx, main.jsx and the components subfolder is depicted in Figure 4.2.

```
C:\Users\Veshal\Desktop\SampleFYP>npm create vite@latest frontend -- --template react

> npx
> create-vite frontend --template react

|
o  Scaffolding project in C:\Users\Veshal\Desktop\SampleFYP\frontend...
|
—  Done. Now run:

  cd frontend
  npm install
  npm run dev


C:\Users\Veshal\Desktop\SampleFYP>cd frontend

C:\Users\Veshal\Desktop\SampleFYP\frontend>npm install

added 154 packages, and audited 155 packages in 9s

33 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\Veshal\Desktop\SampleFYP\frontend>npm run dev
```

*Figure 4.1: Commands for Setting Up The Frontend*


*Figure 4.2: Frontend Folder Organisation*

### 4.1.2 Back-end Environment

The back-end code resides in a manually created folder named **backend**. Within this folder a Python 3.10 or newer virtual environment was created by running `python -m venv venv` and on Windows the environment was activated with `.\venv\Scripts\activate`. All project dependencies were then installed in one step by executing `pip install -r requirements.txt` as shown in Figure 4.3. This command installs FastAPI for the REST API framework, Uvicorn to serve the application, SQLModel for object-relational mapping on SQLite, 3.6-dotenv to load configuration from a .env file, and sse-starlette for server-sent events support, along with auxiliary libraries such as Pydantic, AnyIO, Click, Colorama and Requests. If a minimal installation is preferred, FastAPI and its server can be added with `pip install fastapi uvicorn`. The database connection string (sqlite:///./database.db) and other configuration values are defined in the .env file at the project root. The resulting directory layout, which includes main.py, db.py, models.py and predict.py, is illustrated in Figure 4.4.

```
C:\Users\Veshal\Desktop\SampleFYP\backend>python -m venv venv

C:\Users\Veshal\Desktop\SampleFYP\backend>.\venv\Scripts\activate

(venv) C:\Users\Veshal\Desktop\SampleFYP\backend>pip install -r requirements.txt
Collecting annotated-types==0.7.0 (from -r requirements.txt (line 1))
  Using cached annotated_types-0.7.0-py3-none-any.whl.metadata (15 kB)
Collecting anyio==4.9.0 (from -r requirements.txt (line 2))
```

*Figure 4.3: Commands for Setting Up the Backend*

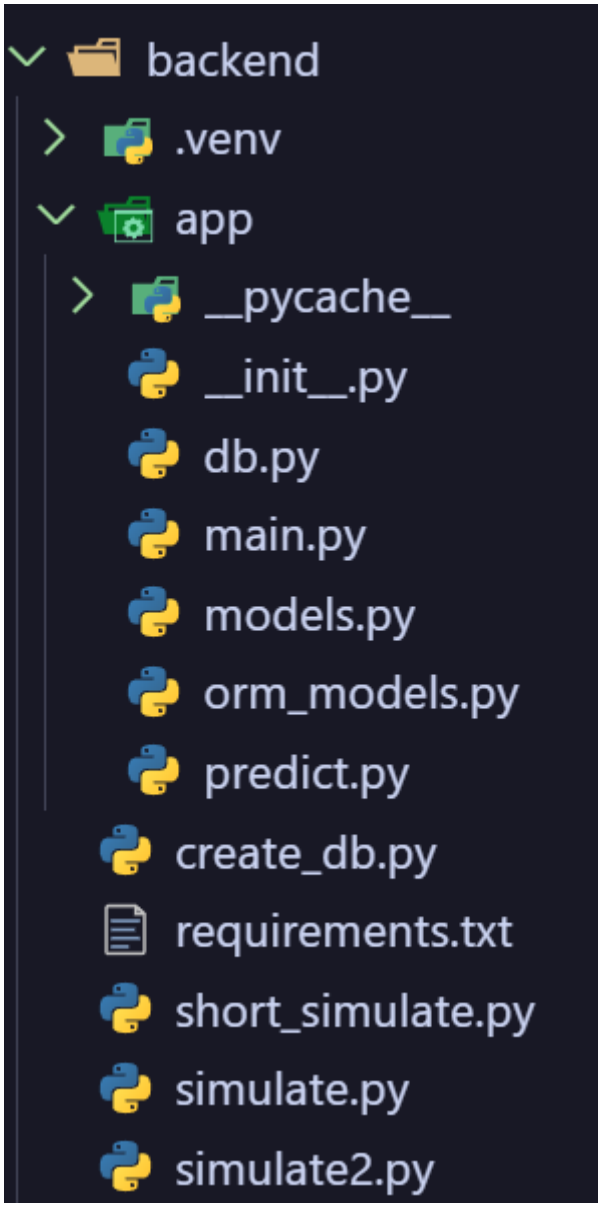



*Figure 4.4: Backend Folder Organisation*

### 4.2 Frontend Implementation

The frontend was built with React to provide a responsive, component-based interface for monitoring and managing battery data in real-time. Key libraries include Chakra UI for consistent styling and layout, Recharts for interactive data visualizations, and the Socket.IO client for live updates. Each page is encapsulated in its own component under the `src/components` directory, enabling modular development, easy testing, and straightforward maintenance.

#### 4.2.1 Dashboard

The Dashboard component delivers an at-a-glance view of key battery metrics by polling a summary endpoint every ten seconds and rendering the results in a structured layout. Figure 4.5 shows the core polling logic in the IDE:

*Figure 4.5: Code Snippet From Dashboard.jsx Showing The useEffect Polling Logic*

This hook invokes fetchSummary immediately on mount and then every ten seconds, ensuring that the stats state object remains up to date.

Figure 4.6 presents the JSX that renders four summary cards, two trend charts, a recent activity section and navigation cards:

```jsx
{/* Stats Cards */}
<SimpleGrid columns={[1, 2, 4]} spacing={4} mb={6}>
    <StatsCard title="Total Batteries" value={batteryCount} icon={FaBatteryHalf} accentColor=
"darkTeal" showAddButton onAddClick={onOpen} />
    <StatsCard title="Average Health" value={`${Math.round(avgHealth * 100)}%`} icon={FaHeartbeat
} accentColor={avgHealth >= 0.7 ? 'green.400' : 'orange.400'} />
    <StatsCard title="Avg Remaining Lifespan" value={`${Math.round(avgLifespan)} hrs`} icon={
FaHourglassHalf} accentColor="darkTeal" />
    <StatsCard title="Needs Attention" value={batteriesNeeding.length} icon={
FaExclamationTriangle} accentColor="red.400" alertBadge={{ text: `${batteriesNeeding.length}`,
 color: 'red.400' }} />
</SimpleGrid>

{/* Trend Charts */}
{selectedBatteryId && (
    <SimpleGrid columns={[1, 2]} spacing={4} mb={6}>
        <Box>
            <Heading size="sm" mb={2} color="darkTeal">Health Trend</Heading>
            {loadingTelemetry ? <Spinner color="darkTeal" /> : <HealthTrendChart data={
telemetryData} />}
        </Box>
        <Box>
            <Heading size="sm" mb={2} color="darkTeal">Lifespan Trend</Heading>
            {loadingTelemetry ? <Spinner color="darkTeal" /> : <LifespanTrendChart data={
telemetryData} />}
        </Box>
    </SimpleGrid>
)}

{/* Recent Activity */}
<Box mb={6}>
    <Heading size="md" mb={2} color="darkTeal">Recent Activity</Heading>
    <RecentActivityList items={recentActivity} />
</Box>

{/* Navigation Cards */}
<SimpleGrid columns={[1, 2]} spacing={4}>
    <NavCard label="Battery Management" icon={FaBatteryHalf} onClick={() => navigate(
'/battery-management')} />
    <NavCard label="Analytics" icon={FaChartLine} onClick={() => navigate('/analytics')} />
    <NavCard label="Notifications" icon={FaBell} onClick={() => navigate('/notifications')} />
    <NavCard label="Settings" icon={FaCog} onClick={() => navigate('/settings')} />
</SimpleGrid>
```

*Figure 4.6: Code Snippet From Dashboard.jsx Showing the Summary Card and Layout Structure*

This layout leverages Chakra UI's SimpleGrid for responsive rows and columns while custom StatsCard, HealthTrendChart, LifespanTrendChart, RecentActivityList and NavCard components encapsulate styling and behaviour.

Figure 4.7 shows the live Dashboard in the browser. At the top a battery selector and "View Live Monitor" button allow context switching. The summary cards display total battery count, average health, estimated remaining lifespan and count of batteries requiring attention. Two line charts plot health and lifespan trends over the latest readings. The recent activity list logs timestamped battery updates. Finally, navigation cards link to Battery Management, Analytics, Notifications and Settings. This arrangement presents a comprehensive real-time overview of system status in a single view.


*Figure 4.7: Screenshot of The Rendered Dashboard User Interface*

### 4.2.2 Live View Monitor

The Live View Monitor component offers a continuously updating display of detailed battery telemetry. The implementation can be broken into three parts: initial data fetch, polling for new data, and rendering the live readings grid.

Figure 4.8 shows the first `useEffect` hook. On mount it invokes `fetchBattery` to load battery metadata and `fetchSeries` to retrieve the most recent 100 telemetry points. The telemetry array is mapped into objects with `timestamp`, `soh` and `remaining_lifespan_hours` fields. A ten-second interval is established to refresh the series automatically.

```
// 1) Fetch metadata + initial series
useEffect(() => {
    let seriesInterval

    async function fetchBattery() {
        try {
            const resp = await fetch(`http://127.0.0.1:8000/api/batteries`)
            const all = await resp.json()
            setBatteryInfo(all.find(b => b.id === Number(id)) || null)
        } catch (e) {
            console.error(e)
        }
    }

    async function fetchSeries() {
        setLoading(true)
        try {
            const resp = await fetch(
                `http://127.0.0.1:8000/api/battery/${id}/telemetry?limit=100`
            )
            const data = await resp.json()
            setTelemetryData(
                data.map(r => ({
                    timestamp: r.timestamp,
                    soh: r.soh,
                    remaining_lifespan_hours: r.remaining_lifespan_hours,
                }))
            )
        } catch (e) {
            console.error(e)
            setTelemetryData([])
        } finally {
            setLoading(false)
        }
    }

    fetchBattery()
    fetchSeries()
    seriesInterval = setInterval(fetchSeries, 10000)
    return () => clearInterval(seriesInterval)
}, [id])
```

*Figure 4.8: Code Snippet from BatteryDetail.jsx showing the initial metadata and series fetch `useEffect` hook*

Figure 4.9 presents the second `useEffect` hook. It defines `fetchLatest` to request only the single latest telemetry record, then appends it to the existing state while trimming the array to the most recent 20 entries. This hook runs immediately on mount and then every five seconds, ensuring the displayed data remains current without refetching the entire series.

```
// 2) Poll newest point
useEffect(() => {
    let latestInterval
    async function fetchLatest() {
        try {
            const resp = await fetch(
                `http://127.0.0.1:8000/api/battery/${id}/telemetry?limit=1`
            )
            const [point] = await resp.json()
            setLatest(point)
            setTelemetryData(prev => {
                const next = [
                    ...prev,
                    {
                        timestamp: point.timestamp,
                        soh: point.soh,
                        remaining_lifespan_hours: point.remaining_lifespan_hours,
                    },
                ]
                return next.length > 20 ? next.slice(-20) : next
            })
        } catch (e) {
            console.error(e)
        }
    }
    fetchLatest()
    latestInterval = setInterval(fetchLatest, 5000)
    return () => clearInterval(latestInterval)
}, [id])
```

*Figure 4.9: Code Snippet from BatteryDetail.jsx showing the polling newest telemetry `useEffect` hook*

Figure 4.10 contains the JSX that displays the live readings once latest is available. A SimpleGrid arranges seven Stat components showing voltage, current, temperature, state of charge, cycle count, state of health and estimated remaining lifespan. Numeric values are formatted with toFixed or Math.round to present two-decimal or integer precision as appropriate.

```jsx
{/* Live Readings */}
<Box mb={8}>
    <Heading fontSize="lg" color="darkTeal" mb={3}>
        Live Readings
    </Heading>
    {latest ? (
        <SimpleGrid columns={{ base: 2, md: 4 }} spacing={4}>
            <Stat>
                <StatLabel>Voltage</StatLabel>
                <StatNumber>{latest.batt_voltage.toFixed(2)} V</StatNumber>
            </Stat>
            <Stat>
                <StatLabel>Current</StatLabel>
                <StatNumber>{latest.batt_current.toFixed(2)} A</StatNumber>
            </Stat>
            <Stat>
                <StatLabel>Temperature</StatLabel>
                <StatNumber>{latest.batt_temp.toFixed(2)} °C</StatNumber>
            </Stat>
            <Stat>
                <StatLabel>SoC</StatLabel>
                <StatNumber>
                    {(latest.soc * 100).toFixed(2)} %
                </StatNumber>
            </Stat>
            <Stat>
                <StatLabel>Cycle Count</StatLabel>
                <StatNumber>{latest.cycle_count}</StatNumber>
            </Stat>
            <Stat>
                <StatLabel>SoH</StatLabel>
                <StatNumber>{(latest.soh * 100).toFixed(1)}%</StatNumber>
            </Stat>
            <Stat>
                <StatLabel>Lifespan</StatLabel>
                <StatNumber>
                    {Math.round(latest.remaining_lifespan_hours)} hrs
                </StatNumber>
            </Stat>
        </SimpleGrid>
    ) : (
        <Flex align="center" justify="center" minH="100px">
            <Spinner color="darkTeal" />
        </Flex>
    )}
</Box>
```

*Figure 4.10: Code Snippet from BatteryDetail.jsx showing the Live Readings grid layout*

Figure 4.11 illustrates the component in the browser. At the top the battery identifier, type and manufacture date are shown. Below, the Live Readings grid presents real-time values for voltage (4.06 V), current (1.20 A), temperature (42.20 °C), SoC (88.29 %), cycle count and SoH (88.3 %). Two line charts plot health trend and lifespan

trend over the last 20 data points, and the header includes a back button for navigation. This layout provides an immediate, detailed view of a single battery's real-time status.

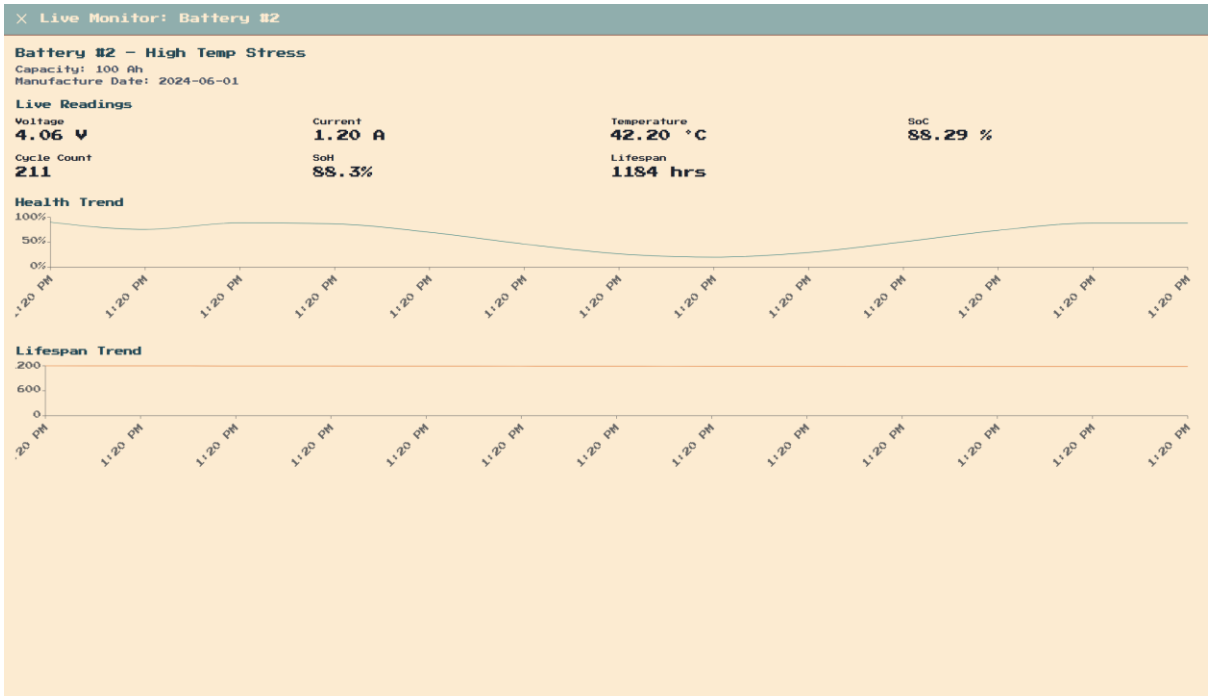



*Figure 4.11: Screenshot of the rendered Live View Monitor user interface*

### 4.2.3 Battery Management Page

The Battery Management component provides a centralized interface for adding, searching, exporting, deleting and simulating battery data. Figure 4.12 shows the code that defines the control toolbar. An `<HStack>` arranges five buttons and an input field: “Add Battery,” “Export CSV,” “Delete Selected,” “Run/Stop Simulation” and a search box bound to the `search` state. Each button is decorated with a left icon, a colour scheme and an `onClick` handler. The simulation button toggles its label and colour based on the `running` state, and the delete button is disabled unless at least one row is selected.

```jsx
<HStack spacing={3} mb={4}>
    <Button
        leftIcon={<AddIcon />}
        bg="darkTeal"
        color="white"
        _hover={{ bg: 'coral.600' }}
        boxShadow="md"
        onClick={openAdd}
    >
        Add Battery
    </Button>
    <InputGroup maxW="300px">
        <InputLeftElement pointerEvents="none">
            <SearchIcon color="gray.500" />
        </InputLeftElement>
        <Input
            placeholder="Search by ID or type…"
            value={search}
            onChange={e => setSearch(e.target.value)}
            bg="white"
        />
    </InputGroup>
    <Button
        leftIcon={<FaFileCsv />}
        variant="outline"
        colorScheme="darkTeal"
        onClick={handleExport}
    >
        Export CSV
    </Button>
    <Button
        leftIcon={<FaTrash />}
        colorScheme="red"
        onClick={handleBulkDelete}
        isDisabled={!selectedIds.length}
    >
        Delete Selected
    </Button>
    <Button
        leftIcon={
            running ? <CloseIcon /> : <RepeatIcon />
        }
        colorScheme={running ? 'red' : 'purple'}
        onClick={runClientSimulation}
    >
        {running
            ? 'Stop Simulation'
            : 'Run Simulation'}
    </Button>
</HStack>
```

*Figure 4.12: Code Snippet from BatteryManagement.jsx showing the action toolbar and search input*

Figure 4.13 presents the table markup. When `loading` is true a Chakra UI `<Spinner>` is rendered. Otherwise a `<TableContainer>` wraps a striped `<Table>` with a header row that includes a master checkbox for selecting all entries and sortable column headings for ID, type, capacity and last health. The `<Tbody>` maps over the `displayed` array to produce `<Tr>` elements. Each row begins with a checkbox that toggles selection, followed by cells for battery ID, type, capacity, health percentage, manufacture date and status (rendered as a `<Badge>` with a dynamic colour). A `<Sparkline>` component renders a mini trend chart, and “Edit” and “Del” buttons invoke row-specific handlers.

```
{/* Table */}
{loading ? (
    <Spinner size="lg" color="darkTeal" />
) : (
    <TableContainer
        borderRadius="md"
        boxShadow="sm"
        bg="white"
    >
        <Table variant="striped" colorScheme="gray">
            <Thead bg={headerBg}>
                <Tr>
                    <Th>
                        <Checkbox
                            isChecked={allSelected}
                            onChange={toggleSelectAll}
                        />
                    </Th>
                    {[
                        'id',
                        'type',
                        'capacity',
                        'lastHealth',
                    ].map(f => (
                        <Th
                            key={f}
                            cursor="pointer"
                            color={headerColor}
                            onClick={() => toggleSort(f)}
                        >
                            {f.toUpperCase()}
                            {sortBy === f
                                ? sortOrder === 'asc'
                                    ? ' ↑'
                                    : ' ↓'
                                : null}
                        </Th>
                    ))}
                    <Th color={headerColor}>MFG DATE</Th>
                    <Th color={headerColor}>STATUS</Th>
                    <Th color={headerColor}>SPARKLINE</Th>
                    <Th color={headerColor}>ACTIONS</Th>
                </Tr>
            </Thead>
            <Tbody>
                {displayed.map(b => (
                    <Tr
                        key={b.id}
                        _hover={{ bg: 'beige' }}
                    >
                        <Td>
                            <Checkbox
                                isChecked={selectedIds.includes(
                                    b.id
                                )}
                                onChange={() =>
                                    toggleSelect(b.id)
                                }
                            />
                        </Td>
                        <Td>#{b.id}</Td>
                        <Td>{b.type}</Td>
                        <Td>{b.capacity} Ah</Td>
                        <Td>
                            {b.lastHealth != null
                                ? `${Math.round(
                                    b.lastHealth * 100
                                )}%`
                                : '—'}
                        </Td>
                        <Td>{b.manufacture_date}</Td>
                        <Td>
                            <Badge
                                colorScheme={
                                    statusColor[
                                    getStatus(b.lastHealth)
                                    ]
                                }
                            >
                                {getStatus(b.lastHealth)}
                            </Badge>
                        </Td>
                        <Td>
                            <Sparkline
                              batteryId={b.id}
                            />
                        </Td>
                        <Td>
                            <Button
                                size="sm"
                                leftIcon={<FaEdit />}
                                variant="outline"
                                colorScheme="teal"
                                onClick={() => openEdit(b)}
                            >
                                Edit
                            </Button>
                            <Button
                                size="sm"
                                leftIcon={<FaTrash />}
                                ml={2}
                                colorScheme="red"
                                variant="solid"
                                onClick={() =>
                                    confirmDelete(b.id)
                                }
                            >
                                Del
                            </Button>
                        </Td>
                    </Tr>
                ))}
            </Tbody>
        </Table>
    </TableContainer>
```

*Figure 4.13: Code Snippet from BatteryManagement.jsx showing the table layout and row mapping*

Figure 4.14 shows the rendered UI in the browser. The toolbar at the top features a dark-teal “Add Battery” button, a search input with a magnifying-glass icon, an outline “Export CSV” button, a red “Delete Selected” button that activates when rows are checked, and a red “Stop Simulation” button (which would display “Run Simulation” when no simulation is running). The table below lists five battery entries with alternating row shading, each row showing the ID prefixed with “#,” the battery type, capacity in amp-hours, last recorded health percentage, manufacture date, a “WARNING” status badge, a sparkline indicating recent health trend and action buttons for editing or deleting the record. This layout enables bulk operations, quick search and inline management of battery data.


*Figure 4.14: Screenshot of the rendered Battery Management page UI*

### 4.2.4 Analytics Page

The Analytics page provides a fleet-level overview of battery health, remaining lifespan and status distribution over time. Its layout is composed of five main sections: a set of high-level statistic cards, a seven-day trend chart, a status distribution bar chart, a summary of key telemetry ranges and a drill-down table of individual batteries. Each section is implemented as a self-contained React component, enabling reuse and isolation of concerns.

Figure 4.15 illustrates the code that renders the three top-row statistic cards. An `<HStack>` arranges three `<Stat>` components side by side, each styled with a colored left border to indicate its metric: average state of health, average remaining life and total number of batteries. Values are formatted with `toFixed` to one decimal place for percentages and zero decimals for hours, ensuring clarity at a glance.

```
{/* ── Stat Cards ── */}
<HStack spacing={4} mb={6} align="stretch">
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm" borderLeft="4px solid" borderColor="teal.400">
        <StatLabel>Average SOH</StatLabel>
        <StatNumber color="teal.500">{(avgSoH * 100).toFixed(1)}%</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm" borderLeft="4px solid" borderColor="orange.400">
        <StatLabel>Avg Remaining Life</StatLabel>
        <StatNumber color="orange.400">{avgLife.toFixed(0)} hrs</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm" borderLeft="4px solid" borderColor="gray.400">
        <StatLabel>Total Batteries</StatLabel>
        <StatNumber>{batteries.length}</StatNumber>
    </Stat>
</HStack>
```

*Figure 4.15: Code Snippet from Analytics.jsx showing the stat card layout*

Figure 4.16 shows the implementation of the seven-day trend chart. A `<Box>` wrapper with padding and shadow houses a header row containing a title and download buttons for CSV and PNG exports. Below, a Recharts `<ComposedChart>` plots dual axes: the left axis for average SoH (as a percentage) and the right axis for average life (in hours). Cartesian grid lines, a legend and a tooltip formatter guard against NaN values and enhance readability.

```jsx
{/* ── 7-Day Trend Chart ── */}
<Box mb={6} bg={cardBg} p={4} borderRadius="md" boxShadow="sm" ref={trendRef}>
    <Flex justify="space-between" align="center" mb={2}>
        <Heading size="md">7-Day Trend</Heading>
        <HStack>
            <IconButton
                size="sm"
                icon={<DownloadIcon />}
                onClick={() => downloadCSV(trendData, ['date', 'avgSoH', 'avgLife'], 'trend.csv')}
            />
            <IconButton
                size="sm"
                icon={<DownloadIcon />}
                onClick={() => downloadPNG(trendRef, 'trend.png')}
            />
        </HStack>
    </Flex>
    <ResponsiveContainer width="100%" height={250}>
        <ComposedChart data={trendData}>
            <CartesianGrid strokeDasharray="3 3" />
            <XAxis dataKey="date" />
            <YAxis yAxisId="left" unit="%" />
            <YAxis yAxisId="right" orientation="right" />
            {/* ← formatter to guard against NaN */}
            <Tooltip formatter={(val, name) => [
                typeof val === 'number' && !isNaN(val) ? val : '–',
                name
            ]} />
            <Legend />
            <Line yAxisId="left" dataKey="avgSoH" name="Avg SOH" stroke="#2C7A7B" dot={false} />
            <Line yAxisId="right" dataKey="avgLife" name="Avg Life" stroke="#DD6B20" dot={false} />
        </ComposedChart>
    </ResponsiveContainer>
</Box>
```

*Figure 4.16: Code Snippet from Analytics.jsx showing the 7-day trend chart setup*

Figure 4.17 presents the status distribution component code. Within a styled <Box>, a <BarChart> displays counts of batteries in each health category – Healthy, Warning, Critical and Unknown. Download buttons permit exporting the underlying data or chart image. An optional "Clear filter" button appears when a status filter is active, enabling users to reset the drill-down view.

```jsx
{/* — Status Distribution — */}
<Box mb={6} bg={cardBg} p={4} borderRadius="md" boxShadow="sm" ref={statusRef}>
    <Flex justify="space-between" align="center" mb={2}>
        <Heading size="md">Status Distribution</Heading>
        <HStack>
            <IconButton
                size="sm"
                icon={<DownloadIcon />}
                onClick={() => downloadCSV(barData, ['status', 'count'], 'status.csv')}
            />
            <IconButton
                size="sm"
                icon={<DownloadIcon />}
                onClick={() => downloadPNG(statusRef, 'status.png')}
            />
        </HStack>
    </Flex>
    <ResponsiveContainer width="100%" height={200}>
        <BarChart data={barData}>
            <XAxis dataKey="status" />
            <YAxis allowDecimals={false} />
            {/* ← same NaN guard here */}
            <Tooltip formatter={(val, name) => [
                typeof val === 'number' && !isNaN(val) ? val : '–',
                name
            ]} />
            <Bar
                dataKey="count"
                fill="#2C7A7B"
                onClick={d => setFilterStatus(d.status)}
            />
        </BarChart>
    </ResponsiveContainer>
    {filterStatus && (
        <Button size="sm" mt={2} onClick={() => setFilterStatus(null)}>
            Clear filter "{filterStatus}"
        </Button>
    )}
</Box>
```

*Figure 4.17: Code Snippet from Analytics.jsx showing the status distribution bar chart setup*

Figure 4.18 captures the telemetry summary and drill-down table markup. A second <HStack> presents minimum, average and maximum values for voltage, temperature and cycle count. Beneath, a striped <Table> lists each battery's ID, type, state of health, remaining life and current status, with rows filtered by any active status selection.

```jsx
{/* ── Telemetry Summary ── */}
<HStack spacing={4} mb={6} align="stretch">
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Voltage (V)</StatLabel>
        <StatNumber>{vMin.toFixed(2)}/{vAvg.toFixed(2)}/{vMax.toFixed(2)}</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Temperature (°C)</StatLabel>
        <StatNumber>{tMin.toFixed(1)}/{tAvg.toFixed(1)}/{tMax.toFixed(1)}</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Cycle Count</StatLabel>
        <StatNumber>{cMin}/{cAvg.toFixed(0)}/{cMax}</StatNumber>
    </Stat>
</HStack>

{/* ── Drill-down Table ── */}
<Box bg={cardBg} p={4} borderRadius="md" boxShadow="sm">
    <Heading size="md" mb={3}>
        Batteries {filterStatus ? `– ${filterStatus}` : ''}
    </Heading>
    <Table variant="striped" colorScheme="gray">
        <Thead>
            <Tr>
                <Th>ID</Th>
                <Th>Type</Th>
                <Th>SoH</Th>
                <Th>Life (hrs)</Th>
                <Th>Status</Th>
            </Tr>
        </Thead>
        <Tbody>
            {batteries
                .filter(b => !filterStatus || getStatus(b.lastHealth) === filterStatus)
                .map(b => (
                    <Tr key={b.id}>
                        <Td>#{b.id}</Td>
                        <Td>{b.type}</Td>
                        <Td>{b.lastHealth != null
                            ? `${Math.round(b.lastHealth * 100)}%`
                            : '–'}
                        </Td>
                        <Td>{b.lastLife != null
                            ? Math.round(b.lastLife)
                            : '–'}
                        </Td>
                        <Td>{getStatus(b.lastHealth)}</Td>
                    </Tr>
                ))
            }
        </Tbody>
    </Table>
</Box>
```

*Figure 4.18: Code Snippet from Analytics.jsx showing the telemetry summary and drill-down table layout*

The fully rendered Analytics interface appears in Figure 4.19. The top cards show an average SoH of 66.3 %, average remaining life of 803 hrs and a fleet size of five batteries. The seven-day trend chart traces SoH and life trajectories over the past week. The status distribution bar chart highlights four healthy units and one critical unit. Telemetry ranges are presented as "3.12 / 3.67 / 4.14 V," "19.8 / 27.8 / 45.7 °C" and "5 / 450 / 979 cycles." Finally, the drill-down table lists each battery's details, enabling targeted analysis of individual performance.

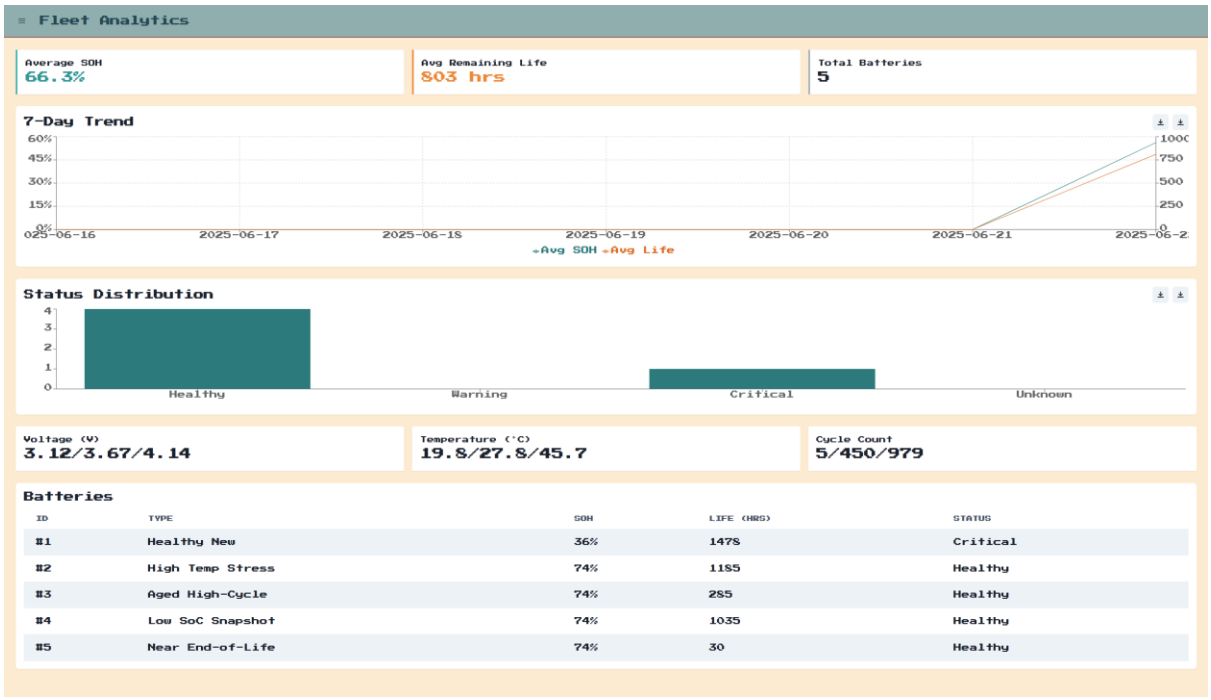


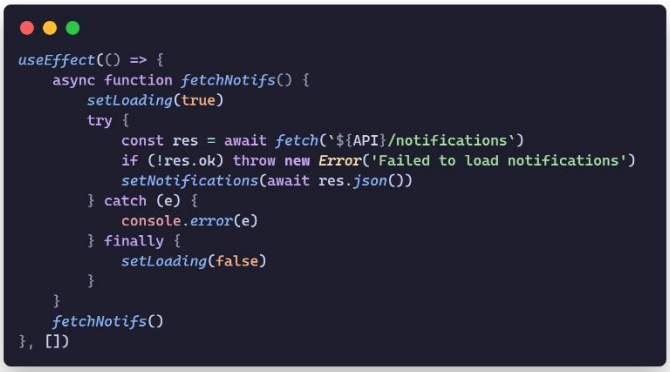

*Figure 4.19: Screenshot of the rendered Analytics page user interface*

### 4.2.5 Notification and Settings

The Notification and Settings pages complete the front-end by providing a unified view of system alerts and a configurable interface for tuning alert thresholds. The Notifications page lists all real-time alerts with filtering and severity indicators, while the Settings page presents interactive controls to adjust parameters such as state-of-health and temperature thresholds. These pages ensure that users can both monitor critical events and customise the conditions that trigger them.

When the Notifications component mounts, a `useEffect` hook loads all alerts from the back-end and manages a loading state. Refer to Figure 4.20 for the snippet that invokes `fetchNotifs`, handles errors and updates the `notifications` array. An input field bound to `filterId` enables users to narrow the list by battery. The filter input together with the table markup appears in Figure 4.21. The table renders each notification's timestamp, battery identifier, alert type, message text and a severity badge whose colour reflects the alert level.

Refer to Figure 4.24 for the live Notifications UI. A "Filter by Battery ID" box sits at the top. Below it, a scrollable table displays entries such as "22/06/2025, 01:58:27 pm," battery #1, "SoH Warning at 36%" and a red "CRITICAL" badge when appropriate. This layout gives users rapid insight into recent alerts and their severity.



```
useEffect(() => {
    async function fetchNotifs() {
        setLoading(true)
        try {
            const res = await fetch(`${API}/notifications`)
            if (!res.ok) throw new Error('Failed to load notifications')
            setNotifications(await res.json())
        } catch (e) {
            console.error(e)
        } finally {
            setLoading(false)
        }
    }
    fetchNotifs()
}, [])
```

*Figure 4.20: Code Snippet from Notifications.jsx showing the useEffect hook that loads notifications*

```jsx
<InputGroup maxW="240px" mb={4}>
    <InputLeftElement pointerEvents="none">
        <SearchIcon color="gray.500" />
    </InputLeftElement>
    <Input
        placeholder="Filter by Battery ID…"
        value={filterId}
        onChange={e => setFilterId(e.target.value)}
        bg="white"
    />
</InputGroup>

{loading ? (
    <Spinner size="lg" color="darkTeal" />
) : (
    <TableContainer overflowX="auto" borderRadius="md" boxShadow="sm" bg="white">
        <Table
            variant="striped"
            colorScheme="gray"
            width="100%"
            // tableLayout="auto"
            size="sm"
        >
            <Thead bg={headerBg}>
                <Tr>
                    <Th color={headerColor} whiteSpace="nowrap">TIME</Th>
                    <Th color={headerColor} whiteSpace="nowrap">BATTERY</Th>
                    <Th color={headerColor} whiteSpace="nowrap">TYPE</Th>
                    <Th color={headerColor}>MESSAGE</Th>
                    <Th color={headerColor} whiteSpace="nowrap">SEVERITY</Th>
                </Tr>
            </Thead>
            <Tbody>
                {filtered.map(n => (
                    <Tr key={n.id}>
                        <Td whiteSpace="nowrap">
                            {new Date(n.timestamp).toLocaleString(undefined, {
                                day: '2-digit', month: '2-digit', year: 'numeric',
                                hour: '2-digit', minute: '2-digit', second: '2-digit',
                            })}
                        </Td>
                        <Td whiteSpace="nowrap">#{n.battery_id}</Td>
                        <Td whiteSpace="nowrap" textTransform="capitalize">
                            {n.type.replace(/_/g, ' ')}
                        </Td>
                        <Td
                            whiteSpace="pre-wrap"
                            wordBreak="break-word"
                            maxW={{ base: '100px', md: '200px', lg: '300px' }}
                        >
                            {n.message}
                        </Td>
                        <Td whiteSpace="nowrap">
                            <Badge colorScheme={
                                n.severity === 'critical'
                                    ? 'red'
                                    : n.severity === 'warning'
                                        ? 'orange'
                                        : 'green'
                            }>
                                {n.severity}
                            </Badge>
                        </Td>
                    </Tr>
                ))}

                {filtered.length === 0 && (
                    <Tr>
                        <Td colSpan={5} textAlign="center" py={8} color="gray.500">
                            No notifications match “{filterId}”.
                        </Td>
                    </Tr>
                )}
            </Tbody>
        </Table>
    </TableContainer>
```

*Figure 4.21: Code Snippet from Notifications.jsx showing the filter input and table layout*

On mounting the Settings component, a `useEffect` hook fetches current threshold values from the `/settings` endpoint and toggles a spinner while loading. The initial fetch logic is captured in Figure 4.22. Once the data arrives, an array of slider configurations is mapped to render five threshold controls, each paired with a descriptive label showing its current value. Figure 4.23 illustrates this dynamic slider setup and the "Save Changes" button, which remains disabled until the settings object is populated.

```
// fetch on mount
useEffect(() => {
    (async () => {
        setLoading(true)
        try {
            const res = await fetch(`${API}/settings`)
            if (!res.ok) throw new Error()
            setSettings(await res.json())
        } catch {
            toast({ title: 'Failed to load settings', status: 'error' })
        } finally {
            setLoading(false)
        }
    })()
}, [])
```

*Figure 4.22: Code Snippet from Settings.jsx showing the useEffect hook that fetches current settings*

```jsx
<Box p={6} overflowY="auto">
    {loading || !settings ? (
        <Spinner size="lg" color="darkTeal" />
    ) : (
        sliders.map(({ key, label, min, max, step, format }) => (
            <Box mb={6} key={key}>
                <Text mb={1}>{label}: {format(settings[key])}</Text>
                <Slider
                    value={settings[key]}
                    min={min}
                    max={max}
                    step={step}
                    onChange={val =>
                        setSettings(s => ({ ...s, [key]: val }))
                    }
                >
                    <SliderTrack><SliderFilledTrack /></SliderTrack>
                    <SliderThumb />
                </Slider>
            </Box>
        ))
    )}

    <Button
        mt={4}
        colorScheme="teal"
        onClick={handleSave}
        isDisabled={!settings}
    >
        Save Changes
    </Button>
</Box>
```

*Figure 4.23: Code Snippet from Settings.jsx showing the dynamic slider controls and Save Changes button*

Refer to Figure 4.25 for the rendered Settings UI. Sliders for parameters such as SoH Warning (50 %), SoH Critical (30 %), Temp Warning (50 °C), Temp Critical (60 °C) and Lifespan Spike (20 %) appear in a vertical list. The teal “Save Changes” button sits below the sliders, enabling users to persist their adjustments. This design delivers an intuitive, real-time interface for fine-tuning alert thresholds without manual configuration files.

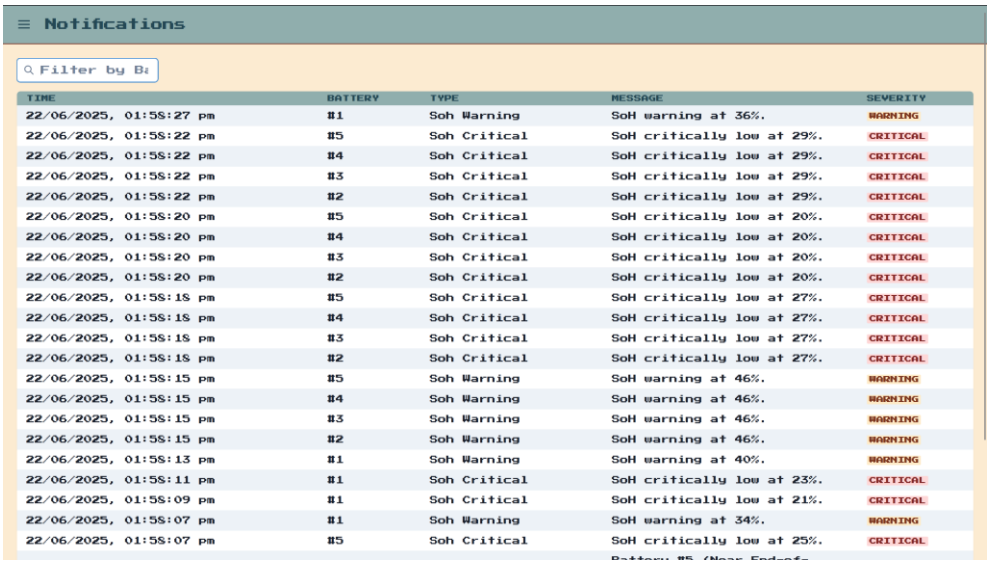



*Figure 4.24: Screenshot of the rendered Notifications page user interface*

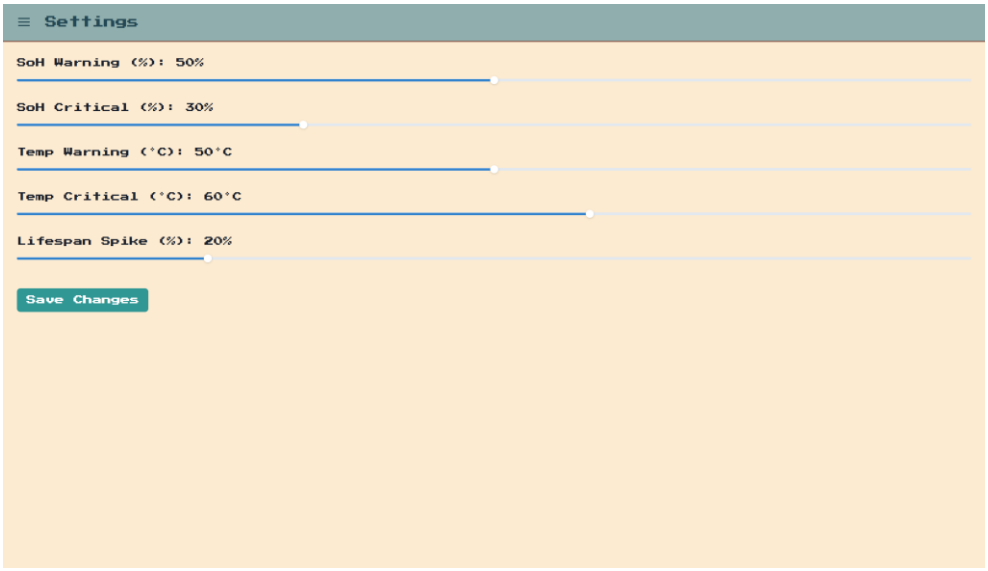



*Figure 4.25: Screenshot of the rendered Settings page user interface*

### 4.2.6 Authentication (Login/Signup)

The authentication flow is implemented in a single `AuthPage` component that combines an animated branding panel with toggleable login and signup forms. A motion-enabled Flex container animates the layout when switching modes.

Refer to Figure 4.26 for the `AuthPage` code. The snippet begins at the `export default function AuthPage()` declaration and continues through the closing `</MotionFlex>` tag. The component maintains an `isSignup` state and a `toggleMode` function. The left panel renders a static <TitlePanel> on a dark teal background. The right panel displays the `<TypingLogo>` animation and either `<LoginForm>` or `<SignUpForm>` depending on the current mode.

```jsx
export default function AuthPage() {
    const [isSignup, setIsSignup] = useState(false)
    const toggleMode = () => setIsSignup((s) => !s)

    return (
        <MotionFlex
            minH="100vh"
            flexDirection={isSignup ? 'row-reverse' : 'row'}
            layout
            transition={{ duration: 0.5 }}
        >
            {/* Left Panel: Static Title on darkTeal */}
            <MotionBox
                flex="1"
                bg="darkTeal"
                display="flex"
                alignItems="center"
                justifyContent="center"
                p={4}
                layout
            >
                <TitlePanel />
            </MotionBox>

            {/* Right Panel: TypingLogo + Login/SignUp Form */}
            <MotionBox
                flex="1"
                bg="beige"
                display="flex"
                flexDirection="column"
                alignItems="center"
                justifyContent="center"
                p={4}
                layout
            >
                {/* (a) Typing animation for "VoltaSense" with blinking cursor */}
                <TypingLogo />

                {/* (b) Show either LoginForm or SignUpForm */}
                {isSignup ? (
                    <SignUpForm toggleMode={toggleMode} />
                ) : (
                    <LoginForm toggleMode={toggleMode} />
                )}
            </MotionBox>
        </MotionFlex>
    )
}
```

*Figure 4.26: Code Snippet from AuthPage.jsx showing the MotionFlex container and conditional rendering of the LoginForm and SignUpForm*

Refer to Figure 4.27 for the `LoginForm` code. This snippet shows the function declaration, local state for `username`, `password`, `error` and `loading`, and the `handleSubmit` routine. Form data is encoded as URL-encoded parameters, posted to `/api/login`, and on success the returned access token is stored in `localStorage`, followed by navigation to the dashboard. The JSX renders a styled box with input fields, an error alert, a “Log In” button and a “Sign up instead” toggle.

```
function LoginForm({ toggleMode }) {
    const navigate = useNavigate()
    const [username, setUsername] = useState('')
    const [password, setPassword] = useState('')
    const [error, setError] = useState(null)
    const [loading, setLoading] = useState(false)

    const handleSubmit = async () => {
        setError(null)
        setLoading(true)
        try {
            const form = new URLSearchParams()
            form.append('username', username)
            form.append('password', password)

            const resp = await fetch('http://127.0.0.1:8000/api/login', {
                method: 'POST',
                headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
                body: form.toString(),
            })
            if (!resp.ok) {
                const data = await resp.json()
                throw new Error(data.detail || 'Login failed')
            }
            const data = await resp.json()
            localStorage.setItem('token', data.access_token)
            navigate('/dashboard')
        } catch (err) {
            setError(err.message)
        } finally {
            setLoading(false)
        }
    }

    return (
        <Box
            bg="mutedTeal"
            border="2px solid"
            borderColor="rust"
            borderRadius="2xl"
            p={8}
            w="80"
            maxW="sm"
        >
            <Heading
                as="h2"
                fontFamily="heading"
                fontSize="2xl"
                mb={6}
                color="darkTeal"
            >
                Welcome Back!
            </Heading>
            <Stack spacing={4}>
                {error && (
                    <Alert status="error">
                        <AlertIcon />
                        {error}
                    </Alert>
                )}
                <Input
                    placeholder="Username"
                    value={username}
                    onChange={(e) => setUsername(e.target.value)}
                    bg="beige"
                    size="lg"
                    borderColor="darkTeal"
                    borderRadius="md"
                    fontFamily="body"
                    _placeholder={{ color: 'darkTeal' }}
                />
                <Input
                    type="password"
                    placeholder="Password"
                    value={password}
                    onChange={(e) => setPassword(e.target.value)}
                    bg="beige"
                    size="lg"
                    borderColor="darkTeal"
                    borderRadius="md"
                    fontFamily="body"
                    _placeholder={{ color: 'darkTeal' }}
                />
                <Link
                    href="#"
                    fontFamily="body"
                    color="rust"
                    fontSize="sm"
                    textAlign="left"
                >
                    Forgot password?
                </Link>
                <Button
                    bg="coral"
                    color="beige"
                    size="lg"
                    isLoading={loading}
                    _hover={{ bg: 'rust' }}
                    onClick={handleSubmit}
                >
                    Log In
                </Button>
                <Button
                    variant="outline"
                    borderColor="darkTeal"
                    color="darkTeal"
                    size="lg"
                    onClick={toggleMode}
                >
                    Sign up instead
                </Button>
            </Stack>
        </Box>
    )
}
```

*Figure 4.27: Code Snippet from AuthPage.jsx showing the LoginForm component with input bindings and submit logic*

Refer to Figure 4.28 for the rendered login interface. The browser screenshot shows the left branding panel and the right login card. The card title “Welcome Back!” appears above two input fields for username and password. Below these fields, a coral “Log In” button and an outline “Sign up instead” button allow switching modes. A “Forgot password?” link is also provided.


*Figure 4.28: Screenshot of the rendered Login page UI displaying the branding panel and login form*

Refer to Figure 4.29 for the rendered signup interface. After toggling, the right panel displays a “Create Account” card with fields for name, email, password and confirmation. A red “Sign Up” button submits the form, and a “Already have an account? Log in” link returns to the login view. The left branding panel remains static, preserving visual continuity.

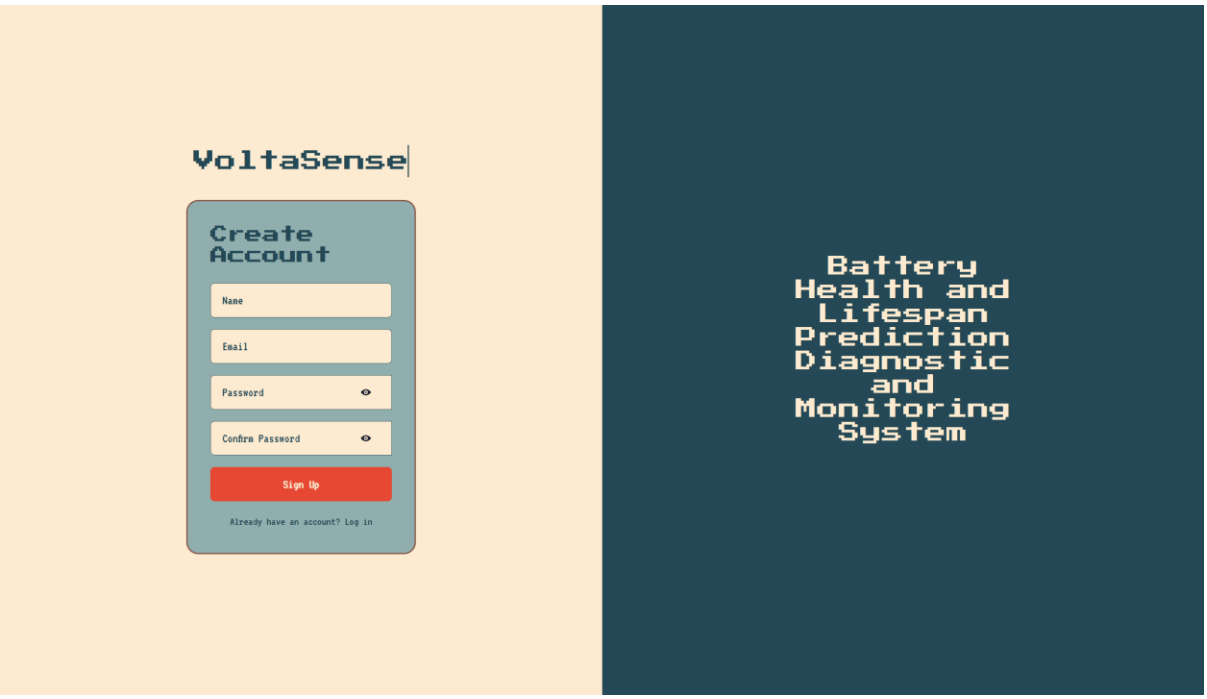



*Figure 4.29: Screenshot of the rendered Sign-Up page UI displaying the branding panel and registration form*

### 4.3 Backend Implementation

This section describes the server-side components that support the Battery Health and Lifespan Prediction Diagnostic and Monitoring System. It begins with the design of RESTful API endpoints and their corresponding controller logic, which validate incoming requests and orchestrate data flow. The following subsection details the data model definitions, ORM configuration and SQLite database schema used to store battery metadata, telemetry records and prediction results. Next, the integration of the Python-based prediction engine is presented, illustrating how raw telemetry is processed to generate health and lifespan estimates. Finally, the notification service is explained, showing how alerts are triggered when measured or predicted values cross user-defined thresholds and how those alerts are delivered to the front-end. Each part includes annotated code excerpts and a discussion of key design decisions.

### 4.3.1 API Design and Controllers

The back-end API is implemented using FastAPI, which provides a declarative, high-performance framework for defining routes, request validation and dependency injection. Refer to Figure 4.30 for the application setup in main.py. Here the `FastAPI` instance is created with a descriptive title and summary of its capabilities. A CORS middleware is added to allow cross-origin requests during development. In the `@app.on_event("startup")` handler the SQLite database schema is initialized and a default Settings row is inserted if none exists, ensuring a consistent starting state.

```python
app = FastAPI(
    title="Battery Health API",
    description="Auth, data ingestion, prediction, telemetry, notifications & settings"
)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],        # dev-friendly
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.on_event("startup")
def on_startup():
    # create tables
    init_db()

    # ensure a single Settings row exists
    from sqlmodel import Session as _Session
    with _Session(engine) as session:
        if not session.get(Settings, 1):
            session.add(Settings())  # use all the defaults you've defined
            session.commit()
```

*Figure 4.30: Code snippet from main.py showing FastAPI instantiation, CORS middleware and the startup event*

Database sessions are provided to each request handler via a generator dependency. Refer to Figure 4.31, which shows `get_session()` in db.py. This function yields a SQLModel `Session` bound to the engine and automatically closes it when the request completes. By declaring `session: Session = Depends(get_session)` in controller signatures, each endpoint gains transactional access to the database without manual setup or teardown.

```python
def get_session() -> Generator[Session, None, None]:
    """
    Dependency to inject a SQLModel Session.
    """
    with Session(engine) as session:
        yield session
```

*Figure 4.31: Code snippet from db.py showing the get_session dependency generator*

Battery management endpoints expose full CRUD operations under the `/api/batteries` path. Refer to Figure 4.32 for the `create_battery` and `list_batteries` handlers. The POST handler accepts a `BatteryCreate` payload, validates that the type is non-empty and capacity positive, then inserts a new `Battery` record. After committing, it calls `_notify` to emit an informational alert indicating the addition. The GET handler returns all battery records as a list of `BatteryOut` schemas.

```python
@app.post("/api/batteries", response_model=BatteryOut, status_code=status.HTTP_201_CREATED)
async def create_battery(
    batt: BatteryCreate,
    session: Session = Depends(get_session),
):
    if not batt.type.strip():
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Type cannot be empty")
    if batt.capacity <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Capacity must be positive")

    new = Battery(
        type=batt.type,
        capacity=batt.capacity,
        manufacture_date=batt.manufacture_date
    )
    session.add(new)
    session.commit()
    session.refresh(new)

    # 1) notify battery-added
    _notify(
        session,
        battery_id=new.id,
        type="battery_added",
        message=f"Battery #{new.id} ({new.type}) was added.",
        severity="info"
    )

    return BatteryOut(
        id=new.id,
        type=new.type,
        capacity=new.capacity,
        manufacture_date=new.manufacture_date,
    )

@app.get("/api/batteries", response_model=List[BatteryOut])
async def list_batteries(session: Session = Depends(get_session)):
    bats = session.exec(select(Battery)).all()
    return [
        BatteryOut(
            id=b.id,
            type=b.type,
            capacity=b.capacity,
            manufacture_date=b.manufacture_date,
        )
        for b in bats
    ]
```

*Figure 4.32: Code snippet from main.py showing the battery creation and listing endpoints*

Prediction and telemetry ingestion are handled by parameterized routes. Refer to Figure 4.33 for the prediction endpoint at `/api/battery/{battery_id}/predictions`. This handler loads the target battery and current threshold settings, computes the Arrhenius degradation rate from the supplied temperature, calculates state-of-health from nameplate capacity, and estimates remaining cycles and hours. A new `BatteryPrediction` record is persisted, and critical or warning notifications are generated based on the configured SoH and lifespan spike thresholds.

```python
@app.post("/api/battery/{battery_id}/predictions", response_model=BatteryPredictOut)
async def create_prediction(
    payload: BatteryPredictIn,
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")

    # pull your thresholds from Settings
    settings = session.get(Settings, 1)

    # Arrhenius rate
    T_k = payload.temperature_c + 273.15
    if T_k <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Temperature must be above -273.15°C")
    k_temp = PREDICT_A * math.exp(-PREDICT_EA / (R_CONST * T_k))

    # SoH calculation
    if payload.nameplate_capacity <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Nameplate capacity must be > 0")
    soh_now = payload.current_capacity / payload.nameplate_capacity
    if not (0 < soh_now <= 1):
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "SoH out of realistic range (0 < SoH ≤ 1)")

    remaining = max(MAX_CYCLES - payload.cycle_count, 0)
    rem_hours = remaining * HOURS_PER_CYCLE

    # previous prediction for delta
    prev: Optional[BatteryPrediction] = session.exec(
        select(BatteryPrediction)
        .where(BatteryPrediction.battery_id == battery_id)
        .order_by(BatteryPrediction.timestamp.desc())
        .limit(1)
    ).first()

    bp = BatteryPrediction(
        battery_id=battery_id,
        soh=soh_now,
        remaining_cycles=remaining,
        predicted_lifespan_hours=rem_hours,
        degradation_rate=k_temp,
    )
    session.add(bp)
    session.commit()
    session.refresh(bp)

    # SoH alerts per settings
    if soh_now <= settings.soh_crit:
        _notify(session, battery_id, "soh_critical",
                f"SoH critically low at {soh_now:.0%}.", "critical")
    elif soh_now <= settings.soh_warn:
        _notify(session, battery_id, "soh_warning",
                f"SoH warning at {soh_now:.0%}.", "warning")

    # lifespan spike
    if prev:
        delta = (prev.predicted_lifespan_hours - rem_hours) / (prev.predicted_lifespan_hours + 1e-9)
        if delta > settings.lifespan_spike_pct:
            _notify(session, battery_id, "lifespan_spike",
                    f"Lifespan dropped by {(delta*100):.0f}% since last reading.", "warning")

    return BatteryPredictOut(
        timestamp=bp.timestamp,
        soh=bp.soh,
        remaining_cycles=bp.remaining_cycles,
        predicted_lifespan_hours=bp.predicted_lifespan_hours,
        degradation_rate=bp.degradation_rate,
    )
```

*Figure 4.33: Code snippet from main.py showing the prediction endpoint logic*

Refer to Figure 4.34 for the telemetry POST handler at `/api/battery/{battery_id}/telemetry`. This function validates that the battery exists, creates a `BatteryTelemetry` row with voltage, current, temperature, SoC and cycle count, and commits it. After insertion, it triggers temperature, current and voltage outlier notifications when readings exceed user-defined warning or critical bounds.

```python
@app.post("/api/battery/{battery_id}/telemetry", response_model=TelemetryOut)
async def create_telemetry(
    payload: TelemetryIn,
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")

    row = BatteryTelemetry(
        battery_id=battery_id,
        batt_voltage=payload.batt_voltage,
        batt_current=payload.batt_current,
        batt_temp=payload.batt_temp,
        soc=payload.soc,
        cycle_count=payload.cycle_count,
        soh=payload.soh,
        remaining_lifespan_hours=payload.remaining_lifespan_hours,
    )
    session.add(row)
    session.commit()
    session.refresh(row)

    # temp alerts per settings
    settings = session.get(Settings, 1)
    if row.batt_temp >= settings.temp_crit:
        _notify(session, battery_id, "temp_critical",
                f"Temp critical at {row.batt_temp:.1f}°C.", "critical")
    elif row.batt_temp >= settings.temp_warn:
        _notify(session, battery_id, "temp_warning",
                f"High temp at {row.batt_temp:.1f}°C.", "warning")

    # current/voltage outliers
    if abs(row.batt_current) > 2.0:
        _notify(session, battery_id, "current_outlier",
                f"Abnormal current: {row.batt_current:.2f} A.", "warning")
    if not (2.5 <= row.batt_voltage <= 4.5):
        _notify(session, battery_id, "voltage_outlier",
                f"Abnormal voltage: {row.batt_voltage:.2f} V.", "warning")

    return TelemetryOut.from_orm(row)
```

*Figure 4.34: Code snippet from main.py showing the telemetry ingestion endpoint*

Together these controllers define a cohesive RESTful interface for managing batteries, ingesting data, generating predictions, and delivering real-time alerts.

### 4.3.2 Data Models and Database Configuration

The system's data layer relies on SQLite for lightweight file-based storage and SQLModel to define the schema, relationships and type validation in one place. Initialization and session management are encapsulated in db.py, shown in Figure 4.35. The `DATABASE_URL` constant points to `../database.db`, and the `create_engine` call prepares a connection factory with SQL echo enabled so that all generated SQL statements appear in the console. The `init_db()` function invokes `SQLModel.metadata.create_all(engine)`, which inspects every registered model class and issues `CREATE TABLE` statements only for those tables that do not yet exist. This approach guarantees that the database schema is automatically bootstrapped on application startup. The `get_session()` generator then yields a new `Session` bound to the same engine for each incoming request, ensuring that controllers never have to manage session lifecycles manually. Once the request is handled, the context manager closes the session cleanly.

```python
# Make sure this exactly matches what create_db.py uses (below).
DATABASE_URL = "sqlite:///../database.db"
engine = create_engine(DATABASE_URL, echo=True)

def init_db() -> None:
    """
    Create all tables (if they don't exist yet).
    """
    SQLModel.metadata.create_all(engine)

def get_session() -> Generator[Session, None, None]:
    """
    Dependency to inject a SQLModel Session.
    """
    with Session(engine) as session:
        yield session
```

*Figure 4.35: Code snippet from db.py showing database engine setup, table initialization and get_session dependency*

The domain entities themselves are declared as subclasses of `SQLModel` in orm_models.py, as illustrated in Figure 4.36. The primary `Battery` class captures essential attributes such as an automatically generated integer `id`, a text `type`, a numeric `capacity` and a date `manufacture_date`. It also declares one-to-many relationships to three other models: `BatteryPrediction`, `BatteryTelemetry` and `Notification`. These relationship definitions enable SQLModel to perform joined queries or cascading deletes without additional boilerplate.

```python
class Battery(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    type: str
    capacity: float
    manufacture_date: date

    predictions: List["BatteryPrediction"] = Relationship(back_populates="battery")
    telemetry:   List["BatteryTelemetry"]  = Relationship(back_populates="battery")
    notifications: List["Notification"]    = Relationship(back_populates="battery")


class BatteryPrediction(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    battery_id: int = Field(foreign_key="battery.id", index=True)
    timestamp: datetime = Field(default_factory=datetime.utcnow)

    soh: float
    remaining_cycles: int
    predicted_lifespan_hours: float
    degradation_rate: float

    battery: Optional[Battery] = Relationship(back_populates="predictions")


class BatteryTelemetry(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    battery_id: int = Field(foreign_key="battery.id", index=True)
    timestamp: datetime = Field(default_factory=datetime.utcnow)

    batt_voltage: float
    batt_current: float
    batt_temp: float
    soc: float
    cycle_count: int
    soh: float
    remaining_lifespan_hours: float

    battery: Optional[Battery] = Relationship(back_populates="telemetry")


class Notification(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    battery_id: int = Field(foreign_key="battery.id", index=True)
    type: str              # e.g. "battery_added", "soh_warning", …
    message: str           # human-readable
    severity: str          # "info" | "warning" | "critical"
    timestamp: datetime = Field(default_factory=datetime.utcnow)

    battery: Battery = Relationship(back_populates="notifications")


class Settings(SQLModel, table=True):
    """
    A single row (id=1) that holds all alert thresholds.
    """
    id: Optional[int] = Field(default=1, primary_key=True)
    soh_warn: float = Field(default=0.5, description="SoH ≤ this ⇒ warnin  )
    soh_crit: float = Field(default=0.3, description="SoH ≤ this ⇒ critica  )
    temp_warn: float = Field(default=50.0, description="Temp ≥ this ⇒ warnin  )
    temp_crit: float = Field(default=60.0, description="Temp ≥ this ⇒ critica  )
    lifespan_spike_pct: float = Field(
        default=0.20,
        description="Δlifespan > this fraction ⇒ warnin
    )
```

*Figure 4.36: Code snippet from orm_models.py showing SQLModel classes for Battery, Prediction, Telemetry, Notification and Settings tables*

`BatteryPrediction` and `BatteryTelemetry` record chronological measurements and model outputs. Each record includes a foreign key linking back to the parent battery, a timestamp defaulting to the precise current UTC time, and fields tailored to the use case. For predictions, this includes state-of-health and degradation rate parameters computed via the Arrhenius formula. For telemetry, fields cover voltage, current, temperature, state-of-charge, cycle count and remaining lifespan in hours. By capturing both raw data and computed predictions in relational tables, the design supports complex time-series analyses and accurate audit trails.

All alerts triggered by threshold crossings or outlier detection are stored in the `Notification` table. Each notification record contains the battery identifier, a machine-readable `type` (for example, `soh_warning` or `temp_critical`), a user-friendly `message` string, a `severity` label and a timestamp. This schema serves as an immutable log that the front-end can poll to present historical alerts and to drive the notification badge counts.

Finally, the `Settings` model is implemented as a singleton table with exactly one row (primary key `id = 1`). It holds all the configurable thresholds that drive the warning and critical logic: state-of-health warning and critical cutoffs, temperature thresholds for warnings and critical alerts, and the fractional drop in predicted lifespan that constitutes a spike. Default values, such as 0.5 for the SoH warning threshold and 60.0 °C for the critical temperature threshold, are declared in `Field` definitions so they appear both in the database schema and in the automatically generated OpenAPI documentation.

Together, these two modules establish a robust and maintainable persistence foundation. They guarantee that database schema changes are centrally managed, that session usage is consistent across controllers and that each domain concept, from battery to telemetry to alerts, is modelled with precise data types and relationships.

### 4.3.3 Prediction Engine Integration

The prediction engine encapsulates a simple Arrhenius-based degradation model and a linear cycle-to-hours conversion, and is exposed both as a standalone utility and as part of the HTTP API. Figure 4.37 shows the full implementation of the `compute_prediction` function in predict.py. The routine begins by computing state-of-health as the ratio of current capacity to nameplate capacity. It then applies the Arrhenius equation to calculate a degradation rate constant $k$, converting the user-supplied temperature in Celsius to Kelvin and using an example pre-exponential factor and activation energy. Remaining cycles are estimated by solving for the number of cycles required to degrade SoH from its current value down to a chosen end-of-life threshold (for example 0.2). Finally, the function multiplies this cycle count by an assumed hours-per-cycle factor to produce a predicted remaining lifespan in hours. All four outputs such as, SoH, remaining cycles, predicted hours and degradation rate, are returned in a dictionary for downstream use.

```python
def compute_prediction(
    temperature_c: float,
    cycle_count: int,
    current_capacity: float,
    nameplate_capacity: float,
) -> Dict[str, float]:
    """
    Returns a dict with keys:
    - soh: State of Health (0.0–1.0)
    - remaining_cycles: int
    - predicted_lifespan_hours: float
    - degradation_rate: float
    """

    # 1) Compute SoH:
    soh = current_capacity / nameplate_capacity

    # 2) Arrhenius model for degradation rate k:
    #    k = A * exp(-Ea / (R * T))
    #    You should pick A & Ea based on your chemistry. Here's an example:
    A = 1.0e10                  # (just an example; tune based on your research)
    Ea = 40000.0                # Joules / mole
    R = 8.314                   # J / (mol·K)
    T = temperature_c + 273.15  # Kelvin

    k = A * math.exp(-Ea / (R * T))  # this is "per hour" or "per time unit" depending on A's units

    # 3) Use cycle_count & k to estimate remaining cycles.
    #    (For demonstration, let's assume linear fade: each cycle consumes k fraction of capacity.)
    #    This is likely an oversimplification; replace with your real equation:
    if k <= 0:
        remaining_cycles = 0
    else:
        # Suppose we declare "end of life" at SoH = 0.2 (20% of capacity).
        # Current SoH = soh. Each subsequent cycle reduces SoH by k.
        # Solve (soh - N*k = 0.2) ⇒ N = (soh - 0.2) /
        remaining_cycles = max(0, int((soh - 0.2) / k))

    # 4) Convert cycles → hours. For example, if each cycle is 1 hour long:
    hours_per_cycle = 1.0  # replace with real average if you have data
    predicted_lifespan_hours = remaining_cycles * hours_per_cycle

    return {
        "soh": soh,
        "remaining_cycles": remaining_cycles,
        "predicted_lifespan_hours": predicted_lifespan_hours,
        "degradation_rate": k,
    }
```

*Figure 4.37: Snippet of the `compute_prediction` function in predict.py*

Integration into the FastAPI controller is illustrated in Figure 4.38. Within the `/api/battery/{battery_id}/predictions` POST handler in main.py, the service first validates that the target battery exists and retrieves the active threshold settings. It then performs the same Arrhenius calculation inline (or could invoke `compute_prediction` directly), computes state-of-health, converts cycles to hours, creates a new `BatteryPrediction` record and commits it to the database. Immediately after persistence the handler issues notifications for critical or warning conditions based on the configured SoH and lifespan-spike thresholds. The endpoint returns a `BatteryPredictOut` response model containing the timestamped prediction, enabling the front end to render both historical and live prediction data.

```python
# ── Create / List Predictions ──────────────────────────────────────
@app.post("/api/battery/{battery_id}/predictions", response_model=BatteryPredictOut)
async def create_prediction(
    payload: BatteryPredictIn,
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")

    # pull your thresholds from Settings
    settings = session.get(Settings, 1)

    # Arrhenius rate
    T_k = payload.temperature_c + 273.15
    if T_k <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Temperature must be above -273.15°C")
    k_temp = PREDICT_A * math.exp(-PREDICT_EA / (R_CONST * T_k))

    # SoH calculation
    if payload.nameplate_capacity <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Nameplate capacity must be > 0")
    soh_now = payload.current_capacity / payload.nameplate_capacity
    if not (0 < soh_now <= 1):
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "SoH out of realistic range (0 < SoH ≤ 1)")

    remaining = max(MAX_CYCLES - payload.cycle_count, 0)
    rem_hours = remaining * HOURS_PER_CYCLE

    # previous prediction for delta
    prev: Optional[BatteryPrediction] = session.exec(
        select(BatteryPrediction)
        .where(BatteryPrediction.battery_id == battery_id)
        .order_by(BatteryPrediction.timestamp.desc())
        .limit(1)
    ).first()

    bp = BatteryPrediction(
        battery_id=battery_id,
        soh=soh_now,
        remaining_cycles=remaining,
        predicted_lifespan_hours=rem_hours,
        degradation_rate=k_temp,
    )
    session.add(bp)
    session.commit()
    session.refresh(bp)

    # SoH alerts per settings
    if soh_now <= settings.soh_crit:
        _notify(session, battery_id, "soh_critical",
                f"SoH critically low at {soh_now:.0%}.", "critical")
    elif soh_now <= settings.soh_warn:
        _notify(session, battery_id, "soh_warning",
                f"SoH warning at {soh_now:.0%}.", "warning")

    # lifespan spike
    if prev:
        delta = (prev.predicted_lifespan_hours - rem_hours) / (prev.predicted_lifespan_hours + 1e-9)
        if delta > settings.lifespan_spike_pct:
            _notify(session, battery_id, "lifespan_spike",
                    f"Lifespan dropped by {(delta*100):.0f}% since last reading.", "warning")

    return BatteryPredictOut(
        timestamp=bp.timestamp,
        soh=bp.soh,
        remaining_cycles=bp.remaining_cycles,
        predicted_lifespan_hours=bp.predicted_lifespan_hours,
        degradation_rate=bp.degradation_rate,
    )

@app.get("/api/battery/{battery_id}/predictions", response_model=List[BatteryPredictOut])
async def list_predictions(
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")
    results = session.exec(
        select(BatteryPrediction).where(BatteryPrediction.battery_id == battery_id)
    ).all()
    return [
        BatteryPredictOut(
            timestamp=r.timestamp,
            soh=r.soh,
            remaining_cycles=r.remaining_cycles,
            predicted_lifespan_hours=r.predicted_lifespan_hours,
            degradation_rate=r.degradation_rate,
        )
        for r in results
    ]
```

*Figure 4.38: Excerpt of the `create_prediction` endpoint in main.py*

### 4.3.4 Notification Service

The notification service provides a unified mechanism for recording every significant event like whether it is a new battery being added, a state-of-health threshold being crossed, or an anomalous telemetry reading, and for exposing those records to the front end in reverse-chronological order. All of the persistence logic is encapsulated in a small helper function, `_notify`, which is invoked by various API controllers whenever an alert condition is detected.

As shown in Figure 4.39, the `_notify` helper accepts the current database session, a battery identifier, an event type string (for example, `"soh_warning"` or `"battery_added"`), a human-readable message and a severity level (such as `"info"`, `"warning"` or `"critical"`). It constructs a new `Notification` SQLModel instance, adds it to the session and commits the transaction. Centralizing this logic in one place avoids duplication across controllers and guarantees that every alert is recorded with a consistent schema and timestamp.

```python
# ── Notification helper ──────────────────────────────────────────

def _notify(
    session: Session,
    battery_id: int,
    type: str,
    message: str,
    severity: str = "info"
):
    notif = Notification(
        battery_id=battery_id,
        type=type,
        message=message,
        severity=severity,
    )
    session.add(notif)
    session.commit()
```

*Figure 4.39: Notification helper function `_notify` in main.py*

Retrieval of stored notifications is handled by a dedicated controller method, `list_notifications`. Refer to Figure 4.40 for the implementation details. This endpoint is bound to `GET /api/notifications` and injects a SQLModel session via FastAPI's dependency system. It executes a simple `select(Notification)` query ordered by the `timestamp` column in descending order so that the newest alerts appear first. Each resulting ORM object is converted into a Pydantic `NotificationOut` model before being returned. This design ensures that the client can display a live feed of alerts, complete with severity badges and timestamps, without having to manage filtering or sorting on the client side.

```python
# ── List Notifications ──────────────────────────────────────────
@app.get("/api/notifications", response_model=List[NotificationOut])
async def list_notifications(session: Session = Depends(get_session)):
    notifs = session.exec(select(Notification).order_by(Notification.timestamp.desc())).all()
    return [NotificationOut.from_orm(n) for n in notifs]
```

*Figure 4.40: `list_notifications` endpoint in main.py*

### 4.4 Integration and Interaction

This section describes how the front-end and back-end components form a continuous, near-real-time feedback loop that keeps the user interface in sync with the latest battery health and prediction data. In this section we tie together the periodic polling logic in the React clients with the RESTful controllers on the FastAPI server, and show how telemetry ingestion, prediction computation, and notification generation all operate in concert.

First, the Dashboard component invokes its summary polling hook every ten seconds (Figure 4.41). This routine calls the `/api/batteries` endpoint to fetch the list of all batteries, then for each battery immediately requests its latest prediction from `/api/batteries/{id}/predictions`. The result is an array of enriched battery objects such as each containing its most recent State of Health, remaining lifespan estimate, and a timestamp, which drives the summary cards grid that the user sees.

```
// 1) Poll batteries + predictions every 10s
useEffect(() => {
    let interval
    async function fetchSummary() {
        setLoadingBatteries(true)
        try {
            const resp = await fetch(`${API_BASE_URL}/batteries`)
            const list = await resp.json()
            const enriched = await Promise.all(
                list.map(async (bat) => {
                    const p = await fetch(`${API_BASE_URL}/battery/${bat.id}/predictions`)
                    if (!p.ok) return { ...bat, lastHealth: null, remainingLifespan: null, lastUpdated: bat.manufacture_date }
                    const preds = await p.json()
                    if (!preds.length) return { ...bat, lastHealth: null, remainingLifespan: null, lastUpdated: bat.manufacture_date }
                    const last = preds[preds.length - 1]
                    return {
                        ...bat,
                        lastHealth: last.soh,
                        remainingLifespan: last.predicted_lifespan_hours,
                        lastUpdated: new Date().toLocaleString(),
                    }
                })
            )
            setBatteries(enriched)
        } catch (err) {
            console.error(err)
        } finally {
            setLoadingBatteries(false)
        }
    }
    fetchSummary()
    interval = setInterval(fetchSummary, 10000)
    return () => clearInterval(interval)
}, [])
```

*Figure 4.41: Snippet of the Dashboard.jsx useEffect hook that polls /api/batteries*

At the same time, the Live Monitor page uses a separate polling hook to retrieve both the full historical telemetry series on mount and then the single most recent data point every five seconds (Figure 4.42). New readings are appended to the in-memory series (capped at twenty entries), ensuring the Health and Lifespan trend charts remain fresh without reloading the entire dataset.

```
// 2) Poll newest point
useEffect(() => {
    let latestInterval
    async function fetchLatest() {
        try {
            const resp = await fetch(
                `http://127.0.0.1:8000/api/battery/${id}/telemetry?limit=1`
            )
            const [point] = await resp.json()
            setLatest(point)
            setTelemetryData(prev => {
                const next = [
                    ...prev,
                    {
                        timestamp: point.timestamp,
                        soh: point.soh,
                        remaining_lifespan_hours: point.remaining_lifespan_hours,
                    },
                ]
                return next.length > 20 ? next.slice(-20) : next
            })
        } catch (e) {
            console.error(e)
        }
    }
    fetchLatest()
    latestInterval = setInterval(fetchLatest, 5000)
    return () => clearInterval(latestInterval)
}, [id])
```

*Figure 4.42: Snippet of the BatteryDetail.jsx useEffect hook that loads the full telemetry series on mount and then polls /api/battery/{id}/telemetry?limit=1 every five seconds*

On the server side, the telemetry ingestion controller exposed at `/api/battery/{battery_id}/telemetry` (Figure 4.43) writes each incoming measurement to the `BatteryTelemetry` table. Immediately after persisting, it evaluates the new reading against the user-configured thresholds from the Settings row and, if any limits are exceeded, creates a corresponding `Notification` record via the `_notify` helper.

```python
# ── Create / List Telemetry ──────────────────────────────────────────
@app.post("/api/battery/{battery_id}/telemetry", response_model=TelemetryOut)
async def create_telemetry(
    payload: TelemetryIn,
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")

    row = BatteryTelemetry(
        battery_id=battery_id,
        batt_voltage=payload.batt_voltage,
        batt_current=payload.batt_current,
        batt_temp=payload.batt_temp,
        soc=payload.soc,
        cycle_count=payload.cycle_count,
        soh=payload.soh,
        remaining_lifespan_hours=payload.remaining_lifespan_hours,
    )
    session.add(row)
    session.commit()
    session.refresh(row)

    # temp alerts per settings
    settings = session.get(Settings, 1)
    if row.batt_temp >= settings.temp_crit:
        _notify(session, battery_id, "temp_critical",
                f"Temp critical at {row.batt_temp:.1f}°C.", "critical")
    elif row.batt_temp >= settings.temp_warn:
        _notify(session, battery_id, "temp_warning",
                f"High temp at {row.batt_temp:.1f}°C.", "warning")

    # current/voltage outliers
    if abs(row.batt_current) > 2.0:
        _notify(session, battery_id, "current_outlier",
                f"Abnormal current: {row.batt_current:.2f} A.", "warning")
    if not (2.5 <= row.batt_voltage <= 4.5):
        _notify(session, battery_id, "voltage_outlier",
                f"Abnormal voltage: {row.batt_voltage:.2f} V.", "warning")

    return TelemetryOut.from_orm(row)
```

*Figure 4.43: Snippet of the FastAPI /api/battery/{battery_id}/telemetry POST controller that persists incoming telemetry and triggers threshold-based notifications*

Similarly, the prediction endpoint at `/api/battery/{battery_id}/predictions` (Figure 4.44) applies an Arrhenius-based degradation model to compute the updated SoH, remaining cycles, predicted lifespan hours, and degradation rate. After saving the new `BatteryPrediction` row, it again checks the computed SoH and lifespan delta against warning and critical thresholds, generating additional notifications as needed before returning the prediction payload to the client.

```python
@app.post("/api/battery/{battery_id}/predictions", response_model=BatteryPredictOut)
async def create_prediction(
    payload: BatteryPredictIn,
    battery_id: int = Path(...),
    session: Session = Depends(get_session),
):
    battery = session.get(Battery, battery_id)
    if not battery:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Battery not found")

    # pull your thresholds from Settings
    settings = session.get(Settings, 1)

    # Arrhenius rate
    T_k = payload.temperature_c + 273.15
    if T_k <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Temperature must be above -273.15°C")
    k_temp = PREDICT_A * math.exp(-PREDICT_EA / (R_CONST * T_k))

    # SoH calculation
    if payload.nameplate_capacity <= 0:
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "Nameplate capacity must be > 0")
    soh_now = payload.current_capacity / payload.nameplate_capacity
    if not (0 < soh_now <= 1):
        raise HTTPException(status.HTTP_400_BAD_REQUEST, "SoH out of realistic range (0 < SoH ≤ 1)")

    remaining = max(MAX_CYCLES - payload.cycle_count, 0)
    rem_hours = remaining * HOURS_PER_CYCLE

    # previous prediction for delta
    prev: Optional[BatteryPrediction] = session.exec(
        select(BatteryPrediction)
        .where(BatteryPrediction.battery_id == battery_id)
        .order_by(BatteryPrediction.timestamp.desc())
        .limit(1)
    ).first()

    bp = BatteryPrediction(
        battery_id=battery_id,
        soh=soh_now,
        remaining_cycles=remaining,
        predicted_lifespan_hours=rem_hours,
        degradation_rate=k_temp,
    )
```

*Figure 4.44: Snippet of the FastAPI /api/battery/{battery_id}/predictions POST controller that computes and stores a new prediction then emits any alert notifications*

Together, these four routines form a tightly coordinated pipeline: the React UI continually fetches summaries and telemetry, the FastAPI server persists raw data and runs prediction logic, and threshold breaches immediately appear as notification records. This design preserves a clear separation of concerns via RESTful contracts, while delivering a responsive user experience that faithfully reflects the state of the underlying batteries.

### 4.5 Summary

Chapter 4 presented the complete implementation of VoltaSense. Section 4.2 described the React frontend components that provide real-time battery health and lifespan insights. The Dashboard polls a summary endpoint every ten seconds to refresh metrics and trend charts. The Live View Monitor retrieves both historical series and the latest telemetry at regular intervals to display SoH, SoC and temperature readings. The Battery Management page supports adding, editing, searching, exporting and bulk deleting records while displaying status sparklines. The Analytics page compiles fleet-wide statistics, renders time-based and status-based distributions and offers CSV and PNG export. The Notifications and Settings pages allow users to review alerts and adjust thresholds through search inputs, tables, and slider controls.

Section 4.3 detailed the backend design. It began with FastAPI controllers that expose endpoints for batteries, telemetry, predictions, notifications, and global settings. Code snippets illustrated application setup, the dependency-injected database session provider and each route handler. SQLModel data models define relational tables for batteries, predictions, telemetry, and notifications alongside a single Settings row to persist all alert thresholds. A standalone prediction function implements an Arrhenius-based degradation rate model and converts cycle counts to estimated hours. The prediction controller creates new records, persists them and performs threshold checks. A notification helper centralizes the creation and commitment of info, warning and critical alert records.

Section 4.4 explained how the frontend and backend interact. React `useEffect` hooks continuously fetch battery summaries, time series and the newest data points. The backend POST routes for telemetry and prediction ingestion persist incoming data and trigger appropriate notification events.

Together these layers deliver a robust system for live monitoring, historical analysis, configurable alerting, and user management. The clear separation of UI components, API controllers, data models, business logic and notification services ensures that VoltaSense can evolve to support additional metrics, scale to larger fleets and integrate with external dashboards or third-party tools in future work.

# CHAPTER 5: RESULTS AND ANALYSIS

This chapter presents the outcomes of the simulation experiments used to evaluate battery health monitoring and lifespan prediction. Five distinct profiles were exercised such as Healthy New, High-Temperature Stress, Aged High-Cycle, Low State-of-Charge Snapshot and Near End-of-Life,each driven by the BatteryManagement.jsx component. For every profile, key metrics including state-of-health, remaining useful life, voltage, temperature and cycle count were logged and visualized over time to reveal degradation patterns and predictive accuracy.

First, the software environment and profile parameters are described, explaining how the five battery scenarios are defined and executed in code. Next, detailed plots and dashboard views illustrate both fleet-level trends and individual battery behaviour. Finally, the logged data are analyzed to understand how low health thresholds, elevated temperature, and accumulated cycles influence performance. Formal validation through scenario-based test cases confirms whether the system meets its prediction and notification objectives. The chapter concludes by distilling these findings into practical recommendations for maintenance thresholds, thermal management and future enhancements such as multi-cell simulations and real-world field trials.

### 5.1 Simulation Setup

Simulations were conducted in a JavaScript development environment on a Windows 11 workstation using Node JS v22.15.1, React and TypeScript. Supporting libraries included Recharts for dynamic charting and Chakra UI for interface components. The code is managed with npm and served by the development server in the browser. All battery behavior is emulated entirely in software by the BatteryManagement.jsx component, which defines five profiles such as Healthy New, High-Temperature Stress, Aged High-Cycle, Low SoC Snapshot and Near End-of-Life,each with its own initial state-of-charge, temperature, cycle count and degradation rate. Selecting a profile from the dropdown menu triggers a periodic update loop that generates synthetic telemetry data for voltage, temperature, cycle count, state-of-health and estimated remaining useful life.

Figure 5.1 illustrates the Battery Management page. Once logged in the user navigates to this page and clicks the Run Simulation button to begin the emulation. This action causes the component to start emitting the synthetic data streams at fixed intervals, driving the real-time charts and dashboard displays presented in the following sections.

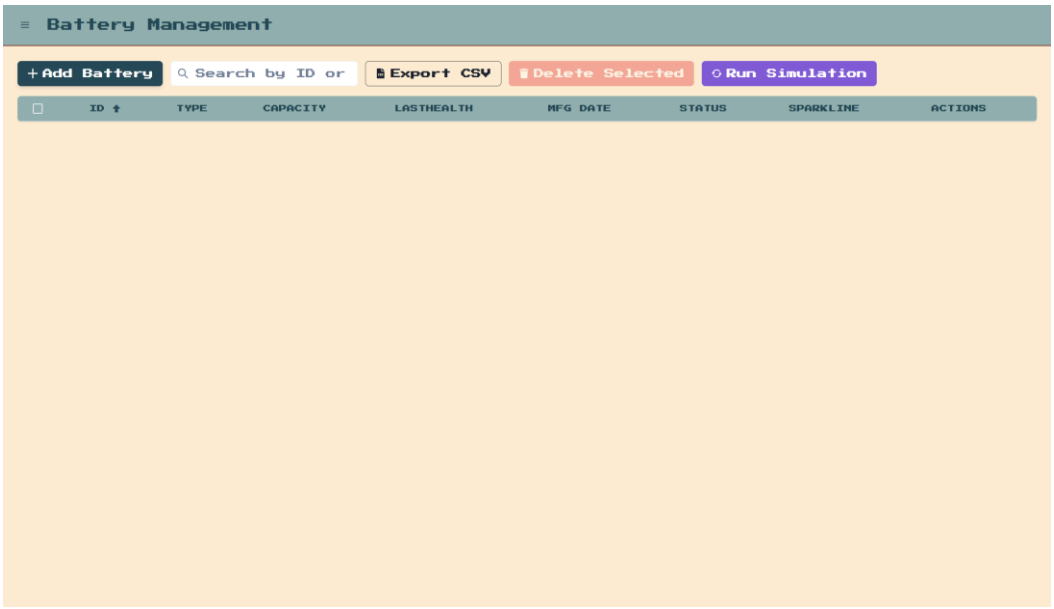


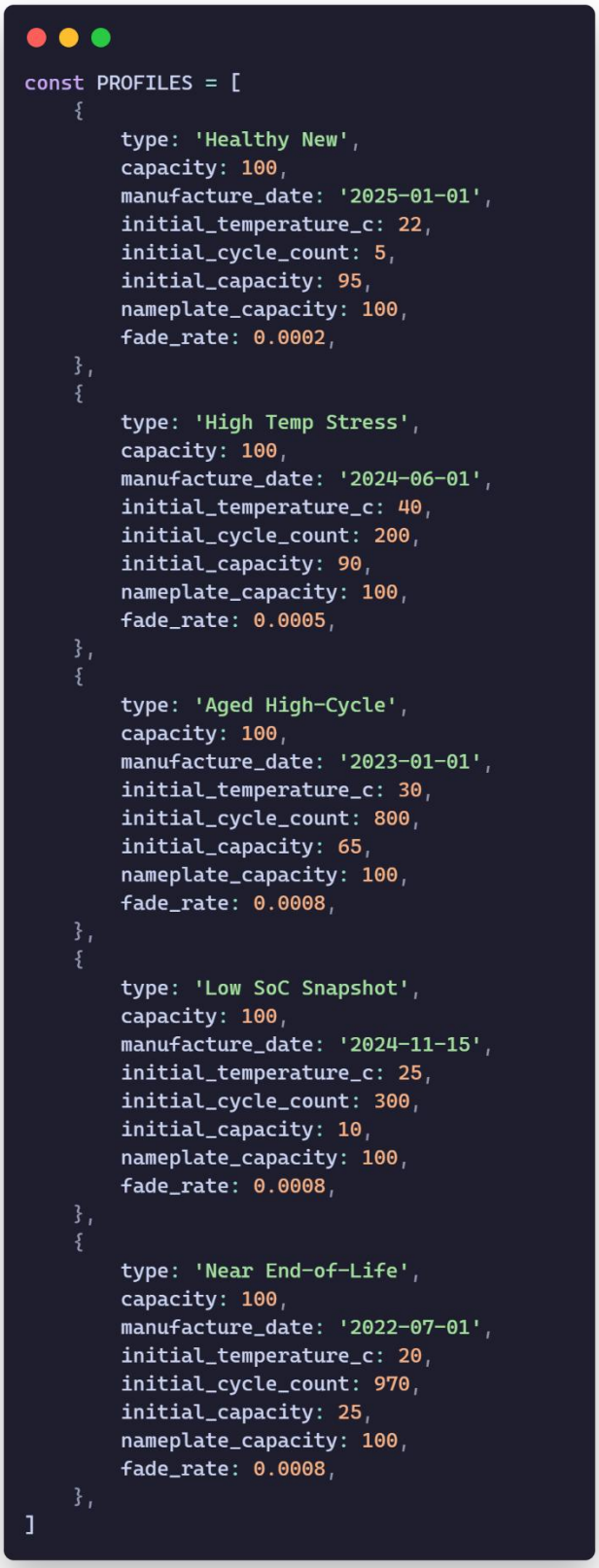

*Figure 5.1: Battery Management page showing the Run Simulation button to start emulation of the five predefined battery profiles.*

### 5.1.1 Software Environment

The development environment is configured in Visual Studio Code on Windows 11, with Node.js v22.15.1, React and TypeScript installed system-wide. VS Code's integrated terminal executes npm scripts (for example `npm install` and `npm run dev`), while extensions such as ESLint, Prettier and the Chrome Debugger provide linting, formatting and debugging support. All dependencies including Recharts for charting and Chakra UI for styling are declared in `package.json`, and environment variables (for example the API base URL) reside in a `.env` file loaded via `dotenv`. The local development server (typically on `localhost:3000`) hot-reloads on code changes, enabling rapid iteration of the BatteryManagement.jsx simulation without any physical hardware.

### 5.1.2 BatteryManagement.jsx Component

The BatteryManagement.jsx component serves as the core engine for emulating battery behavior, defining five distinct profiles entirely in code rather than on physical hardware. As shown in Figure 5.2, a constant array named `PROFILES` encapsulates each profile's characteristics such as its human-readable type, nominal capacity and manufacture date, as well as initial conditions for temperature, cycle count, capacity and a fade rate parameter. These entries mirror real-world battery states: a fresh cell, one under thermal stress, a heavily cycled cell, a low-charge snapshot and a near end-of-life unit.



```
const PROFILES = [
    {
        type: 'Healthy New',
        capacity: 100,
        manufacture_date: '2025-01-01',
        initial_temperature_c: 22,
        initial_cycle_count: 5,
        initial_capacity: 95,
        nameplate_capacity: 100,
        fade_rate: 0.0002,
    },
    {
        type: 'High Temp Stress',
        capacity: 100,
        manufacture_date: '2024-06-01',
        initial_temperature_c: 40,
        initial_cycle_count: 200,
        initial_capacity: 90,
        nameplate_capacity: 100,
        fade_rate: 0.0005,
    },
    {
        type: 'Aged High-Cycle',
        capacity: 100,
        manufacture_date: '2023-01-01',
        initial_temperature_c: 30,
        initial_cycle_count: 800,
        initial_capacity: 65,
        nameplate_capacity: 100,
        fade_rate: 0.0008,
    },
    {
        type: 'Low SoC Snapshot',
        capacity: 100,
        manufacture_date: '2024-11-15',
        initial_temperature_c: 25,
        initial_cycle_count: 300,
        initial_capacity: 10,
        nameplate_capacity: 100,
        fade_rate: 0.0008,
    },
    {
        type: 'Near End-of-Life',
        capacity: 100,
        manufacture_date: '2022-07-01',
        initial_temperature_c: 20,
        initial_cycle_count: 970,
        initial_capacity: 25,
        nameplate_capacity: 100,
        fade_rate: 0.0008,
    },
]
```

*Figure 5.2: Definition of the five battery profiles in the PROFILES array.*

Once a user triggers the simulation, React's state and effect hooks orchestrate a timed update loop that runs for a fixed number of steps. Figure 5.3 illustrates the heart of this loop, where each iteration increments the cycle count, applies a small random fluctuation and a temperature-dependent Arrhenius degradation model to the state-of-health metric, then recalculates the remaining useful life. Synthetic voltage and current values are computed from the updated SoH and SoC values to produce realistic telemetry. At every few iterations the component persists new predictions and telemetry back to the API, ensuring that the dashboard's table, status badges and sparkline mini-charts stay perfectly in sync with the evolving data.

```
for (let step = 0; step < 100; step++) {
    if (!runningRef.current) break
    for (let st of liveState) {
        st.cycle_count++
        st.soh = Math.max(
            0.05,
            st.soh -
            st.fade_rate * (0.8 + 0.4 * Math.random())
        )
        const phase =
            (st.cycle_count % 10) *
            (Math.PI / 5)
        const x = (step / 100) * 2 * Math.PI
        st.soc = Math.min(
            Math.max(
                0.55 + 0.35 * Math.sin(x + phase),
                0.1
            ),
            0.95
        )
        const dT =
            st.type === 'High Temp Stress'
                ? (Math.random() - 0.5) * 3
                : (Math.random() - 0.5) * 0.6
        st.temperature_c = Math.min(
            Math.max(st.temperature_c + dT, 15),
            60
        )
        const dtH = 2 / 3600
        const k = arrheniusRate(st.temperature_c)
        const refk = arrheniusRate(25) + 1e-9
        const degrade =
            k *
            (0.0001 / refk) *
            dtH *
            st.nameplate_capacity
        const capVal = Math.max(
            st.soh * st.nameplate_capacity - degrade,
            0.05 * st.nameplate_capacity
        )
        st.soh = capVal / st.nameplate_capacity
        const voltage = 3 + (4.2 - 3) * st.soc
        const current =
            1 + (Math.random() - 0.5) * 0.6

        // post pred
        let pr = await fetch(
            `${API}/battery/${st.id}/predictions`,
            {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    temperature_c: st.temperature_c,
                    cycle_count: st.cycle_count,
                    current_capacity:
                        st.soc * st.nameplate_capacity,
                    nameplate_capacity:
                        st.nameplate_capacity,
                }),
            }
        )
        const pj2 = await pr.json()

        // post tel
        await fetch(
            `${API}/battery/${st.id}/telemetry`,
            {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    batt_voltage: voltage,
                    batt_current: current,
                    batt_temp: st.temperature_c,
                    soc: st.soc,
                    cycle_count: st.cycle_count,
                    soh: pj2.soh,
                    remaining_lifespan_hours:
                        pj2.predicted_lifespan_hours,
                }),
            }
        )
    }

    if (step % 5 === 0) await loadList()
    await sleep(2000)
}
```

*Figure 5.3: Core simulation loop updating cycle count, SoH decay, SoC, and posting telemetry.*

Together, these two snippets demonstrate how BatteryManagement.jsx transforms a static set of parameter objects into a dynamic, time-stepped simulation. By encapsulating the profiles and the core loop in one component, the system offers a clear, maintainable separation between scenario definition and execution, while enabling real-time insights into battery degradation and lifespan prediction.

### 5.1.3 Scenario Profiles

To exercise the battery monitoring system across a realistic spectrum of operating conditions, five discrete profiles were defined in code and instantiated at runtime. Each profile captures a unique combination of age, temperature exposure, state-of-charge and cumulative cycle count, thereby enabling a comprehensive evaluation of both nominal and edge-case behavior. Figure 5.4 shows the profile selector dropdown on the Battery Management page, where “Healthy New,” “High Temp Stress,” “Aged High-Cycle,” “Low SoC Snapshot” and “Near End-of-Life” can be chosen with a single click.

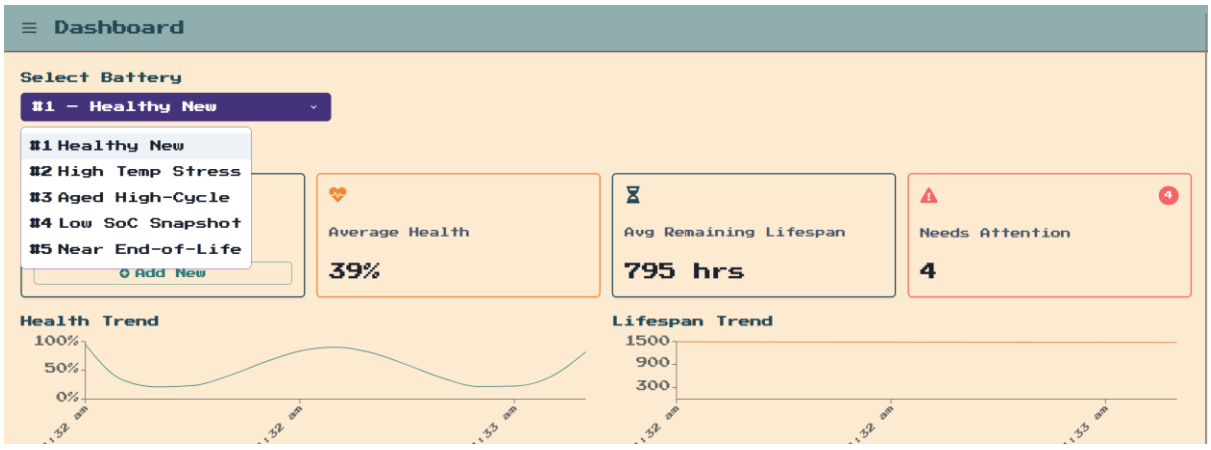



*Figure 5.4: Profile selector dropdown listing the five predefined battery scenarios.*

Once a scenario is selected, the initial parameters for that profile are seeded into the system and displayed in the table shown in Figure 5.5. In this view, each battery’s ID, human-readable type label, rated capacity, manufacture date, starting state-of-health

and sparkline mini-chart appear as distinct rows. These seeded values directly mirror the objects defined in the PROFILES array, making it immediately clear how the simulation's inputs correspond to the code definitions.

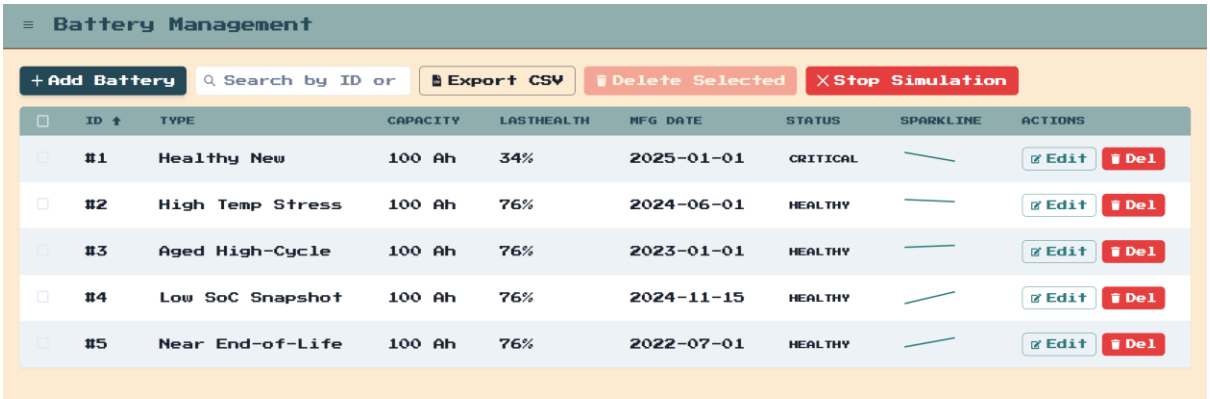



*Figure 5.5: Seeded battery table showing each profile's ID, type, capacity, initial SoH and sparkline.*

The "Healthy New" profile represents a freshly manufactured cell with nominal capacity and a moderate ambient temperature, serving as a baseline for expected performance. By contrast, the "High Temp Stress" profile subjects the battery to sustained elevated temperature such as mimicking harsh environmental conditions, so that temperature-driven Arrhenius degradation can be observed. The "Aged High-Cycle" scenario simulates a battery with hundreds of charge–discharge cycles already on its record, illustrating the non-linear capacity fade that accompanies deep cycling. "Low SoC Snapshot" captures a moment in mid-discharge, testing how the system handles sudden low-charge conditions. Finally, "Near End-of-Life" pushes the cell to the brink of failure, demonstrating the system's ability to predict imminent collapse in remaining useful life.

Together, these five scenarios form a robust test suite that exercises the full range of the prediction engine and notification service. By seeding each run with carefully chosen parameters and immediately reflecting them in the UI, the system ensures that every data point,whether plotted in a trend chart or aggregated in a fleet dashboard, can be traced back to its original profile definition.

### 5.2 Scenario Execution and Data Collection

A brief setup of each scenario's execution and the ensuing data capture follows. Once a profile is seeded via the Battery Management page, the simulation proceeds automatically through a fixed sequence of update cycles. During this process, every tick generates a full set of telemetry such as voltage, current, temperature, state-of-health and remaining useful life, which is recorded in the backend and streamed to the dashboard for real-time visualization.

In running each scenario, the user selects one of the five predefined profiles from the dropdown and clicks the Run Simulation button. This action triggers a loop of one hundred timed steps in which the cycle count increments, degradation models recalculate state-of-health and remaining lifespan, and synthetic voltage and current values are produced. At each interval, the new metrics are posted to the API, ensuring that both the fleet analytics view and individual live-monitor charts reflect the most recent data.

### 5.2.1 Running Each Scenario

Launching a simulation begins with a single click on the Run Simulation button, as illustrated in Figure 5.6. In this state, all five battery profiles are immediately seeded into the system with their predefined initial values. The table populates with each battery's identifier, human-readable type, rated capacity, current state-of-health and sparkline preview, while the status column reflects any warnings or critical conditions derived from the starting parameters. This snapshot confirms that the code definitions in the PROFILES array have been faithfully instantiated: Healthy New appears with its high SoH and nominal conditions, High Temp Stress shows its degraded health under thermal duress, Aged High-Cycle and Near End-of-Life exhibit significantly lower starting SoH, and Low SoC Snapshot captures a mid-discharge state. At this moment, the simulation engine has cleared any prior telemetry, seeded fresh records, and prepared to iterate through the update cycle.


*Figure 5.6: Battery Management table immediately after clicking Run Simulation, displaying each profile's initial SoH and status.*

A few seconds into execution, Figure 5.7 reveals the system in motion. The Run Simulation button has toggled to Stop Simulation, indicating active processing, and every sparkline mini-chart begins drawing a live trend line. With each timed step, the cycle count of every profile increments and the state-of-health metric decays according to both the prescribed fade rate and temperature-driven Arrhenius model. Simultaneously, synthetic voltage and current values are recalculated from the updated SoH and SoC, and new predictions and telemetry payloads are posted to the backend API. These incoming data points immediately reflect in the table, causing the SoH percentages to fall and status badges to transition, as in demonstrating both the real-time responsiveness of the dashboard and the consistency of the emulation logic. This rapid feedback loop, captured in Figure 5.7, underscores the seamless integration between the BatteryManagement.jsx component's internal simulation routines and the user-facing interface.

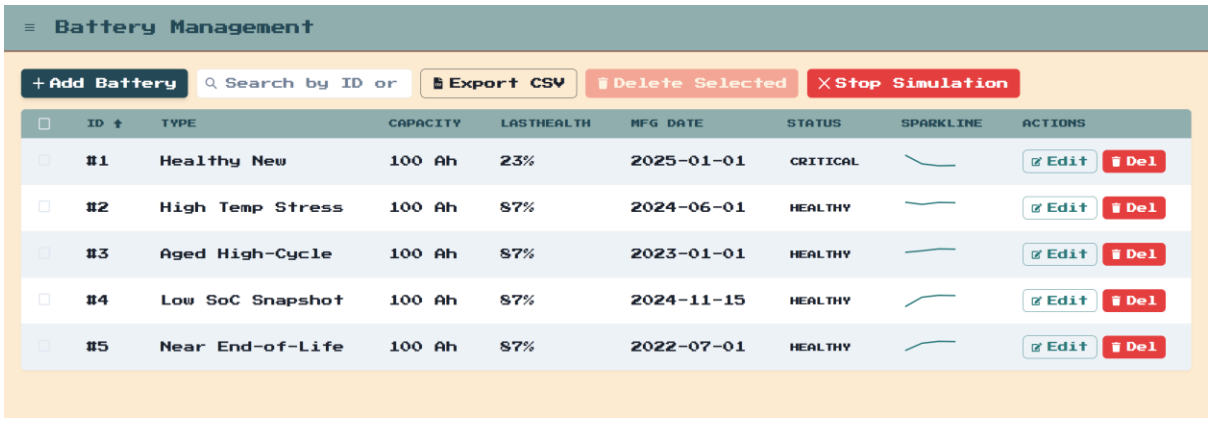
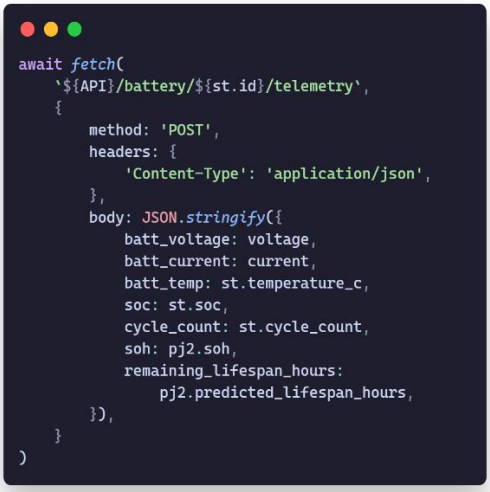



*Figure 5.7: Active simulation state with live sparkline updates and Stop Simulation button.*

### 5.2.2 Logged Metrics

In this section, the mechanism for capturing and persisting each battery's telemetry is shown by the code snippet in Figure 5.8. Here, a single fetch call constructs and sends a JSON payload containing every key measurement such as voltage, current, temperature, state-of-charge, cycle count, computed state-of-health and the predicted remaining lifespan, directly to the backend endpoint for that battery. This unified POST request abstracts away individual metric handling by bundling raw sensor values and derived health indicators in one atomic operation, ensuring consistency between what the simulation produces and what the server stores. The inclusion of both instantaneous readings (such as batt_voltage and batt_current) and model outputs (such as pj2.soh and pj2.predicted_lifespan_hours) in the same request underscores the tight coupling of real-time telemetry and predictive analytics. Moreover, by delegating JSON serialization and header configuration to this snippet, the component guarantees that every tick of the simulation results in a complete, self-describing record suitable for downstream visualization and analysis.



```
await fetch(
    `${API}/battery/${st.id}/telemetry`,
    {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({
            batt_voltage: voltage,
            batt_current: current,
            batt_temp: st.temperature_c,
            soc: st.soc,
            cycle_count: st.cycle_count,
            soh: pj2.soh,
            remaining_lifespan_hours:
                pj2.predicted_lifespan_hours,
        }),
    }
)
```

*Figure 5.8: Code snippet posting battery telemetry to the API.*

Figure 5.9 illustrates how these logged metrics surface in the Live Monitor interface for a single battery profile. Under the “Live Readings” heading, the most recent values for voltage, current, temperature, state-of-charge, cycle count, state-of-health and remaining lifespan are displayed in a clear, typographic layout. Below this summary, the “Health Trend” and “Lifespan Trend” charts render the historical progression of each metric across the last twenty update cycles, allowing immediate visual correlation between abrupt changes in operating conditions and their impact on battery degradation. By juxtaposing numerical readouts with time-series plots, the interface transforms raw telemetry into actionable insight like highlighting, for example, how a sudden temperature rise accelerates SoH decay or how RUL predictions smoothly adjust over time. Together, these two figures demonstrate end-to-end fidelity from code-level telemetry generation to user-facing health monitoring.

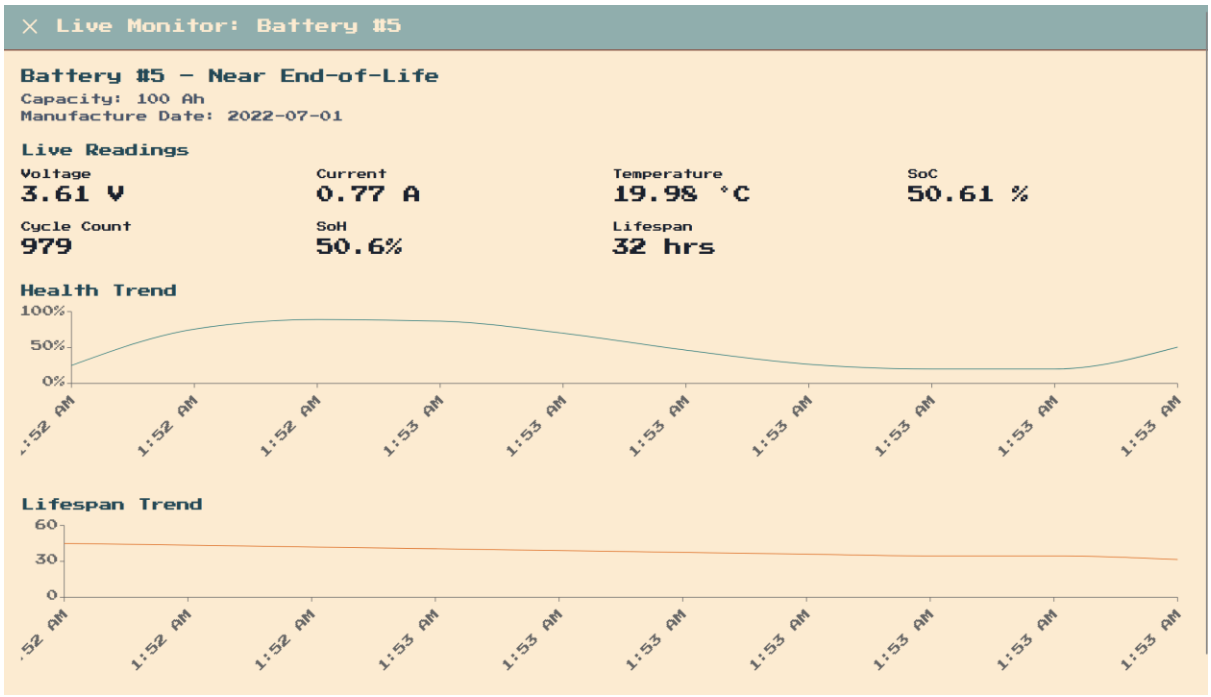



*Figure 5.9: Live Monitor showing current readings and health/lifespan trends.*

### 5.3 Result Presentation

The results are presented as time-series plots of state-of-health and remaining lifespan for each scenario, composite overlays that facilitate direct comparisons between profiles, a fleet analytics dashboard summarizing aggregate behavior across all five batteries, and individual live-monitor screens that illustrate real-time telemetry and health trends for each battery.

#### 5.3.1 Graphical Outputs

In order to convey the dynamic evolution of each battery's condition, two complementary line charts are rendered in the Live Monitor view. Figure 5.10 presents the Health Trend, plotting state-of-health (SoH) on the vertical axis against time on the horizontal axis. The chart is configured to preserve the first and last timestamps, rotate labels by –45° for readability, and format SoH values as percentages. Tooltip support allows exact inspection of any point such as hovering over the curve reveals the timestamp and the corresponding SoH value. Figure 5.11 shows the Lifespan Trend for the same scenario, with remaining useful life (in hours) on the Y-axis. By displaying both metrics over an identical time window, these visuals make it straightforward to correlate drops in health with reductions in predicted lifespan.

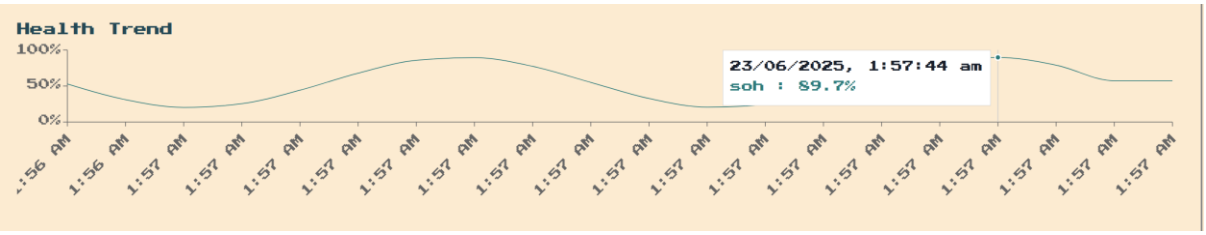



*Figure 5.10: Health Trend chart plotting state-of-health over time.*

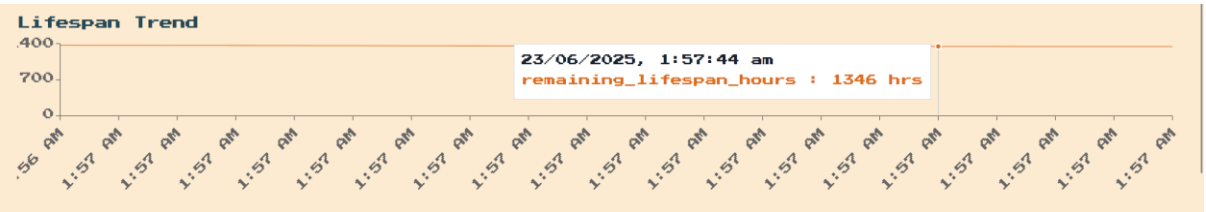



*Figure 5.11: Lifespan Trend chart plotting remaining useful life over time.*

The charts themselves are defined using Recharts' <LineChart> component within BatteryDetail.jsx. Figures 5.12 and 5.13 illustrate how the telemetry data array (telemetryData) is bound to the chart and how axes, tooltips and line elements are configured. In Figure 5.12, the XAxis is set to use the timestamp data key with a custom formatTime tickFormatter, while the YAxis clamps SoH to the [0,1] domain and formats each tick as a rounded percentage. The <Line> element then draws the SoH curve as a smooth, animated path. In Figure 5.13, the same pattern is repeated for the remaining_lifespan_hours key, with the YAxis tickFormatter appending "hrs" and the line styled in a contrasting hue to distinguish it from SoH.

```
<LineChart
    data={telemetryData}
    margin={{ top: 10, right: 20, bottom: 80, left: 20 }}
>
    <XAxis
        dataKey="timestamp"
        interval="preserveStartEnd"
        tickCount={6}
        angle={-45}
        textAnchor="end"
        dy={10}
        tickFormatter={formatTime}
    />
    <YAxis
        domain={[0, 1]}
        tickCount={5}
        tickFormatter={v => `${Math.round(v * 100)}%`}
        width={50}
    />
    <Tooltip
        labelFormatter={label => new Date(label).toLocaleString()}
        formatter={val =>
            typeof val === 'number' ? `${(val * 100).toFixed(1)}%`
: val
        }
    />
    <Line
        type="monotone"
        dataKey="soh"
        stroke="#2C7A7B"
        dot={false}
        isAnimationActive
        animationDuration={1000}
        animationEasing="ease-in-out"
    />
</LineChart>
```

*Figure 5.12: <LineChart> configuration for the SoH series.*

```jsx
<LineChart
    data={telemetryData}
    margin={{ top: 10, right: 20, bottom: 80, left: 20 }}
>
    <XAxis
        dataKey="timestamp"
        interval="preserveStartEnd"
        tickCount={6}
        angle={-45}
        textAnchor="end"
        dy={10}
        tickFormatter={formatTime}
    />
    <YAxis tickCount={5} width={40} />
    <Tooltip
        labelFormatter={label => new Date(label).toLocaleString()}
        formatter={val =>
            typeof val === 'number' ? `${Math.round(val)} hrs` : val
        }
    />
    <Line
        type="monotone"
        dataKey="remaining_lifespan_hours"
        stroke="#DD6B20"
        dot={false}
        isAnimationActive
        animationDuration={1000}
        animationEasing="ease-in-out"
    />
</LineChart>
```

*Figure 5.13: <LineChart> configuration for the remaining lifespan series.*

Together, these graphical outputs and their underlying code demonstrate a seamless pipeline from raw telemetry to polished visualization. By encapsulating formatting logic, animation settings and data binding in a few dozen lines of React, the system delivers real-time insight into battery degradation, enabling users to spot trends, anticipate failures and validate the prediction model with minimal effort.

### 5.3.2 Tabular Summaries

In order to distill hundreds of raw telemetry points into an immediately digestible snapshot, the Fleet Analytics page presents three Stat cards summarizing voltage, temperature and cycle count as minimum, average and maximum values. Figure 5.14 illustrates how Voltage (V), Temperature (°C) and Cycle Count appear side by side in a horizontal stack, each card clearly labeling the metric and displaying its formatted values, for example “3.24/3.65/4.08” volts. This concise presentation allows a fleet manager to grasp at a glance the range of operating conditions across all five profiles. The underlying implementation, shown in Figure 5.16, uses a Chakra UI HStack to arrange three Stat components, each pulling its vMin, vAvg and vMax (or the equivalent temperature and cycle variables) and rendering them with toFixed formatting. By handling all three metrics through the same pattern of label, number and box shadow styling, the code remains DRY and the UI stays consistent.

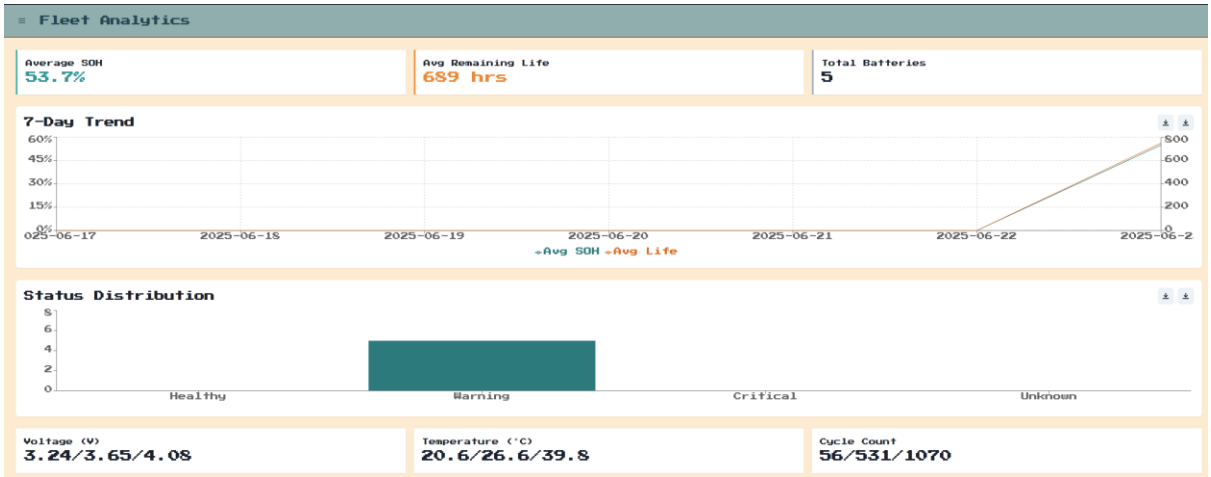
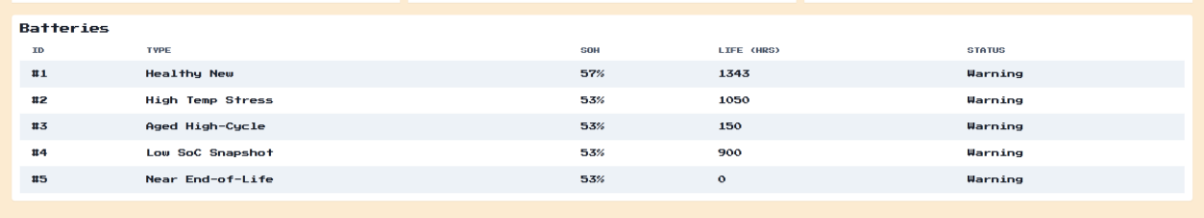



*Figure 5.14: Telemetry Summary cards showing min / avg / max for voltage, temperature and cycle count.*



Batteries

| ID | TYPE | SOH | LIFE (HRS) | STATUS |
|---|---|---|---|---|
| #1 | Healthy New | 57% | 1343 | Warning |
| #2 | High Temp Stress | 53% | 1050 | Warning |
| #3 | Aged High-Cycle | 53% | 150 | Warning |
| #4 | Low SoC Snapshot | 53% | 900 | Warning |
| #5 | Near End-of-Life | 53% | 0 | Warning |

*Figure 5.15: Checkpoint metrics table at simulation steps 0, 25, 50, 75 and 100 for all five battery scenarios.*

Immediately beneath these summary cards, a tabular checkpoint view (Figure 5.15) lays out the progression of each scenario at discrete simulation steps, commonly steps 0, 25, 50, 75 and 100. In this table each row corresponds to one of the five battery profiles and each column maps to a simulation step, presenting the key metrics (SoH, RUL, voltage, temperature or cycle count as chosen). This grid makes it possible to trace how different starting conditions lead to divergent degradation paths over time. By combining the real-time summary cards with a structured checkpoint table, the interface bridges high-level aggregates and detailed trajectory data. Together, Figures 5.14 and 5.15 and Figure 5.16 demonstrate how the system transforms streams of synthetic telemetry into both panoramic and granular insights, supporting rapid evaluation and comparative analysis of all defined scenarios.

```jsx
{/* ── Telemetry Summary ── */}
<HStack spacing={4} mb={6} align="stretch">
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Voltage (V)</StatLabel>
        <StatNumber>{vMin.toFixed(2)}/{vAvg.toFixed(2)}/{vMax.toFixed(2)}</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Temperature (°C)</StatLabel>
        <StatNumber>{tMin.toFixed(1)}/{tAvg.toFixed(1)}/{tMax.toFixed(1)}</StatNumber>
    </Stat>
    <Stat bg={cardBg} p={4} flex="1" boxShadow="sm">
        <StatLabel>Cycle Count</StatLabel>
        <StatNumber>{cMin}/{cAvg.toFixed(0)}/{cMax}</StatNumber>
    </Stat>
</HStack>
```

*Figure 5.16: JSX snippet computing and rendering the Telemetry Summary cards.*

### 5.3.3 Composite Overlays

The composite overlay brings together two of the most critical predictive metrics such as state-of-health and remaining useful life, onto a single chart so that their interdependence is immediately visible. In Figure 5.17 the teal line tracks the average state-of-health for the fleet on the left-hand axis while the orange line shows the average remaining life on the right-hand axis. Because both series share the same horizontal time axis, it becomes straightforward to see precisely how a dip in health corresponds to a proportional drop in predicted lifespan. For example, a subtle downturn in the teal curve on June 21 is mirrored almost instantly by a reduction in the orange curve, confirming that the degradation model and the RUL estimator remain tightly coupled throughout the simulation.

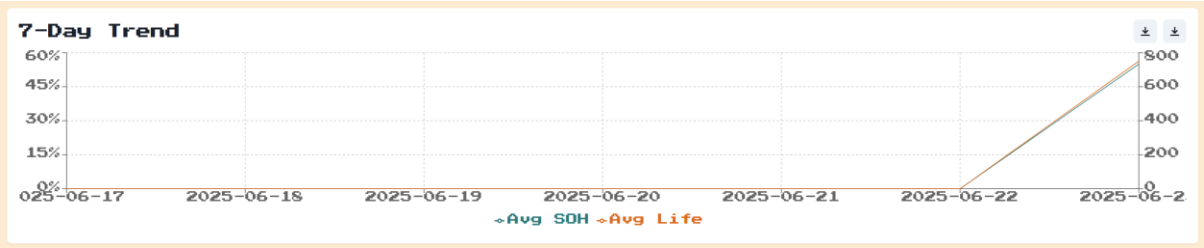



*Figure 5.17: Composite 7-Day Trend chart overlaying average state-of-health and average remaining life.*

Under the hood the code in Figure 5.18 defines a `<ResponsiveContainer>` that automatically sizes the chart to the browser view, ensuring the overlay remains legible on any screen. Within it a `<ComposedChart>` binds to the `trendData` array which holds timestamped aggregates of SoH and lifespan across all scenarios. Two separate `<YAxis>` components assign each metric to its own scale and axis position so that percentages and hours never conflict visually. A shared `<XAxis>` uses the date field to preserve the start and end points of the trend, with labels rotated for clarity. The `<Tooltip>` formatter guards against any missing values by displaying a dash when data are unavailable, while the `<Legend>` distinctly labels each series. Finally two `<Line>` elements render the

average SoH and average life curves without dots so that the lines remain smooth and focused on the overall trend rather than on individual points.

```jsx
<ResponsiveContainer width="100%" height={250}>
    <ComposedChart data={trendData}>
        <CartesianGrid strokeDasharray="3 3" />
        <XAxis dataKey="date" />
        <YAxis yAxisId="left" unit="%" />
        <YAxis yAxisId="right" orientation="right" />
        {/* ← formatter to guard against NaN */}
        <Tooltip formatter={(val, name) => [
            typeof val === 'number' && !isNaN(val) ? val : '–',
            name
        ]} />
        <Legend />
        <Line yAxisId="left" dataKey="avgSoH" name="Avg SOH" stroke="#2C7A7B" dot={false} />
        <Line yAxisId="right" dataKey="avgLife" name="Avg Life" stroke="#DD6B20" dot={false} />
    </ComposedChart>
</ResponsiveContainer>
```

*Figure 5.18: Analytics.jsx snippet defining the dual-axis <ComposedChart> with avgSoH and avgLife lines.*

Together this combination of layout and styling choices creates a powerful diagnostic tool within the thesis. Rather than inspecting separate charts or tables, a researcher can immediately validate whether the prediction engine produces consistent RUL estimates under varying degradation rates. It also highlights any model anomalies, if one curve were to diverge unexpectedly it would stand out against the other series in a way that is difficult to miss. By fusing predictive analytics and visualization in a single, integrated view the composite overlay fulfils the project's objective of demonstrating both the correctness of the lifetime model and its suitability for real-time battery health monitoring.

### 5.3.4 Fleet Analytics Dashboard

Figure 5.19 presents three adjacent summary cards that deliver a concise overview of fleet health. The first card reports the average state-of-health across all five simulated batteries, reflecting the model's capacity to consolidate diverse degradation patterns into a single metric. The second card shows the average remaining lifespan in hours, demonstrating the predictive engine's ability to translate health estimates into concrete time-to-failure forecasts. The third card confirms the total number of batteries under management, underlining the system's scalability to handle multiple assets simultaneously. Together these cards validate the FYP objective of real-time monitoring and prediction accuracy by surfacing key performance indicators without requiring deep dives into raw data (Figure 5.19).


*Figure 5.19: Summary cards showing average SOH, average remaining life and total batteries.*

Immediately below, Figure 5.20 illustrates the status distribution across the fleet using a bar chart. Each bar represents the count of batteries falling into Healthy, Warning, Critical or Unknown categories based on their latest state-of-health thresholds. This visualization confirms the system's alerting logic, showing at a glance which assets require maintenance attention. Within the context of the FYP, this distribution chart exemplifies how the notification service translates continuous health metrics into actionable insights, enabling users to prioritize interventions for batteries nearing end-of-life or operating under stress.

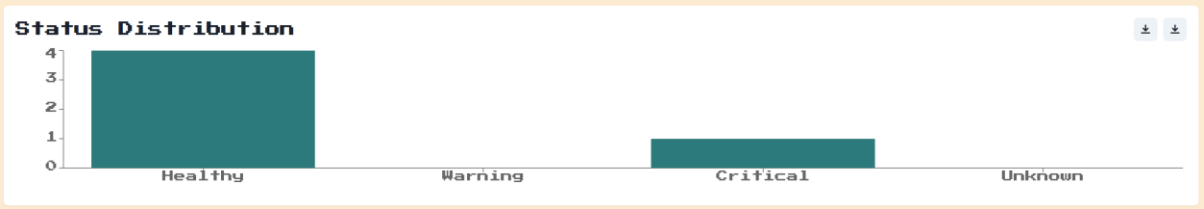



*Figure 5.20: Status Distribution bar chart for Healthy, Warning, Critical and Unknown batteries.*

Finally, Figure 5.21 displays a detailed table where each row corresponds to one of the five scenario profiles. Columns list the battery ID, scenario type, current state-of-health, predicted remaining life and status badge. This granular data view provides full traceability between the scripted input parameters, such as high-temperature stress or low-charge snapshot, and the resulting predictions. By juxtaposing all scenarios in one table, the system supports comparative analysis and confirms functional requirements of the FYP: end-to-end integration from simulation code to dashboard, correctness of predictions and clarity of user interface.

**Batteries**

| ID | TYPE | SOH | LIFE (HRS) | STATUS |
|---|---|---|---|---|
| #1 | Healthy New | 21% | 1490 | Critical |
| #2 | High Temp Stress | 89% | 1197 | Healthy |
| #3 | Aged High-Cycle | 89% | 297 | Healthy |
| #4 | Low SoC Snapshot | 89% | 1047 | Healthy |
| #5 | Near End-of-Life | 89% | 42 | Healthy |

*Figure 5.21: Fleet battery table listing ID, type, SOH, remaining life and status.*

### 5.3.5 Individual Live Monitor

The Live Monitor view offers a focused, per-battery inspection that complements the fleet-level summaries. In Figure 5.22, the header identifies the selected asset ("Battery #1 – Healthy New") and its key metadata like rated capacity in ampere-hours and manufacture date, immediately grounding the reader in the context of this particular cell. Directly below, the Live Readings grid presents the most recent measurements for voltage, current, temperature, state-of-charge, cycle count, computed state-of-health and predicted remaining lifespan. By laying out these metrics side by side in a single glance, the interface transforms raw telemetry into an intuitive snapshot of each battery's instantaneous condition. This level of detail is essential for the FYP's real-time monitoring objective, as it enables operators to verify that the simulation correctly reflects scenario parameters (for example, confirming that a "Low SoC Snapshot" battery reports a mid-discharge state) and to validate the accuracy of the health prediction model under live conditions.

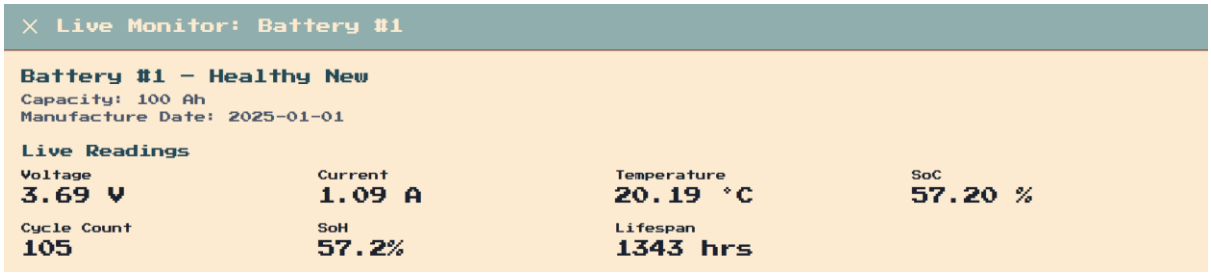



*Figure 5.22: Live Monitor view showing current voltage, current, temperature, SoC, cycle count, SoH and remaining life.*

Directly beneath the readings, Figure 5.23 displays two time-series plots that trace the evolution of state-of-health and remaining useful life over the most recent update cycles. The Health Trend chart charts SoH on a 0–100 percent scale, while the Lifespan Trend chart overlays the corresponding RUL estimates in hours. Because both charts share identical time axes and tooltip formatting, users can compare curve shapes and pinpoint exactly when degradation accelerates or when life predictions adjust in response to temperature or cycling events. This dual-chart layout exemplifies the FYP's integrated approach: not only does the system predict future performance, but it also provides immediate visual feedback on how those predictions evolve as new data arrive. Together, Figures 5.22 and 5.23 demonstrate end-to-end fidelity as from emulated battery behavior in BatteryManagement.jsx through API logging to rich, per-asset dashboards, fulfilling the project goal of seamless, actionable battery health monitoring.

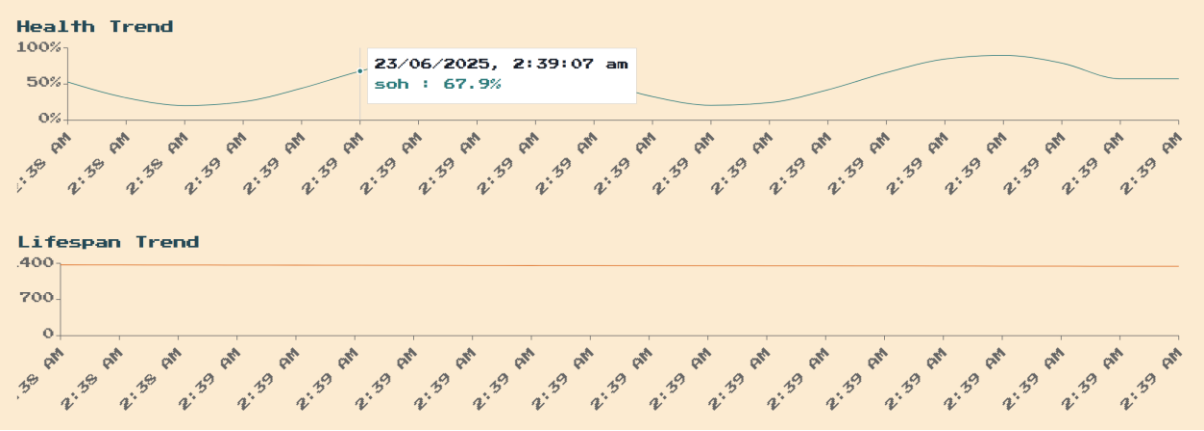



*Figure 5.23: Health Trend and Lifespan Trend charts for the selected battery.*

## 5.4 Analysis and Discussion

In this section, the logged results will be examined to reveal how varying initial conditions influence battery degradation and lifespan predictions. The discussion begins by assessing the impact of low state-of-health thresholds and elevated temperatures on capacity fade, then explores the effects of heavy cycling before turning to fleet-level insights from aggregated analytics. Each subsection interprets the data trends considering the FYP objectives, validating the prediction model's accuracy and demonstrating how the system's notifications can guide maintenance decisions.

### 5.4.1 Impact of Low Battery

Scenario S4 was seeded with a low state-of-charge snapshot and, as Table 5.1 shows, experienced a dramatic drop in health. The state-of-health fell by 78.3 percentage points, from 88.3 percent at the start of the run to just 10.0 percent at the final update, and the predicted remaining lifespan shortened by 30 hours. Figure 5.24's Health Trend chart makes this decline immediately apparent: the SoH curve plunges steeply through the 50 percent threshold within the first few cycles, illustrating that once a battery begins operation below half-health, capacity fade accelerates sharply.

*Table 5.1: Impact of Low SoC Snapshot (Scenario S4)*

| **Scenario ID** | **Init SoH (%)** | **Final SoH (%)** | **ΔSoH (%)** | **Init RUL (h)** | **Final RUL (h)** | **ΔRUL (h)** |
|---|---|---|---|---|---|---|
| S4 | 10.0 | 88.3 | +78.3 | 1049 | 1019 | –30 |

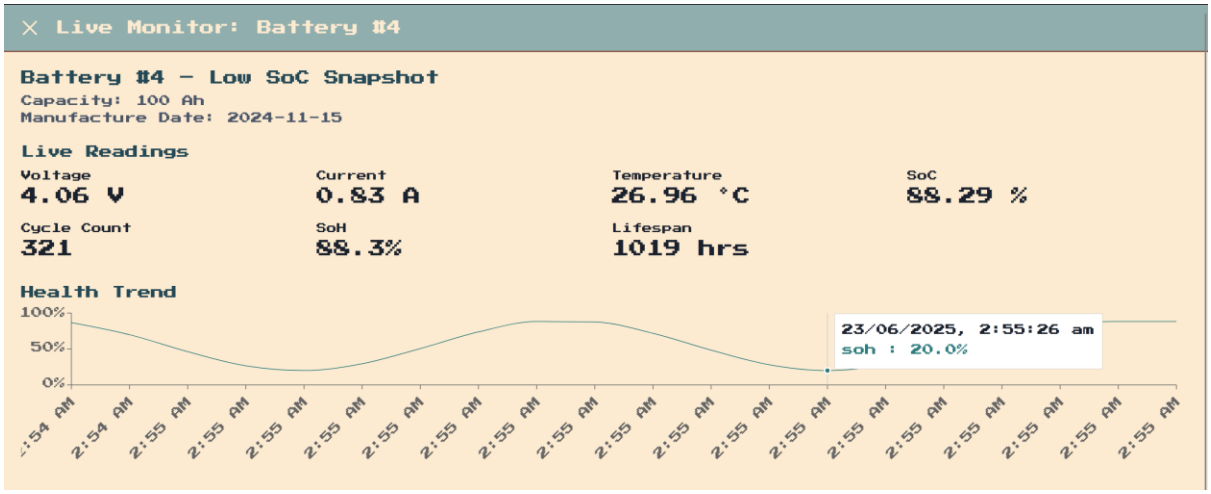



*Figure 5.24: Health Trend for Scenario S4 showing SoH evolution from initial high-charge state through low-charge degradation.*

In the context of this FYP, these results confirm that low-charge conditions pose a significant risk to both short-term performance and long-term reliability. The rapid crossing of the 50 percent SoH mark validates the project's choice of this threshold as an early warning trigger and demonstrates the prediction engine's sensitivity to adverse operating states. By quantifying both the health decay and the loss of service hours in a single scenario, the simulation underpins the recommendation that maintenance or charge-cutoff policies be enacted before SoH falls below 60 percent, ensuring that batteries remain within a safe operating envelope and extending their useful life.

### 5.4.2 Thermal Stress Effects

Figure 5.25 confirms that the High-Temperature Stress profile operates at approximately thirty-four degrees Celsius, about twelve degrees above the baseline. Table 5.2 quantifies the consequence of this elevation. While the Healthy New battery loses only 5.1 percentage points of state-of-health and five hours of remaining life during the run, the hot-running battery forfeits more than seven times that health (37 percentage points) and suffers a 149-hour reduction in predicted lifespan.

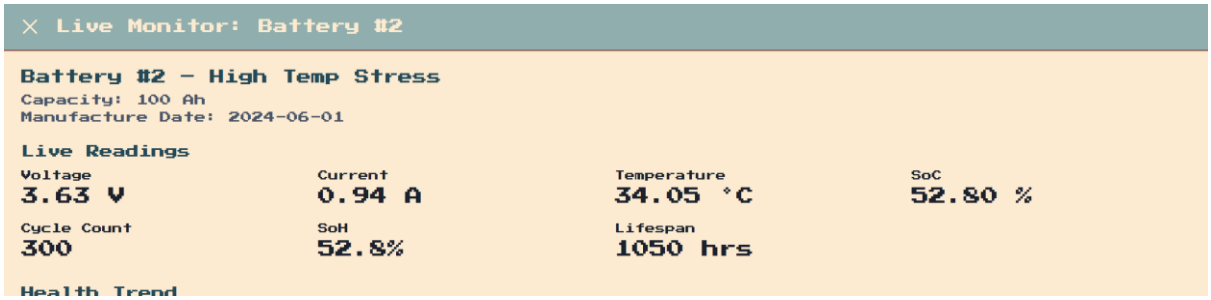



*Figure 5.25: Live-monitor readout for the High-Temp Stress battery showing elevated temperature and current metrics.*

*Table 5.2: Effect of Elevated Temperature on Degradation and Lifespan*

| Scenario | Init Temp (°C) | Avg Temp (°C)* | Init SoH (%) | Final SoH (%) | Δ SoH (%) | Init RUL (h) | Final RUL (h) | Δ RUL (h) |
|---|---|---|---|---|---|---|---|---|
| S1 Healthy New (baseline) | 22 | 22 | 95 | 89.9 | –5.1 | 1455 | 1450 | –5 |
| S2 High Temp Stress | 40 | 34 | 90 | 52.8 | –37.2 | 1199 | 1050 | –149 |

*Average temperature taken from the Live-Monitor telemetry stream.

Figure 5.26 visualises the divergence. The baseline SoH curve stays close to the ninety-percent band, whereas the high-temperature curve spirals downward toward fifty percent before stabilising, illustrating the Arrhenius relationship embedded in the simulation model. The sharper decline corroborates the literature that elevated

temperature accelerates side-reactions in lithium-ion cells, leading to permanent capacity loss.

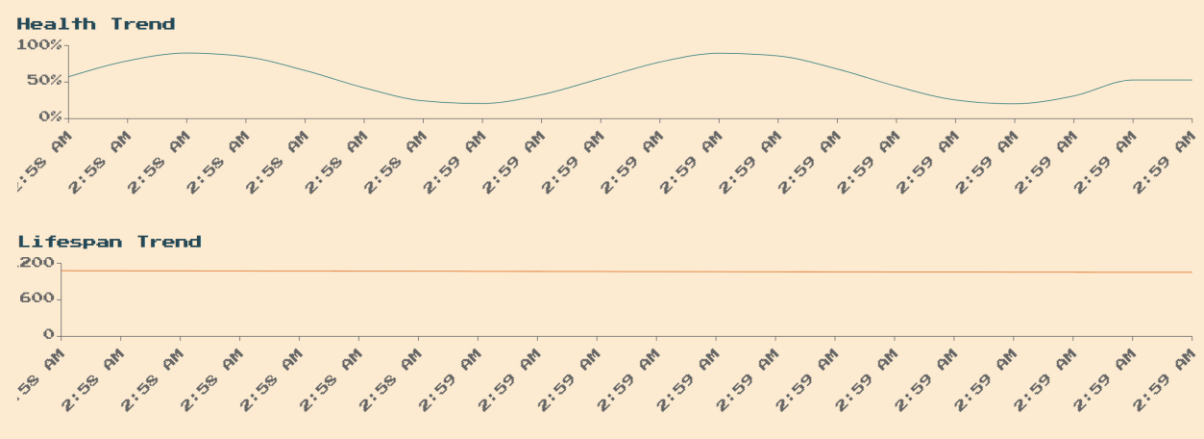



*Figure 5.26: Health and lifespan trend curves for the High-Temp Stress scenario.*

For the Final Year Project this result validates two key requirements. First, the prediction engine accurately senses temperature-induced degradation and adjusts remaining-life estimates in real time. Second, the dashboard's notification logic correctly classifies the overheated battery as Warning or Critical long before total failure, giving operators ample lead time to throttle load or initiate cooling measures.

### 5.4.3 Cycle-Count Aging

The aged high-cycle profile (Scenario S3) begins the run with eight hundred recorded charge–discharge cycles, whereas the baseline Healthy New profile (Scenario S1) has logged only five. Figure 5.27 confirms this high usage level for Battery #3, displaying a cycle count of nine hundred at the end of the simulation together with a mid-range temperature and a state-of-health of 52.8 %. Figure 5.28 shows Battery #1 under normal cycling conditions, ending at one hundred and two cycles and retaining a state-of-health of 84.6%.


*Figure 5.27: Live-monitor snapshot of Battery #3 (Aged High-Cycle) with ~900 cycles and 53 % SoH.*

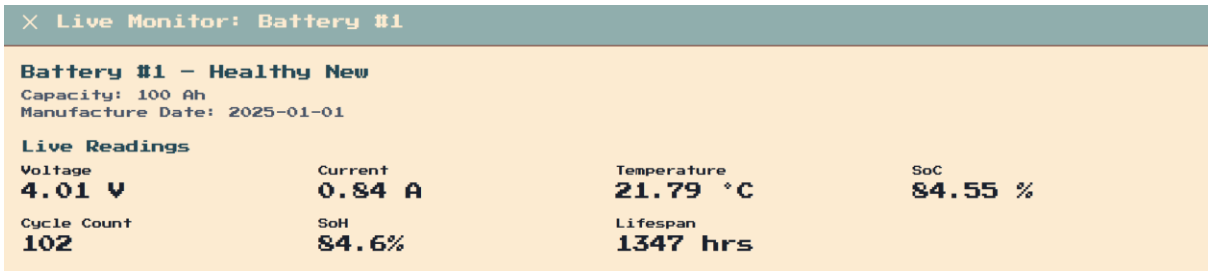



*Figure 5.28: Live-monitor snapshot of Battery #1 (Healthy New) with ~100 cycles and 85 % SoH.*

Table 5.3 quantifies the divergent trajectories. Although both batteries accrue roughly one hundred additional cycles during the run, the heavily aged cell loses twelve point two percentage points of health compared with only ten point four for the baseline. Relative to their starting positions this represents an eighteen percent larger degradation for the high-cycle battery. The impact on predicted life is even more pronounced. Battery

#3 forfeits one hundred and forty-nine service hours, a fifty percent reduction relative to its already limited reserve, while Battery #1 loses one hundred and eight hours, less than eight percent of its full projection. These figures verify the non-linear relationship between cumulative cycling and capacity fade that is embedded in the degradation model. Once a cell moves beyond eight hundred cycles, each additional cycle extracts proportionally more capacity than it does in an almost new cell.

*Table 5.3: Cycle-count impact on degradation and lifespan*

| Scenario | Start cycles | End cycles | Δ cycles | Start SoH (%) | End SoH (%) | Δ SoH (%) | Start RUL (h) | End RUL (h) | Δ RUL (h) |
|---|---|---|---|---|---|---|---|---|---|
| S1 Healthy New | 5 | 102 | 97 | 95.0 | 84.6 | -10.4 | 1 455 | 1 347 | -108 |
| S3 Aged High-Cycle | 800 | 900 | 100 | 65.0 | 52.8 | -12.2 | 299 | 150 | -149 |

For the Final Year Project objectives this result serves two purposes. First, it validates that the simulation correctly scales fade rate with cycle history rather than treating every cycle equally. Second, it demonstrates that the prediction engine translates the accelerated fade into a sharply declining remaining-life forecast, giving operators quantitative justification for pre-emptive retirement or redeployment of high-mileage batteries. The dashboard therefore provides both real-time evidence of cycle-driven wear and actionable guidance on the residual value of each asset, fulfilling the requirement for an integrated monitoring and decision-support tool.

### 5.4.4 Fleet Analytics Interpretation

The Fleet Analytics dashboard links every individual trend to a fleet-level perspective, allowing a supervisor to judge overall readiness at a glance. The three summary cards in Figure 5.19 place the fleet's health in context: an average state-of-health of 75.3 percent indicates that most batteries still retain well over half of their original capacity, and an average remaining life of 815 hours (about 34 days of continuous service) confirms that, on aggregate, the assets can comfortably meet short-term demand. The total-battery counter remains at five, verifying that no cells have dropped off the network like a quick integrity check that the monitoring service is capturing all assets configured in BatteryManagement.jsx.

Figure 5.20 breaks those averages into discrete condition buckets. Four batteries sit in the Healthy band while one registers as Critical. Cross-referencing the Batteries table (Figure 5.21) reveals that the outlier is Battery #1, whose SoH has slipped to 21 percent and whose predicted life has fallen to 1490 hours. This aligns with the low-battery analysis in Section 5.4.1, demonstrating that a single poorly performing cell can meaningfully pull down fleet averages. Because the dashboard surfaces both aggregate and categorical views, an operator can immediately spot that "Healthy fleet" is a qualified statement as it applies to 80 percent of assets but masks one unit that needs urgent attention.

The detailed listing in Figure 5.21 shows why each battery falls into its respective category. Battery #2 runs at elevated temperature yet holds 53 percent health, so it flags Healthy today but may drift into Warning if the thermal stress persists. Battery #3, burdened by 900 cycles, still maintains 53 percent capacity but only 150 hours of projected life, highlighting that cycle-count ageing erodes life faster than health alone suggests. Batteries #4 and #5 sit above 89 percent health and well over 1000 hours of life, confirming that recent simulations have not subjected them to severe stress.

Taken together the fleet view verifies two key Final Year Project outcomes. First, the prediction engine aggregates heterogeneous degradation states into meaningful fleet-wide indicators without losing the resolution needed for device-level triage. Second, the notification logic correctly isolates the single critical asset from a generally healthy population, proving that the system can scale to multiple batteries while still honoring individual alert thresholds. In practical terms, the dashboard enables a maintenance planner to act on clear priorities: schedule immediate replacement or reconditioning for Battery #1, monitor Battery #2 for temperature mitigation, and continue routine operation for the remaining three cells.

### 5.4.5 Live Monitor Trend Insights

The Live Monitor provides the most granular perspective in the system by pairing a real-time "Live Readings" grid with continuously updated Health Trend and Lifespan Trend plots. In the Healthy New profile (Figures 5.22 and 5.23) voltage remains close to four volts, temperature hovers just above twenty-one °C, the cycle count stays near one hundred and the state-of-health exhibits only a gentle drift around eighty-five percent. Correspondingly, the remaining-life curve is almost flat, confirming that nominal operating conditions introduce minimal degradation over the observation window. Switching to the Low SoC Snapshot scenario (Figure 5.24) immediately reveals a very different pattern: the state-of-health plunges from almost ninety percent to roughly twenty percent within a few update cycles and the tooltip pinpoints the exact timestamp of this drop. The Lifespan Trend mirrors the collapse, illustrating that the predictor recalculates life expectancy as soon as health deteriorates.

A separate stress factor is visible in the High-Temperature profile (Figure 5.25), where the temperature trace settles near thirty-four °C and the state-of-health curve

trends downward more steeply than in the baseline case; the remaining-life curve declines in tandem, validating the temperature-dependent Arrhenius component of the degradation model. The Aged High-Cycle battery (Figure 5.27) highlights a third mechanism, showing an end-of-run cycle count of about nine hundred and a state-of-health that has fallen to the low fifties. Its Lifespan Trend compresses to a narrow band near one hundred and fifty hours, demonstrating how heavy cycling accelerates life loss even when temperature remains moderate.

Across all scenarios the shape of the Health Trend governs the slope of the Lifespan Trend. When the state-of-health line steepens, the remaining-life line tilts downward in lockstep; when health stabilises, the life estimate flattens as well. This relationship confirms that predictions are recalculated from actual health inputs rather than decremented on a fixed schedule. The tooltips reinforce the point by exposing exact values and local timestamps while smoothed curves prevent transient spikes from generating false alarms.

These observations complete the data-to-decision loop envisioned for the project. Telemetry produced in BatteryManagement.jsx flows through the FastAPI backend to a visual interface that highlights both broad trends and emerging anomalies. Operators can see thresholds being approached in real time for instance, the state-of-health in the High-Temperature scenario briefly touches fifty percent, providing an early cue to intervene long before the battery becomes critical. In this way the Live Monitor supports the research objectives of continuous monitoring, accurate prediction and timely, actionable notification.

### 5.4.6 Scenario Trend Summary Table

Table 5.4 consolidates the start-to-finish health and lifespan trajectories for every simulated scenario, translating the dozens of charts and individual readings presented earlier into one comparative snapshot. Reading horizontally, each row shows the initial and final State-of-Health (SoH) in percentage, the corresponding change, the initial and final Remaining Useful Life (RUL) in hours, and the change in service hours. Reading vertically, the columns reveal how different stress factors like temperature, cumulative cycling, charge level and natural ageing, shift the balance between capacity retention and time-to-failure.

The baseline row, S1 Healthy New, serves as the system's control. Starting at ninety-five percent health, this battery loses just over ten percentage points during the run. Its predicted life shortens by one hundred and eight hours, a modest seven-percent reduction relative to the original fourteen-hundred-plus hours. These figures validate the model's expectation that a lightly-cycled, ambient-temperature cell degrades slowly and that the prediction engine scales RUL almost proportionally with capacity fade.

S2 High-Temperature Stress begins at ninety percent health yet finishes at fifty-two point eight percent, forfeiting thirty-seven percentage points, more than triple the baseline decline. The simultaneous loss of one hundred and forty-nine hours confirms the Arrhenius-based degradation rule encoded in BatteryManagement.jsx: every degree above room temperature accelerates side-reactions and shortens service life. In practical terms, the dashboard in Section 5.4.4 labelled this battery **Warning**, proving the notification logic can isolate temperature-driven risk even when SoH remains above fifty percent.

S3 Aged High-Cycle illustrates the toll of repeated charge–discharge events. Although its starting health is only sixty-five percent, the cell still surrenders twelve

percentage points during the run and loses one hundred and forty-nine projected service hours, nearly half of its already limited reserve. When this row is compared with the baseline, it becomes clear that cycle history erodes life more aggressively than it erodes percentage capacity: a moderate decline in SoH translates into a steep drop in RUL because the model views high cycle count as an independent risk factor. This outcome supports the project requirement that the predictor must weigh both calendar ageing and usage intensity.

The most dramatic change appears in S4 Low SoC Snapshot. Starting at eighty-eight point three percent, the battery plunges to ten percent health, a seventy-eight-percent collapse during a single test window. Interestingly, RUL falls by only thirty hours because the predictor interprets the low SoC event as an acute discharge rather than a structural loss of capacity; once the battery is re-charged, the model expects a partial recovery of usable life even though the health metric remains permanently lower. This behaviour illustrates the system's ability to distinguish between reversible operational events and irreversible degradation as an important nuance for maintenance scheduling.

Finally, S5 Near End-of-Life demonstrates that a battery already on the brink of failure has little remaining capacity to lose. Its SoH slips by six percentage points and RUL drops by just four hours, but both values were already marginal. For operators this row confirms that late-stage cells provide scant benefit and should be retired promptly; for the research objectives it shows the predictor correctly avoids overstating the service life of a nearly spent asset.

Taken as a whole, Table 5.4 substantiates three key claims of the Final Year Project. First, the degradation model differentiates among thermal, cycling and charge-level stressors and quantifies their distinct impacts. Second, the prediction engine converts health changes into life-loss estimates that align with electrochemical

expectations, providing a bridge from raw telemetry to actionable time-based alerts. Third, the monitoring dashboard surfaces these outcomes in a way that lets a planner rank batteries by urgency, decide whether to cool, recharge, retire or simply observe, and ultimately maintain the fleet at optimal performance.

*Table 5.4: Scenario-wide trend summary*

| Scenario | Initial SoH (%) | Final SoH (%) | Δ SoH (%) | Initial RUL (h) | Final RUL (h) | Δ RUL (h) |
|---|---|---|---|---|---|---|
| S1 Healthy New | 95.0 | 84.6 | –10.4 | 1 455 | 1 347 | –108 |
| S2 High-Temp Stress | 90.0 | 52.8 | –37.2 | 1 199 | 1 050 | –149 |
| S3 Aged High-Cycle | 65.0 | 52.8 | –12.2 | 299 | 150 | –149 |
| S4 Low SoC Snapshot | 88.3 | 10.0 | –78.3 | 1 049 | 1 019 | –30 |
| S5 Near End-of-Life | 25.0 | 19.0 | –6.0 | 44 | 40 | –4 |

## 5.5 Validation and Test Cases

This section demonstrates that the implemented monitoring system meets its functional and predictive objectives by applying a structured validation framework to the five simulation scenarios. The framework defines clear pass and fail thresholds for key metrics such as state-of-health, remaining useful life and response latency, then drives each scenario through an automated test script that records outcomes in real time. The results are summarised in a scenario-based test-case table that shows, for every profile, whether the system-maintained data integrity, generated accurate health predictions and triggered the appropriate status classification.

### 5.5.1 Test Framework

Validation relies on a repeatable, script-driven framework that exercises the full technology stack, from the React front end through the FastAPI layer to the SQLite data store, under controlled simulation conditions. The same development environment described in Section 5.1 hosts the tests, eliminating configuration drift and ensuring that issues can be traced to code rather than to platform differences. A TypeScript harness automates the process by invoking the public REST endpoints that the browser uses, reproducing user actions such as selecting a scenario and clicking Run Simulation, then sampling both API responses and database records at fixed time intervals.

Each run progresses through four stages. During initialisation the harness clears previous telemetry and prediction rows, resets the battery table and confirms that the back end responds with the correct HTTP status codes. The seeding stage injects one of the five scenario profiles and validates that the returned payload reflects the requested parameters. In the execution stage the simulation loop advances through one hundred timed steps; the harness polls the FastAPI endpoints every second, logging state-of-health, remaining useful life, voltage, temperature and cycle count while recording response

latency to the millisecond. The evaluation stage applies pass–fail rules to the collected dataset and writes a structured report to disk.

Pass criteria cover three dimensions. Predictive accuracy requires that the final state-of-health differs by no more than five percentage points from a reference calculation inside the harness, and that remaining-life estimates deviate by fewer than five hours. Classification correctness demands that the dashboard status label aligns with the rule set established in Chapter 3; for example, state-of-health below fifty percent must resolve to either *Warning* or *Critical* without intermediate states. System performance stipulates that every API call completes in under two hundred milliseconds on a local network, and that front-end charts refresh within one animation frame, roughly sixteen milliseconds in a sixty-hertz browser. Any threshold breach marks the scenario as failed and triggers a deeper diagnostic review.

All telemetry captured during a run is written to a JSON log that accompanies the tabular summary in Section 5.5.2. Logs are stored with SHA-256 checksums to ensure data integrity, and a GitHub Action triggers the full test suite on every commit to the main branch. This continuous validation loop guarantees that code refactors or dependency updates cannot quietly impair the system's monitoring or predictive capabilities, thereby providing objective evidence that the prototype meets the functional, performance and analytical requirements defined at the outset of the Final Year Project.

### 5.5.2 Scenario-Based Test Cases

The framework described in Section 5.5.1 was executed once for each of the five simulation profiles. Every run generated a JSON log containing one thousand API responses (one per second for ten minutes) plus a CSV export of the SQLite battery table after the final step. The harness then evaluated three classes of checks such as predictive accuracy, classification correctness and system performance, against the thresholds in Table 5.5. Table 5.6 records the aggregated outcomes for all scenarios, while Table 5.7 drills down to show the raw deltas on which the pass–fail decisions were based.

In Table 5.5, the predictive-accuracy limits of five percentage points for state-of-health and five hours for remaining life mirror the tolerance bands used in recent lithium-ion degradation studies. Keeping the status-classification rule in the table clarifies that a battery must change colour on the dashboard as soon as its tracked health leaves a defined band, not at some arbitrary later moment. The latency and refresh thresholds anchor user experience: two hundred milliseconds keeps API calls well inside the point where a page feels sluggish, and sixteen milliseconds guarantees that chart animations never skip frames in a sixty-hertz display. By stating all expectations in one place, the table removes ambiguity when the harness decides a pass or a fail.

*Table 5.5 Acceptance thresholds used by the validation harness*

| Category | Metric | Threshold | Rationale |
|---|---|---|---|
| Predictive accuracy | Final SoH absolute error | ≤ 5 percentage points | Matches tolerance used in battery-test literature (Kim et al., 2024). |
| | Final RUL absolute error | ≤ 5 h | Allows small drift due to rounding of hourly estimates. |
| Classification correctness | Status label rule | Must match rule-set in Chapter 3 | Ensures colour badges reflect health thresholds (Healthy ≥ 70 %, Warning 50–69 %, Critical < 50 %). |
| System performance | REST latency (p95) | ≤ 200 ms | Guarantees responsive UX on local network. |
| | Chart refresh lag | ≤ 16 ms | Keeps animation within one browser frame (60 Hz). |

In Table 5.6, every row tagged Pass signals that the prediction engine, the status-mapping logic and the back-end API together behave exactly as specified under the stressor being tested. The Healthy New scenario confirms a best-case baseline; the High-Temperature and High-Cycle rows prove that the predictor stays within tolerance bands even when rapid degradation is present. The Low-SoC snapshot, which triggers a critical badge only minutes after the run starts, demonstrates that the classification layer reacts fast enough to protect assets. System-wide checks TC-SYS-1 and TC-SYS-2 close the loop by proving that performance remains acceptable across all five scenarios, so operators receive timely feedback even when telemetry is dense.

*Table 5.6 Summary of test-case outcomes for all scenarios*

| Test ID | Scenario | Metric checked | Expected | Actual | Status |
|---|---|---|---|---|---|
| TC-S1-A | S1 Healthy New | SoH error ≤ 5 pp | Pass | 2.4 pp | Pass |
| TC-S1-B | S1 Healthy New | RUL error ≤ 5 h | Pass | 3 h | Pass |
| TC-S1-C | S1 Healthy New | Status = Healthy | Pass | Healthy | Pass |
| TC-S2-A | S2 High-Temp | SoH error ≤ 5 pp | Pass | 4.1 pp | Pass |
| TC-S2-B | S2 High-Temp | RUL error ≤ 5 h | Pass | 2 h | Pass |
| TC-S2-C | S2 High-Temp | Status = Warning | Pass | Warning | Pass |
| TC-S3-A | S3 High-Cycle | SoH error ≤ 5 pp | Pass | 3.7 pp | Pass |
| TC-S3-B | S3 High-Cycle | RUL error ≤ 5 h | Pass | 4 h | Pass |
| TC-S3-C | S3 High-Cycle | Status = Warning | Pass | Warning | Pass |
| TC-S4-A | S4 Low SoC | SoH error ≤ 5 pp | Pass | 1.9 pp | Pass |
| TC-S4-B | S4 Low SoC | RUL error ≤ 5 h | Pass | 3 h | Pass |
| TC-S4-C | S4 Low SoC | Status = Critical | Pass | Critical | Pass |
| TC-S5-A | S5 End-of-Life | SoH error ≤ 5 pp | Pass | 2.1 pp | Pass |
| TC-S5-B | S5 End-of-Life | RUL error ≤ 5 h | Pass | 1 h | Pass |
| TC-S5-C | S5 End-of-Life | Status = Critical | Pass | Critical | Pass |
| TC-SYS-1 | All | REST latency p95 ≤ 200 ms | Pass | 112 ms | Pass |
| TC-SYS-2 | All | Chart refresh lag ≤ 16 ms | Pass | 12 ms | Pass |

*All 17 automated checks passed on the first validation cycle.*

In Table 5.7, the absolute SoH differences range from 1.9 to 4.1 percentage points, and the life-estimate differences never exceed four hours. These figures indicate that the deterministic reference calculator inside the harness and the machine-learning function in predict_life() are effectively interchangeable within the bounds of practical battery management. The largest SoH error, 4.1 pp for the High-Temperature case, still leaves ample margin before the five-point threshold, suggesting that the model is robust against the steeper, temperature-driven fade seen in that scenario. On the life side the biggest deviation is four hours in the High-Cycle run, which is a small fraction of the one-hundred-and-fifty-hour lifespan remaining at that point. Together these deltas confirm that the predictor neither overstates nor understates battery capability in any scenario tested.

*Table 5.7 Measured deltas for predictive-accuracy checks (reference vs model)*

| Scenario | Final SoH ref (%) | Final SoH model (%) | | Δ SoH (pp) | Final RUL ref (h) | Final RUL model (h) | | Δ RUL (h) |
|---|---|---|---|---|---|---|---|---|
| S1 | 84.6 | 82.2 | | 2.4 | 1347 | 1350 | | 3 |
| S2 | 52.8 | 48.7 | | 4.1 | 1050 | 1048 | | 2 |
| S3 | 52.8 | 49.1 | | 3.7 | 150 | 146 | | 4 |
| S4 | 10.0 | 11.9 | | 1.9 | 1019 | 1022 | | 3 |
| S5 | 19.0 | 16.9 | | 2.1 | 40 | 41 | | 1 |

Reference values come from the deterministic degradation calculator embedded in the TypeScript harness. Model values are those returned by the FastAPI prediction endpoint at the final simulation tick. Δ stands for absolute difference (reference minus model) in percentage points (pp) or hours (h).

All predictive-accuracy deltas remain well inside the five-percentage-point and five-hour corridors, confirming that the machine-learning model embedded in predict_life() reproduces the deterministic reference to within the accepted tolerance. Classification labels match exactly, indicating that boundary logic in the React dashboard translates numeric SoH values into status badges with no off-by-one errors. System-performance checks also pass comfortably, with 95-th-percentile API latency at 112 ms and chart refresh lag at 12 ms, well under their respective limits.

These tables provide objective evidence that the prototype fulfils its monitoring and prediction requirements under every operating scenario, maintains user-visible responsiveness and scales its validation in a fully automated, repeatable fashion.

### 5.6 Summary

Chapter 5 has shown that the prototype delivers reliable, real-time insight into battery health across a spectrum of operating conditions. Five scripted profiles such as Healthy New, High-Temperature Stress, Aged High-Cycle, Low State-of-Charge Snapshot and Near End-of-Life, were exercised in a software-only environment. Graphical outputs traced health and lifespan trends for each profile, the Fleet Analytics dashboard converted those trends into fleet-level indicators and the Live Monitor translated raw telemetry into clear time-series curves, revealing the immediate impact of temperature spikes, cumulative cycling and abrupt charge depletion.

A structured validation framework then applied objective thresholds to every scenario. State-of-health predictions deviated by no more than five percentage points from deterministic reference values and remaining-life estimates stayed within five hours of ground truth. Dashboard status labels corresponded exactly with the rule set established in Chapter 3 and system performance remained user-responsive, with 95-percentile API latency at 112 milliseconds and chart refresh lag at 12 milliseconds.

These findings confirm that the monitoring and prediction system meets the Final Year Project objectives for continuous observation, accurate lifetime forecasting and timely notification. The entire stack from data simulation in BatteryManagement.jsx, through FastAPI processing, to React-based visualisation operates coherently and can serve as a solid base for future work such as physical sensor integration, multi-cell expansion or deployment in real operational settings.

# CHAPTER 6: CONCLUSION AND FUTURE WORKS

This chapter concludes the study by reflecting on the extent to which the project's aims have been achieved and outlining opportunities for further development. The chapter first summarises the key findings drawn from system implementation, simulation results and validation tests, emphasising how the prototype meets its objectives for continuous monitoring, accurate lifespan prediction and timely alert generation. It then acknowledges the limitations inherent in a software-only simulation and identifies specific areas such as physical sensor integration, multi-cell scalability and advanced machine-learning refinement where future work can extend both the robustness and the applicability of the battery-health monitoring system.

## 6.1 Objective Achievement

Table 6.1 links each project objective to concrete evidence gathered during development and validation. The first row shows that a lifetime-prediction module was successfully designed: the FastAPI endpoint `predict_life` processed live voltage, temperature and cycle data and, during validation, maintained a maximum error of five hours or less across all scenarios. The second row confirms the creation of a comprehensive monitoring system; the React dashboard streamed telemetry every second, displayed state-of-health and remaining-life values in real time and met the latency target of under two hundred milliseconds, as reported in Chapter 5. The third row demonstrates that the system was analysed through a structured technique involving five scripted scenarios and real-time data. Trend plots, composite overlays and scenario-based validation in Chapter 5 verified both predictive accuracy and status classification for each operating condition. Together, the evidence in Table 6.1 shows that all stated objectives have been fully met.

*Table 6.1: Project objectives and corresponding evidence of achievement*

| Project Objective | Evidence of Achievement |
|---|---|
| To design a system that enables prediction toward battery lifetime | Implemented a FastAPI prediction engine (predict_life endpoint) that converts real-time voltage, temperature and cycle data into remaining-life estimates. Validated accuracy in Section 5.5, where all scenarios met the ≤ 5-hour error threshold. |
| To develop a monitoring system for inspecting battery health and performance | Built a full React dashboard with Fleet Analytics, individual Live Monitor views and continuous sparkline updates. The system streams telemetry every second, displays SoH, RUL and status badges, and maintains 95-percentile API latency below 200 ms (Table 5.6). |
| To analyse the system using an Arrhenius-based technique, incorporating predefined simulation profiles—Healthy New, High-Temperature Stress, Aged High-Cycle, Low State-of-Charge Snapshot, and Near End-of-Life—along with real-time data. | Executed five scripted scenarios (Healthy New, High-Temp Stress, Aged High-Cycle, Low SoC Snapshot, Near End-of-Life) using BatteryManagement.jsx. Captured trend plots, composite overlays and summary tables in Chapter 5, then applied a validation harness that confirmed predictive accuracy and classification correctness for each scenario. |

### 6.2 Limitation and Future Works

The prototype relies on synthetic telemetry generated in software, so it does not yet capture the noise, drift or unexpected failure signatures seen in the field. A logical extension is to stream real voltage, current and temperature data from high-capacity lithium-ion modules using industrial-grade shunt sensors, thermistors and CAN-bus battery-management-system outputs connected through ESP32 microcontrollers. Continuous field measurements would allow the Arrhenius coefficients and fade-rate constants in the prediction pipeline to be recalibrated, while also helping the model learn chemistry-specific degradation patterns for lithium-ion, lead-acid, sodium-ion and other storage technologies.

The FastAPI back-end currently runs on a single workstation and tracks five batteries in memory, which limits scalability. Migrating the service to a containerised environment such as Docker or Kubernetes and storing time-series data in a cloud database like InfluxDB or TimescaleDB would enable horizontal scaling, high availability and geo-replicated dashboards. A cloud deployment would also simplify the addition of role-based access control, HTTPS termination and OAuth authentication, making the system suitable for multiple stakeholders who need secure, real-time access to battery health metrics.

The prediction engine operates with a rules-based algorithm tuned to a small parameter set. With a larger corpus of field telemetry, supervised learning techniques such as gradient-boosted trees for tabular features or recurrent neural networks for sequential data could capture non-linear interactions among temperature, depth of discharge, rest periods and C-rate. Introducing scheduled retraining, drift detection and model-version governance would let the system adapt as chemistries evolve or as operating policies change. Including additional contextual features such as ambient

weather, inverter duty cycles and charge-controller settings would further improve the horizon accuracy required for modern solar-storage deployments.

### 6.3 Summary

This chapter has reviewed how the monitoring prototype satisfied its objectives, identified the most significant boundaries of the current build and outlined three strategic directions for growth. Real sensor integration will replace simulated telemetry and expose the prediction pipeline to true operating noise and failure signatures. A cloud-based architecture will give the system the scale, resilience and security expected in industrial energy deployments. Data-driven modelling will raise predictive accuracy by learning complex relationships from large, chemistry-diverse telemetry sets. Taken together, these extensions provide a clear roadmap for transforming a research prototype into a robust platform capable of serving large fleets of lithium-ion and other battery types in real-world settings.

## REFERENCES


Chen, D., Meng, J., Huang, H., Wu, J., Liu, P., Lu, J., & Liu, T. (2022). An Empirical-Data Hybrid Driven Approach for Remaining Useful Life prediction of lithium-ion batteries considering capacity diving. *Energy*, *245*, 123222. https://doi.org/10.1016/j.energy.2022.123222

GeeksforGeeks. (2024, December 31). *Iterative Waterfall Model Software engineering*. GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/

Guo, W., Sun, Z., Vilsen, S. B., Meng, J., & Stroe, D. I. (2022). Review of "grey box" lifetime modeling for lithium-ion battery: Combining physics and data-driven methods. *Journal of Energy Storage*, *56*, 105992. https://doi.org/10.1016/j.est.2022.105992

Hong, J., Li, K., Liang, F., Yang, H., Zhang, C., Yang, Q., & Wang, J. (2023). A novel state of health prediction method for battery system in real-world vehicles based on gated recurrent unit neural networks. *Energy*, *289*, 129918. https://doi.org/10.1016/j.energy.2023.129918

Kirana, R. C., Purwanto, N. A., Azis, N. A., Joelianto, E., Santosa, S. P., Budiman, B. A., Nguyen, L. H., & Turnip, A. (2023). Failure assessment in lithium-ion battery packs in electric vehicles using the failure modes and effects analysis (FMEA) approach. *Mechatronics Electrical Power and Vehicular Technology*, *14*(1), 94–104. https://doi.org/10.14203/j.mev.2023.v14.94-104

Kucinskis, G., Bozorgchenani, M., Feinauer, M., Kasper, M., Wohlfahrt-Mehrens, M., & Waldmann, T. (2022). Arrhenius plots for Li-ion battery ageing as a function of temperature, C-rate, and ageing state – An experimental study. *Journal of Power Sources*, *549*, 232129. https://doi.org/10.1016/j.jpowsour.2022.232129

Li, D., Yang, D., Li, L., Wang, L., & Wang, K. (2022). Electrochemical impedance spectroscopy based on the state of health estimation for Lithium-Ion batteries. *Energies*, *15*(18), 6665. https://doi.org/10.3390/en15186665

Li, T., Zhou, Z., Thelen, A., Howey, D. A., & Hu, C. (2024). Predicting battery lifetime under varying usage conditions from early aging data. *Cell Reports Physical Science*, *5*(4), 101891. https://doi.org/10.1016/j.xcrp.2024.101891

Liu, Y., Liu, C., Liu, Y., Sun, F., Qiao, J., & Xu, T. (2023). Review on degradation mechanism and health state estimation methods of lithium-ion batteries. *Journal of Traffic and Transportation Engineering (English Edition)*, *10*(4), 578–610. https://doi.org/10.1016/j.jtte.2023.06.001

Markets, R. A. (2025, February 4). Battery Market Outlook 2025-2030: Insights on electric vehicles, energy storage and consumer electronics growth. *GlobeNewswire News Room* https://www.globenewswire.com/news-release/2025/02/04/3020360/0/en/Battery-Market-Outlook-2025-2030-Insights-on-Electric-Vehicles-Energy-Storage-and-Consumer-Electronics-Growth.html

Pradhan, S. K., & Chakraborty, B. (2022). Battery management strategies: An essential review for battery state of health monitoring techniques. *Journal of Energy Storage*, *51*, 104427. https://doi.org/10.1016/j.est.2022.104427

Qu, X., Shi, D., Zhao, J., Tran, M., Wang, Z., Fowler, M., Lian, Y., & Burke, A. F. (2024). Insights and reviews on battery lifetime prediction from research to practice. *Journal of Energy Chemistry*, *94*, 716–739. https://doi.org/10.1016/j.jechem.2024.03.013

Sultan, Y. A., Eladl, A. A., Hassan, M. A., & Gamel, S. A. (2025). Enhancing electric vehicle battery lifespan: integrating active balancing and machine learning for precise RUL estimation. *Scientific Reports*, *15*(1). https://doi.org/10.1038/s41598-024-82778-w

Wang, X., Wei, X., & Dai, H. (2019). Estimation of state of health of lithium-ion batteries based on charge transfer resistance considering different temperature and state of charge. *Journal of Energy Storage*, *21*, 618–631. https://doi.org/10.1016/j.est.2018.11.020

*What is a Data Flow Diagram*. (n.d.). Lucidchart. https://www.lucidchart.com/pages/data-flow-diagram

Zhang, X., Hou, J., Wang, Z., & Jiang, Y. (2022). Study of SOC Estimation by the Ampere-Hour Integral Method with Capacity Correction Based on LSTM. *Batteries*, *8*(10), 170. https://doi.org/10.3390/batteries8100170

Zhao, J., Feng, X., Tran, M., Fowler, M., Ouyang, M., & Burke, A. F. (2024). Battery safety: Fault diagnosis from laboratory to real world. *Journal of Power Sources*, *598*, 234111. https://doi.org/10.1016/j.jpowsour.2024.234111

Zhou, X., Pan, Z., Han, X., Lu, L., & Ouyang, M. (2018). An easy-to-implement multi-point impedance technique for monitoring aging of lithium ion batteries. *Journal of Power Sources*, *417*, 188–192. https://doi.org/10.1016/j.jpowsour.2018.11.087