FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Design and Implementation of a Real-Time Digital Twin Platform for Monitoring and Managing EV Charging Infrastructure

Tiago da Silva Azevedo

Mestrado em Engenharia de Software

Supervisor: Prof. Francisco Maia

Supervisor at INESC TEC: Eng. Daniel Barros

July 22, 2026

Design and Implementation of a Real-Time Digital Twin Platform for Monitoring and Managing EV Charging Infrastructure

Tiago da Silva Azevedo

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

President: Prof. Ana Paiva
Examiner: Prof. Fábio Coelho
Member: Prof. Francisco Maia

July 22, 2026

Resumo

A transição global para a mobilidade elétrica exige infraestruturas de carregamento mais inteligentes e escaláveis. Os sistemas de gestão de estações de carregamento (*Charging Station Management Systems*, CSMS) atuais asseguram os mecanismos de controlo operacional, como autenticação, faturação e monitorização básica, mas carecem, em geral, da visibilidade em tempo real e da capacidade preditiva necessárias para uma manutenção proativa, funcionando como sombras digitais passivas em vez de gémeos digitais (*Digital Twin*, DT) ativos. Esta dissertação concebe, implementa e avalia uma plataforma de DT em tempo real para a infraestrutura de carregamento de veículos elétricos. A plataforma adota uma arquitetura desacoplada e baseada em event sourcing que ingere e normaliza o tráfego OCPP 1.6 e 2.0.1 num único esquema canónico, reduzindo-o a um modelo de estado das entidades em tempo real e disponibilizando esse estado aos operadores dentro de um segundo. Uma estratégia de integração não disruptiva posiciona o CSMS existente como camada de transporte e o DT como camada de inteligência, permitindo que este coexista com a infraestrutura em produção sem assumir a responsabilidade pela conformidade do protocolo, pela autenticação ou pela gestão das ligações. A plataforma é validada com dados sintéticos e com dados reais de produção: nove meses de tráfego OCPP 2.0.1 genuíno: 1431 sessões de carregamento de 22 carregadores de um CSMS em produção foram ingeridos de ponta a ponta sem perda de mensagens, tendo o sistema cumprido o objetivo de sincronização inferior a um segundo a uma escala de cinquenta carregadores. Uma prova de conceito de *machine-learning* estabelece ainda a telemetria da plataforma como uma base viável para a deteção de anomalias, delimitando de uma forma honesta a diferença entre dados sintéticos e reais. O resultado é uma base validada para um DT proativo e bidirecional, ficando as camadas de atuação em ciclo fechado e de inteligência especificadas na arquitetura e identificadas como trabalho futuro.

Palavras-chave: Infraestrutura de Carregamento de Veículos Elétricos • Sistema de Gestão de Estações de Carregamento • Gémeo Digital • OCPP • Deteção de Anomalias

Abstract

The global shift towards electric mobility demands smarter, more scalable charging infrastructure. Existing Charging Station Management System (CSMS) platforms provide operational control mechanisms, such as authentication, billing, and basic monitoring, but generally lack the real-time visibility and predictive capability required for proactive maintenance, functioning as passive digital shadows rather than active digital twins. This dissertation designs, implements, and evaluates a real-time Digital Twin (DT) platform for electric-vehicle charging infrastructure. The platform adopts a decoupled, event-sourced architecture that ingests and normalises both OCPP 1.6 and 2.0.1 traffic into a single canonical schema, reduces it to a live entity-state model, and exposes that state to operators within a sub-second latency threshold. A non-disruptive integration strategy positions the existing CSMS as a transport layer and the twin as an intelligence layer, allowing the twin to coexist with production infrastructure without assuming responsibility for protocol compliance, authentication, or connection handling. The platform is validated against both synthetic load and real production data: nine months of genuine OCPP 2.0.1 traffic, 1,431 charging sessions across 22 chargers from a production CSMS, were ingested end-to-end without message loss, and the system met its sub-second synchronisation target at a fifty-charger scale. A machine-learning proof-of-concept further establishes the platform's telemetry as a viable foundation for learned anomaly detection, while honestly delineating the gap between synthetic and real data. The result is a validated foundation for a proactive, bidirectional digital twin, with the closed-loop actuation and intelligence layers specified in the architecture and identified as future work.

Keywords: EV Charging Infrastructure • Charging Station Management System • Digital Twin • OCPP • Anomaly Detection

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation contributes to these goals by strengthening the digital infrastructure that supports the transition to electric mobility. By providing a real-time digital twin that gives operators continuous visibility into charging assets and a foundation for predictive maintenance and smart-charging intelligence, this work supports more efficient, resilient, and renewable-aligned charging networks. The addressed SDGs are the following:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 11 Make cities and human settlements inclusive, safe, resilient and sustainable

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.2, 7.3	The architecture positions the twin as an intelligence layer enabling PV-aware smart charging, improving how charging demand is matched to renewable generation.	Share of charging energy aligned with on-site renewable generation; reduction in peak grid draw per site.
9	9.1, 9.4	A real-time, event-sourced digital twin makes EV charging infrastructure observable, recoverable, and analysable, improving reliability and enabling proactive maintenance.	Sub-second state-synchronisation latency; zero-loss ingestion of real OCPP traffic; reduction in charger downtime via earlier fault detection.
11	11.2	More reliable, well-managed public charging lowers a barrier to EV adoption and supports clean urban mobility.	Charger availability/uptime; sessions served without interruption.

Acknowledgements

First, I want to thank my supervisor, Prof. Francisco Maia, for his guidance and support from the very beginning of this dissertation. His constant availability, feedback, and encouragement contributed greatly to the completion of this work.

To Daniel Barros, my INESC TEC supervisor, thank you for the immense help and mentorship since the day I joined the team. His practical insights and technical expertise have been invaluable for my professional and academic growth. My sincere thanks go to Eng. José Silva, my initial supervisor, and the rest of the team.

I also want to thank my great colleagues at FEUP for the shared journey alongside this master's program.

Finally, I owe a huge debt of gratitude to my family, friends, and Ana Lúcia for their love, patience, and encouragement and for supporting me through every phase of my life.

Last of all, this work is for my grandmother. She did not see it completed, but I know she has been watching over me from somewhere else, and I carried her with me through every step of it.

Tiago

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo com as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.

Tiago da Silva Azevedo

Contents

List of Figures

List of Tables

List of Acronyms

AC Alternating Current
ADR Architectural Decision Record
AI Artificial Intelligence
API Application Programming Interface
BMS Battery Management System
CI Continuous Integration
CRUD Create, Read, Update, and Delete
CS Charging Station
CSMS Charging Station Management System
DC Direct Current
DT Digital Twin
EV Electric Vehicle
EVCI Electric Vehicle Charging Infrastructure
EVSE Electric Vehicle Supply Equipment
HTTP Hypertext Transfer Protocol
IoT Internet of Things
JSON JavaScript Object Notation
JWT JSON Web Token
LP Linear Programming
ML Machine Learning
MQTT Message Queuing Telemetry Transport
MVP Minimum Viable Product
OCPP Open Charge Point Protocol
PoC Proof of Concept
PV Photovoltaic
REST Representational State Transfer
SDG Sustainable Development Goal
SoC State of Charge
TLS Transport Layer Security
TTL Time-To-Live
UI User Interface

Chapter 1

Introduction

Current Electric Vehicle Charging Infrastructure (EVCI) management systems remain largely reactive. This dissertation addresses that gap by designing, implementing and validating a real-time Digital Twin (DT) for EVCI. This chapter frames the work: it establishes the context (Section 1.1) and motivation (Section 1.2), states the objectives and research questions (Section 1.3), summarises the contributions (Section 1.4), and outlines the structure of the remainder of the document (Section 1.5).

1.1 Context

The worldwide transition towards sustainable transportation has accelerated the adoption of Electric Vehicles (EVs), placing growing pressure on power grids and infrastructure management systems. In the context of EV Charging, the Charging Station Management System (CSMS) is the core operational layer: a cloud-based platform responsible for user authentication, transaction authorisation, billing, and monitoring of Electric Vehicle Supply Equipment (EVSE).

This growth presents what Tafazzoli et al. [53] describe as a dual challenge: scaling the EVCI while managing the complexities of grid stability. Scalability is constrained by the high costs of deploying widespread charging networks, yet it is essential for equitable access to EV charging points; simultaneously, integrating renewable energy sources and addressing peak demand resulting from concurrent charging activities are critical grid management concerns.

Despite this need for growth, existing charging management systems remain outdated. Yu et al. [64] argue that the state of these systems results in low utilisation rates, poor fault identifiability, and high maintenance fees. DTs have emerged as a solution, yet the review by Morgan and Ali [39] finds that most implementations focus on cybersecurity or grid integration rather than the physical charger itself.

Where monitoring of the charger does exist, it remains passive. Recent advancements in runtime monitoring, such as the explainable Artificial Intelligence (AI) framework of Rahmati [50], can detect faults in real-time but stop at observation: it functions as a Digital Shadow (Section 2.3), lacking the bidirectional Open Charge Point Protocol (OCPP) integration needed for autonomous

fault mitigation. The industry has the intelligence to diagnose faults but lacks a standardised architecture for automated responses. This is the gap this dissertation addresses: the absence of a predictive, real-time DT centred on the physical charger.

1.2 Motivation

The reliability of the EVCI is now critical to the wider transition to electric mobility, yet its operational management remains fundamentally reactive. When a charger fails, the operator typically knows it only after the fact, from raw logs or a user complaint, by which point the consequences have already landed: stranded drivers, lost revenue, and eroded confidence in the network. The CSMS, the operational layer introduced in Section 1.1, is built for billing and authentication rather than diagnosis; it offers a delayed, reactive view of the physical world; it can execute basic commands but lacks the real-time, predictive insight to act on a fault before it escalates.

This motivates the shift from reactive monitoring to proactive management. By extending the DT paradigm with high-frequency telemetry, an event-driven architecture, and AI, it becomes possible to maintain a bidirectional replica that does not just observe each charger but acts on it, forecasting constraints, and ultimately intervening before a fault becomes an outage. Building the foundations of such a twin, without disrupting the CSMS operations it must coexist with, is the problem this dissertation takes on.

1.3 Objectives and Research Questions

The primary objective of this dissertation is to design, implement, and evaluate a real-time DT platform for EVCI. The platform must synchronise with physical chargers via standard protocols, integrate non-disruptively with existing CSMS architectures, and establish the foundation for AI-driven predictive maintenance. The work is scoped as a Minimum Viable Product (MVP) that realises the platform's ingestion, state-synchronisation, and delivery layers in full, together with a Proof of Concept (PoC) of its intelligence layer, and validates them against both synthetic load and real production data. While the architecture is designed to accommodate multiple management protocols behind a common ingestion layer, this realisation targets OCPP alone, the protocol in use in the infrastructure available for this research (Section 2.2).

To guide the research, three Research Questions were defined:

- **RQ1**: How can a DT achieve real-time, bidirectional synchronisation with physical EVSE without disrupting the critical operations (authentication, connection handling) of an existing CSMS?
- **RQ2**: What architectural patterns and polyglot persistence strategies are required to ingest, normalise, and reduce high-frequency telemetry into a real-time, sub-second state representation while preserving a durable, auditable event history?

- **RQ3**: How can Machine Learning (ML) models be integrated into the DT to exploit historical telemetry for predictive anomaly detection, and what does the platform's telemetry surface enable in practice?

1.4 Contributions

By answering these research questions, this dissertation delivers both an architectural design and a working engineering implementation. Five contributions follow, each annotated with the research question it primarily addresses:

1. **A scalable DT architecture (RQ2):** a decoupled, six-layer, event-driven architecture that separates protocol ingestion, state management, intelligence, and actuation, and is recoverable by construction through event sourcing.
2. **A non-disruptive integration strategy with legacy infrastructure (RQ1):** a methodology that positions the existing CSMS as a transport layer while the DT acts as the intelligence layer, normalising OCPP 1.6 and 2.0.1 traffic into a unified canonical schema without compromising the CSMS's authentication and connection handling.
3. **A working MVP implementation (RQ2):** six independently deployable back-end services, backed by a polyglot persistence layer, that ingest high-frequency telemetry, maintain real-time entity state through event sourcing across Redis and PostgreSQL, persist time-series telemetry to TimescaleDB, and drive a live, sub-second twin User Interface (UI).
4. **Validation against real-world production data (RQ1):** the ingestion of nine months of real OCPP 2.0.1 traffic, comprising 1,431 charging sessions across 22 chargers from a production CSMS, through the complete pipeline, demonstrating that the platform and its integration strategy operate on authentic field data rather than only on simulation.
5. **A clearly scoped PoC for the intelligence layer (RQ3):** the development and offline evaluation of an anomaly-detection model over session telemetry, establishing the platform as a viable data source for a learned intelligence layer while honestly delineating what such a model can and cannot yet claim.

1.5 Dissertation Structure

The remainder of this dissertation is organised as follows. Chapter 2 establishes the technical foundations: the EV charging ecosystem, the relevant communication protocols, the DT paradigm, and the ML techniques relevant to anomaly detection. Chapter 3 critically reviews the existing literature, analysing current EVCI management strategies and the application of DTs in this domain, and identifying the technological gaps that this work addresses. Chapter 4 presents the proposed architecture, justifying the technology stack and integration strategies. Chapter 5 details the engineering implementation of the MVP. Chapter 6 presents the evaluation: the system's performance

against its latency threshold, its validation against real-world field data, and the assessment of the ML PoC. Finally, Chapter 7 concludes the dissertation, summarising the contributions and outlining future work.

Chapter 2

Background

This chapter establishes the technical foundations on which the dissertation builds. It first describes the EV charging ecosystem and its principal components (Section 2.1), then details the communication protocols that connect them, from low-level physical signalling to application-level management (Section 2.2). Section 2.3 defines the DT paradigm, distinguishing passive monitoring models from the bidirectional control required for proactive operation, and Section 2.4 reviews the ML techniques relevant to detecting anomalies in charging telemetry.

2.1 The EV Charging Ecosystem

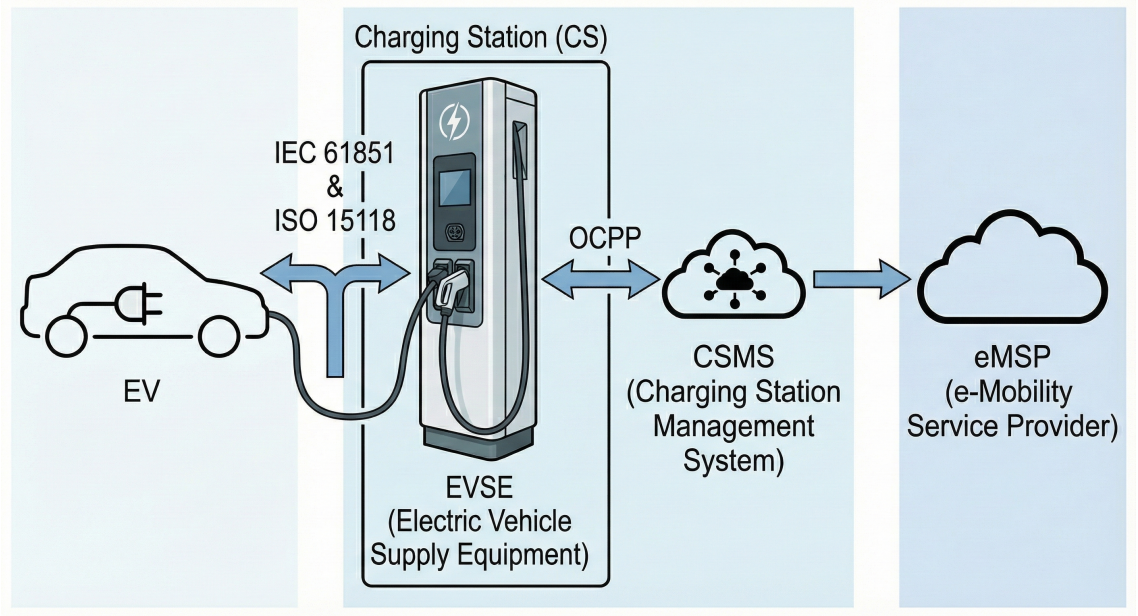

Figure 2.1: The EVCI ecosystem and its principal actors.

Figure 2.1 illustrates the EV charging ecosystem, a hierarchy of physical and logical components. The collective hardware and software used to supply energy to vehicles is the EVCI. At the

physical edge lies the Charging Station (CS), which manages one or more charging points and houses the EVSE and the connectors that link to the EV [16]. The core subsystem within the CS is the EVSE itself: it provides the interface through which an EV connects and charges, and it collects data on charging and connectivity status [16]. Overseeing a set of CSs is the CSMS, a centralised software platform for the monitoring, control, and management of multiple CSs, through which operators handle user access, maintenance, and data analytics [47]. This logical layer in turn supports the E-Mobility Service Provider, which holds the contractual relationship with the EV driver and issues authentication tokens.

Beyond this hierarchy of actors, energy is delivered at three power levels. Waseem et al. [58] categorise charging by power delivery: Levels 1 and 2 use Alternating Current (AC) for slower domestic or public charging, while Level 3, Direct Current (DC) Fast Charging, bypasses the vehicle's onboard charger to deliver high-voltage DC for rapid charging.

2.2 Communication Protocols

The control logic of an EV charger operates on two distinct layers: low-level hardware signalling between the vehicle and the charger, and high-level application management between the charger and the backend.

As explained by Hecht, Figgener, and Sauer [20], IEC 61851-1 governs the physical connection between the EV and the CS. It permits only very simple communication, encoded through resistance and pulse-width values: the resistance indicates the cable's current-carrying capacity, while the pulse-width signal communicates that of the CS. Superimposed on IEC 61851-1, ISO 15118 supports the richer interactions required for services such as bidirectional charging and scheduling [20].

Layered above the physical signalling is the Open Charge Point Protocol (OCPP), developed by the Open Charge Alliance. As described by Hecht, Figgener, and Sauer [20], OCPP carries charger-to-backend communication as JavaScript Object Notation (JSON) messages over a persistent WebSocket connection, supporting functions such as authorisation and meter value reporting. The protocol has evolved across two widely deployed versions that differ significantly in structure: OCPP 1.6, the most prevalent in installed infrastructure, uses a flat key-value message format [44], whereas OCPP 2.0.1 introduces a more complex, object-oriented structure [42] with finer-grained device management, explicit security profiles, and support for ISO 15118 smart-charging features. Both operate as a request-response protocol between each CS and the CSMS.

An OCPP exchange unfolds a sequence of messages that together frame a charging session (a transaction, in OCPP terms). The charger first registers with the backend (`BootNotification`) and reports the state of each connector (`StatusNotification`); a session then opens when a vehicle is authorised and plugged in (`StartTransaction`, unified in 2.0.1 as a `TransactionEvent`), streams periodic readings as it charges (`MeterValues`), and closes when charging ends (`StopTransaction`). This session is the period from plug-in to unplug and is the fundamental

unit the platform models. Since a single deployment may contain chargers speaking either dialect, the resulting version heterogeneity is addressed directly by the normalisation layer of this dissertation (Section 4.5).

OCPP is not the only protocol operating at this level. The IEC 63110 series was developed in parallel by IEC technical committee 69 as a formal standard for the management of EV charging and discharging infrastructures [25]. That same committee has meanwhile published OCPP 2.0.1 as international standard IEC 63584 [26], so that the more recent of the two protocol versions handled in this work is itself the international standard for the EVSE–CSMS interface. Management protocols in this space differ in message syntax and transport, but expose a largely common set of domain concepts: charging sessions, connector state, energy measurements and remote control commands. The architecture proposed in this dissertation therefore confines protocol knowledge to its ingestion and normalisation layer (Section 4.5), so that supporting an additional protocol amounts to adding an adapter rather than altering the state, persistence or intelligence layers above it. Both the dissertation and the MVP focus exclusively on OCPP, as it is the protocol spoken by the charging infrastructure available at INESC TEC, which enabled validation against real operational data.

2.3 The Digital Twin Paradigm

The concept of the DT has evolved from static modelling to dynamic, bidirectional interaction. Kritzinger et al. [31] formalised this evolution by distinguishing between three levels of integration, set apart by the automation and direction of the data flow between the physical and digital objects, as illustrated in Figure 2.2.

- **Digital Model:** A digital representation of an existing or planned physical object with manual data flow: a change in the physical object has no automatic effect on the digital model, and vice-versa.
- **Digital Shadow:** An automated one-way data flow from the physical object to the digital object: a change in the physical object updates the digital object, but not vice-versa.
- **Digital Twin:** An automated bidirectional data flow between the physical and digital objects: the digital entity can control or influence the state of the physical entity.

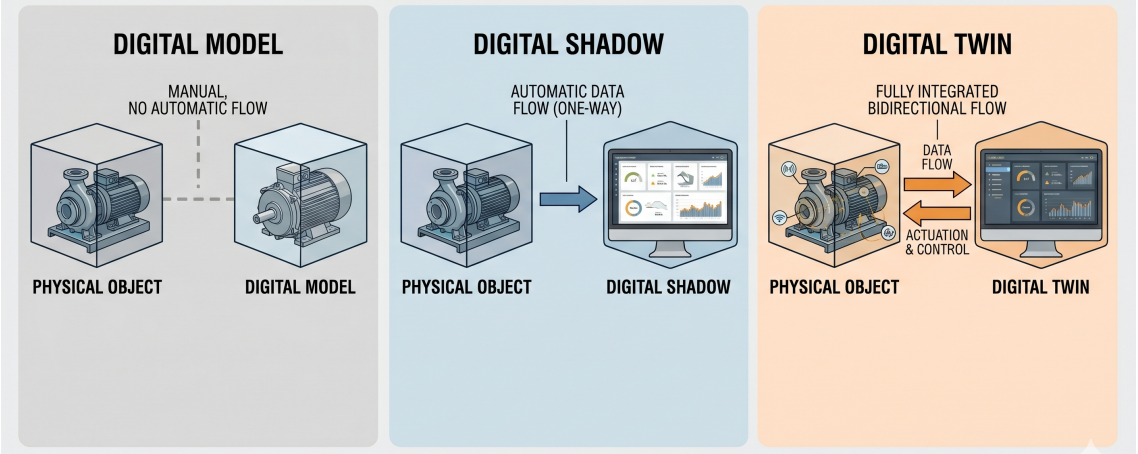

Figure 2.2: The Kritzinger et al. [31] taxonomy.

This distinction is central to the dissertation: while many current EV management systems function as Digital Shadows, merely visualising physical-layer state, a true DT must be capable of feedback control to optimise operations in real time [31].

In the specific context of EVCI, Yu et al. [63] argue that a mature charging DT should be *cognisant* of faults, *adaptive* to grid constraints, *taskable* enough to execute management goals, and *ethical* in its adherence to regulation, criteria that frame the predictive, controllable, and compliant platform this dissertation sets out to build.

2.4 Machine Learning for Time-Series Data

The telemetry produced by charging infrastructure is inherently a time series: each charger reports measurands such as active power, energy, voltage, current, frequency, and battery State of Charge (SoC) at regular intervals throughout a session. Detecting faults in this data is the problem of identifying anomalies: sequences whose behaviour departs from the expected pattern, whether through a sudden power drop, a frozen meter, or a gradual divergence from a physical model. Two broad families of ML techniques address this problem, distinguished by whether labelled examples of faults are available.

Unsupervised models learn the structure of normal operation and flag deviations from it without requiring labelled faults. The Isolation Forest is a widely used example: it isolates anomalies by recursively partitioning the feature space at random, exploiting the observation that outliers require fewer partitions to separate than normal points. Such methods are attractive because labelled fault data is scarce in practice, but they are limited by their lack of knowledge of which deviations are operationally significant. Supervised methods, by contrast, learn a decision boundary from labelled examples of both normal and faulty behaviour. Gradient-boosted decision-tree ensembles such as XGBoost are a strong and widely adopted choice for tabular data of this kind: they combine many weak tree learners into an accurate classifier and handle heterogeneous, non-linear features well. Their accuracy, however, depends entirely on the availability and quality of labels.

A practical challenge specific to session-based telemetry is that a charging session is a variable-length sequence, whereas most classifiers expect a fixed-size input. The common solution is feature engineering: reducing each session's time series to a fixed vector of summary statistics (means, standard deviations, ranges, monotonicity ratios, and comparisons between the first and second halves of the session) that capture its statistical and temporal shape. This transforms anomaly detection over raw time series into a standard tabular classification problem at the granularity of a session. An alternative is to apply deep sequence models (recurrent networks or autoencoders) directly to the raw series, avoiding hand-engineered features; these are powerful but data-hungry and less interpretable, which makes the feature-based tabular approach the more practical choice given the label scarcity discussed next.

The decisive practical obstacle to applying these techniques in production is the availability of labelled fault data. Real charging infrastructure rarely records ground-truth fault annotations, which both motivates unsupervised approaches and explains why much of the literature, as Chapter 3 discusses, resorts to synthetic or simulated data for supervising training. This tension between the power of supervised methods and the scarcity of real labels directly shapes the PoC evaluated later in this dissertation (Sections 5.7, 6.4).

2.5 Summary

This chapter has established the technical foundations on which the dissertation builds. It introduced the EV charging ecosystem and its principal components, the CS, the EVSE, the CSMS, and the E-Mobility Service Provider, and the layered communication standards that connect them, from low-level IEC 61851-1 signalling to the application-level OCPP that carries charging data to the backend. It then defined the DT paradigm through Kritzinger et al. [31]'s distinction between the Digital Model, the Digital Shadow, and the fully bidirectional DT, establishing the conceptual gap between passive monitoring and active control that this work seeks to close. Finally, it outlined the ML techniques relevant to detecting anomalies in charging telemetry, together with the practical obstacle of label scarcity. With these foundations in place, the following chapter reviews the state of the art, examining how existing systems apply these concepts, and identifying the specific gaps this dissertation addresses.

Chapter 3

State of the Art

This chapter presents a critical review of the state of the art in EVCI management and DTs, situating the dissertation within the existing body of work and establishing the gaps it sets out to close. The review is organised thematically. Section 3.1 frames the operational problem: how charging infrastructure is managed today, and why that paradigm falls short. The next three sections examine the technical state of the art: Section 3.2 assesses the protocols available for reliable, bidirectional communication; Section 3.3 distinguishes passive monitoring models (Digital Shadows) from active twins capable of feedback control; and Section 3.4 reviews where machine intelligence is currently directed, namely hardware monitoring, grid optimisation, and cybersecurity. Section 3.5 then synthesises these threads into the two gaps this dissertation addresses: the absence of real-time, standards-based integration between predictive techniques and the live infrastructure, and the lack of validation against authentic production data.

3.1 Current EVCI Management Landscape

The management of EVCI is undergoing a major transformation, from a phase of rapid, uncoordinated deployment towards one that demands reliability and intelligent operation. The scale of this shift is significant: the number of publicly accessible chargers reached roughly 2.7 million units in 2022, a 50% increase over 2020 [63]. This rapid proliferation enlarges the operational and maintenance burden placed on operators, yet the prevailing management focus has remained on expansion rather than upkeep.

This expansion-centric mindset is evident across the literature. Karkaria et al. [29], for instance, apply ML models to decide where new stations should be sited, using location and population data to minimize the risk of a poorly located installation. Valuable as this is for placement, that risk is defined in purely financial terms (the cost of a badly sited asset) and the approach offers no means of monitoring the charger once it is deployed.

Existing CSMSs reflect the same priority. The framework of Hareshma and Sivraj [19] is a web-based application built around user-facing functions such as station filtering and slot booking; it operates primarily as an economic and logistical tool rather than an infrastructure management

system, and the authors themselves position standardised protocol support, including OCPP, only as a future enhancement. Li et al. [32] corroborate this orientation, describing the goals of real-time EV monitoring chiefly as steering drivers towards available chargers, guiding capacity expansion, and enabling load management. They further note that the monitoring layer is itself fragile: technical faults, sensor malfunctions, communication failures, and vandalism can deteriorate the effectiveness and reliability of the monitoring system precisely when it is most needed.

A further obstacle to scalability is the absence of standardised communication. Tafazzoli et al. [53] identify the absence of standardized data formats, charging protocols, and regulatory policies as a major barrier to scaling EVCI. In practice, many implementations rely on custom Application Programming Interfaces (APIs) rather than open protocols, limiting their ability to integrate and scale within broader systems.

3.2 Standards and Connectivity

Despite the management barriers above, the standards required for reliable connectivity are well established. Hecht, Figgener, and Sauer [20] observe that widely deployed OCPP 1.6 covers the basic tasks of operating a CS, such as authentication and the communication of meter readings, while more advanced capabilities such as smart changing or *Plug and Charge* arrive only from OCPP 2.0 onward. Recent work by Raghav et al. [49] demonstrates that real-time, TCP/IP-based communication between the EVSE and the CSMS is achievable with OCPP 2.0.1: using low-cost hardware such as NodeMCU gateways, a complete pipeline can be built with IEC 61851-1 handling the handshake between the EVSE and the vehicle's Battery Management System (BMS) and OCPP carrying the standardised station-backend communication.

At the application layer, the industry is transitioning to OCPP 2.0.1 to enable more advanced control. Hsaini, Ghogho, and Charaf [24] show that the protocol's remote-execution operations can be used to automatically enforce optimised charging schedules, and Papapostolou and Kapsalis [47] confirm that OCPP 2.0.1 is essential to issuing commands that drive charging optimisation. Addressing a different facet, Nambiar, Jadhav, and Ekande [40] tackle the software complexity of multi-gun DC chargers with a shared-memory interface that preserves OCPP compliance while supporting simultaneous sessions, demonstrating that OCPP can also serve as the backbone for operational orchestration.

The standards are not, however, designed with DTs in mind. Morgan and Ali [39] identify a disconnect between current protocols and advanced modelling needs, arguing that standards such as ISO 15118 and OCPP 2.0.1 were formulated without DT architectures in mind, leaving substantial deficiencies in the authentication, authorization, and data integrity requirements specific to DT deployments in EV CS environments.

These limitations have led some researchers to bypass OCPP in favour of generic Internet of Things (IoT) solutions. Hao et al. [18] propose a multi-protocol cloud platform that relies on Message Queuing Telemetry Transport (MQTT) for real-time charger-to-cloud communication;

while this assures low-latency transmission for remote monitoring, it does so at the cost of contributing to EVCI fragmentation. Similarly, Nizam et al. [41] build a real-time monitoring system on ESP32 microcontrollers that combines Modbus (data acquisition), ESP-NOW (local communication), and MQTT (cloud monitoring). By substituting custom MQTT stacks for standardised OCPP, such systems forgo interoperability across hardware vendors and management systems.

This trade-off is not inevitable. Sujatha et al. [52] implement a standard-compliant OCPP client over WebSockets on the same class of low-cost ESP32 hardware, showing that real-time charging-session monitoring is attainable within the standard framework rather than requiring proprietary protocols.

A second line of work layers middleware on top of standard protocols to overcome their limitations. Devendra et al. [8] argue that standard OCPP imposes a purely vertical information flow (charger-to-cloud) that isolates the device from its surroundings, and integrate the oneM2M platform to enable horizontal communication so a charger can react to external triggers. Gupta et al. [17] extend this with Wi-SUN networking to embed chargers into smart-city infrastructure, again via oneM2M, enabling autonomous responses to emergencies such as power disconnection. Yet, these approaches remain reactive: they improve interoperability while leaving predictive maintenance capability unexplored.

Finally, reliance on these protocols introduces significant vulnerabilities. Gandhi and Morsi [13] note that OCPP, particularly the widely deployed 1.6, lacks built-in security measures, creating unauthorised access points that can compromise the stability of connected microgrids. Taken together, the literature shows that while standards for connectivity [49] and remote control [24] are mature, standards for secure and predictive operation remain absent, a gap this dissertation addresses at the architectural level.

3.3 Digital Twin Applications

Current real-time DTs in this domain focus predominantly on energy balancing and renewable integration, treating the charger as a simple switch rather than a complex asset requiring maintenance.

Energy and Fleet Optimisation Recent implementations illustrate the value of DTs for grid-level management. Testasecca et al. [54] present a 2025 DT integrating 15 EVs and a Photovoltaic (PV) system, achieving real-time synchronisation via Apache Kafka to optimise self-consumption. Yet, while it optimises energy flow, it assumes the CS functions perfectly: the DT models the schedule, not the charger itself, leaving a clear gap in predictive maintenance for the EVSE. De Bardi and Gruosso [7] exemplify the same limitation with a co-simulation framework linking a digital grid model with a Python-based CSMS, synchronised over OCPP 1.6 at 10-second intervals; their focus, however, is the electrical behaviour of the grid and the resilience of the communication layer to cyberattacks, with the EVSE hardware itself treated as a black box. In fleet management, Francisco, Monteiro, and Cardoso [10] propose a DT to optimise corporate-fleet

charging, reducing operational costs by synchronising charging with renewable energy availability; yet the digital replica is defined purely by energetic and logistical parameters rather than the physical condition of the EVSE, so the optimisation implicitly assumes 100% infrastructure availability, an assumption that breaks down once the underlying hardware degrades. In the same vein, Jasni, Rashid, and Daud [28] develop an IoT-enabled charging pillar whose monitoring is explicitly limited to energy metering rather than the health of the charging hardware.

Simulation Digital Twins DTs have also been used to validate automation software before deployment. Galkin, Yang, and Vyatkin [12] introduce a framework that automatically generates a DT from IEC 61850 configuration files, building a MATLAB/Simulink model with Python-based OCPP agents to simulate the CSMS-charging-point interaction. While this validates the protocol's remote control and management features over the autogenerated model, it explicitly sets the hardware aside: the authors describe the model as highly simplified and not directly applicable to real hardware, and frame their goal as the successful automatic construction of a DT model rather than the accuracy of the model itself. Menon and Nithin [37] likewise build a simulator for the GB/T standard that models the EV-charger communication flow, but with the aim of helping users understand that flow rather than monitor the physical charger. The pattern is consistent: current digital tools are optimised for logic verification, not predictive maintenance.

Basic Monitoring Approaching the problem from a hardware-safety angle, Sanjana et al. [51] monitor voltage, current, and temperature in real-time via IoT sensors, but the implementation is strictly reactive: the system automatically disconnects the supply whenever current or voltage exceeds predefined safety thresholds. More recently, Gajera, Mohamed, and Azim [11] propose a Dynamic Overload Surge Adjustment Protection system that dynamically adjusts safety thresholds to handle voltage fluctuations, yet it too remains reactive, disconnecting the charger when critical thresholds are crossed rather than predicting the fault. Complementing these sensor-based methods, Zhang et al. [66] apply computer vision to physical safety, detecting user negligence such as a discarded charging gun by integrating prior knowledge into a VGG16 network; despite the sophistication, the focus stays on user-centric anomalies rather than device-centric prediction. Moving beyond physical sensors, Rahmati [50] targets the reliability of the control software itself, delivering a runtime-monitoring engine that uses Bayesian reliability analysis and explainable AI to detect software faults, such as API latency spikes and memory leaks, in real-time. While an advance in predictive diagnostics, the system operates as a Digital Shadow rather than a twin: it alerts a human operator but lacks the bidirectional control loop required for autonomous mitigation, and its reliance on internal software metrics instead of standardised OCPP limits its applicability in multi-vendor environments.

3.4 AI/ML for Predictive EVCI Management

Although ML is widely proposed to address inefficiencies in the EV ecosystem, its application to the physical reliability of the charging hardware remains limited. Three patterns explain this: a bias towards grid and user behaviour, the conflation of anomaly detection with cybersecurity, and an emerging but fragmented body of component-diagnosis work that lacks real-time integration.

The Bias Towards Grid and User Behaviour In a systematic review, Thwany et al. [56] find that most current ML research targets domains such as smart charging, grid integration, dynamic pricing, and battery management and that the battery management work focuses exclusively on the EV battery, ignoring the EVSE. This confirms that the literature treats the CS merely as an interface to the grid; Gajera, Mohamed, and Azim [11], for example, apply an LSTM model to reduce electricity costs and grid stress.

Recent implementations reinforce this bias. Testasecca et al. [54] use K-Means clustering to categorise vehicles by velocity and energy consumption and meta-heuristic algorithms (GWO-WOA) to schedule charging; effective as this is for grid load management, such models do not monitor the internal physical signals, needed for predictive maintenance. The same orientation appears in Hareshma and Sivraj [19], whose "Smart Charging Infrastructure" confines its intelligence to logistical convenience (Gradient Boosting for region clustering, Random Forest for price prediction), in Karkaria et al. [29], who predict profitability and location feasibility, and in Francisco, Monteiro, and Cardoso [10], who apply probabilistic allocation for scheduling. Even in critical scenarios, AI is directed at probabilistic decision-making rather than predictive monitoring.

This extends to the most recent architectures. Hossen et al. [23] integrate an AI framework with OCPP 2.0.1 to enforce charging schedules automatically, reducing peak load by 18%, cutting average wait times, and raising forecasting accuracy and on-site solar utilisation; yet the intelligence remains aimed at grid stability. Dalamagkas et al. [6] illustrate the same focus in their Open V2X Management Platform: although it integrates LSTM and XGBoost pipelines, that intelligence is applied to user profiling (clustering drivers) and load prediction, not hardware health.

The Conflation of Anomaly Detection with Cybersecurity A critical reading of recent work shows that "anomaly detection" is frequently conflated with cybersecurity rather than physical reliability. Bishal, Hilal, and Thapa [4] use Federated Learning to neutralise network intrusions; Purohit and Govindarasu [48] extend this with FL-EVCS, a decentralised framework trained on the CICEVSE2024 dataset whose target variables remain Distributed Denial of Service and injection attacks; Alauthman et al. [2] apply Reinforcement Learning (Deep Q-Networks) to the same problem; and Thapa, Poudel, and Abouyoussef [55] implement TinyML to detect malware injection and Denial of Service attacks. In the generative-AI domain, Honnalli and Farooq [22] use multimodal LLM for visual reasoning over SoC plots, and Viet Ha The et al. [57] use Generative Adversarial Networks to synthesise attack data for training. Even work explicitly badged as

"EVSE anomaly detection" sits here: Hegde et al. [21] benchmark classical and ensemble learners for classifying network attacks on EVSE traffic, finding ensemble methods the most effective.

The same framing recurs where anomalies are defined as economic damage. Kesavan et al. [30] combine ML models (LSTM, Random Forest, Autoencoders) in their GridSentinel framework to neutralise intrusions and validate the logical consistency of billing data rather than monitoring the charger state. Cumplido, Alcaraz, and Lopez [5] propose a collaborative detection system (CADS4CS) whose voting algorithm targets energy theft and billing fraud; Akhila et al. [1] use an SVM to detect False Data Injection attacks against billing integrity; and Mallaiah et al. [35] and Mitikiri et al. [38] address Charge Profile Manipulation and SoC spoofing respectively. In each case the anomaly is data manipulation causing revenue loss or service disruption, not a physical malfunction.

The cybersecurity focus is reviewed comprehensively by Morgan and Ali [39], who introduce the *Guardian Paradox*: when a DT is deployed to secure the physical hardware, the DT itself becomes a high-value attack target, adding a new layer of vulnerability. From a different angle, Zia et al. [69] propose a Rust-based cloud architecture on AWS focused on software memory safety, criticising Python and C++ implementations for their vulnerabilities and improving CSMS stability but leaving charger monitoring untouched.

The OCPP-security survey by Garofalaki et al. [16] reinforces the point: while it catalogues robust countermeasures against Denial of Service and Man-in-the-Middle attacks, it observes that the security of the energy sensors and controls falls outside the scope of standard OCPP security analysis. Benfarhat et al. [3], securing OCPP communications with Temporal Convolutional Networks, reach a complementary conclusion, recommending the integration of ML to detect anomalies and strengthen threat response capabilities in EV CS environments. Together these confirm a clear gap: while algorithms mature for network-intrusion and financial-fraud detection, their application to internal hardware anomaly detection remains markedly under-researched.

Advanced Component Diagnosis and the Integration Gap Emerging research is beginning to address this hardware gap, though approaches vary widely in sophistication. At the most fundamental level, knowledge-based systems are still used to structure maintenance: Liu et al. [33] build an expert system grounded in Fault Tree Analysis, mapping failure modes into a relational (Access) database, while Yang, Gao, and Duan [60] drive a comparable system with FP-Growth and hard-coded weight-coefficient matrices to trigger safety alarms. Both remain passive tools reliant on static logic, without predictive capability.

To move beyond static rules, researchers turned to statistical methods. Yang and Li [62] propose a Multivariate Gaussian Distribution model, criticising traditional thresholding approaches for failing to detect abnormal points or faults in the charging process when they do not account for the correlation between data. By modelling the correlation between parameters such as voltage and current, they flag outliers that respect individual limits but violate joint patterns, while noting the difficulty of applying such models in practice, where raw data requires preprocessing before analysis. Zhao et al. [68] address a related obstacle: inverter fault signals are often submerged in

non-Gaussian noise from electromagnetic interference, and using Empirical Wavelet Transform and Cyclic Correntropy they extract open-circuit fault features that traditional methods miss; yet their approach is validated solely in Matlab/Simulink.

With the rise of deep learning, neural networks have been applied directly to component telemetry. Pal, Sarker, and Bhattacharya [46] develop an LSTM autoencoder to detect anomalies in the inductor current of a fast-charging converter, and Izquierdo Gómez et al. [27] use neural networks to model the thermal behaviour of DC-DC power converters, identifying temperature stress as a principal cause of failure in power electronic components. Most recently, Dui et al. [9] introduce the CHART framework, showing that a hybrid model achieves 99.33% accuracy, on a real-world operational dataset, over standalone CNN or LSTM baselines, while Gao, Lin, and Yang [14] validate the value of hardware monitoring with an Improved Deep Belief Network, using Random Forest feature ranking to show that DC output voltage and current are the most significant predictors of failure.

A systematic bias nonetheless persists: even these advanced architectures are frequently applied to the EV battery or safety monitoring rather than to the EVSE. Yu [65] proposes a CNN-BiGRU to predict battery health, and Zhang, Ma, and Zhang [67] employ the same combination for a fire-early-warning system. The superiority of deep learning is also not universally established: Yang et al. [61] compare DBN, LSTM, and XGBoost for predicting anomalies in EV charging equipment on a real-world dataset, where LSTM performs poorly (65.37%), followed by DBN (76.82%), while XGBoost reaches 90.34%, confirming that gradient-boosted and ensemble methods such as XGBoost and Random Forest are the strongest performers on this kind of data. In a comparable offline setting, Gao, Yuan, and Zhang [15] report 83% accuracy with an Extreme Learning Machine.

Despite these advances, a critical integration gap remains. Izquierdo Gómez et al. [27], Zhang, Ma, and Zhang [67], Pal, Sarker, and Bhattacharya [46], and Zhao et al. [68] all validate their models in simulation or on historical datasets, operating in isolation. Wei et al. [59] begin to close this gap with an Online Incremental Learning framework that diagnoses faults in real-time at 95.4% accuracy, and Lv et al. [34] propose a perception-edge-cloud architecture that delegates urgent decisions to edge-based Fuzzy Neural Networks (response < 150ms) while reserving the cloud for long-term analysis. Yet the prevailing standard remains offline models that lack the real-time OCPP connectivity required for automated intervention. The need to move from reactive to proactive operation is itself acknowledged in the field: Sanjana et al. [51], for instance, identify the integration of ML as a direction for future improvement.

3.5 Identified Gaps and The Road Ahead

The literature confirms that the standards needed for connectivity and operational orchestration are well established. Recent implementations [24, 40, 47, 49] show that the combination of IEC 61851 and OCPP can manage the communication between the BMS and the EVSE as well as remote control. However, as Morgan and Ali [39] note, these protocols were formulated without

consideration of DT architectures, and this mismatch has led some researchers to bypass OCPP altogether, fragmenting the infrastructure. Yet Sujatha et al. [52] demonstrate that real-time connectivity is achievable within standard frameworks, indicating that the integration problem is one of architecture rather than of protocol capability.

Examining existing DT solutions, the literature shows them to be either high-level twins focused on grid optimisation, automation tools concerned with protocol syntax, or low-level protection systems that react to faults only after they occur. Even the most advanced software-monitoring systems remain passive Digital Shadows. As Martins and Rodrigues [36] observe, current systems lack the monitoring and control mechanisms needed to manage these uncertainties while maintaining optimal infrastructure utilization, confirming the absence of a predictive DT capable of detecting faults and intervening autonomously before they escalate.

The same imbalance characterises the AI literature. Most implementations target grid integration, profit maximisation, and user profiling, treating the CS merely as an interface to be managed rather than a physical asset to be maintained. Where anomaly detection is applied, it is frequently conflated with cybersecurity or financial-fraud prevention. And where learned models are applied to charger fault detection specifically, the work is either confined to static, rule-based systems or evaluated offline, on simulated or historical data, in isolation from the live system. Consequently, even though gradient-boosted and ensemble methods such as XGBoost and Random Forest achieve high fault-prediction accuracy, they lack the connectivity to act: as Martins and Rodrigues [36] note, such approaches often operate in isolation, lacking the integrated approach needed to address the inherent complexity of EV charging management.

Two complementary gaps therefore emerge. The first is integration: the predictive techniques that exist are not coupled, in real-time or over a standard protocol, to the live infrastructure they aim to protect. The second is real-world validation: the surveyed twins and detectors are evaluated almost exclusively in simulation or on isolated datasets, with little evidence of operation against authentic, production charger traffic. Martins and Rodrigues [36] frame the resulting need directly, calling for redundant architectures and predictive maintenance algorithms as essential for downtime minimisation, ideally integrated with OCPP to deliver the real-time, predictive maintenance currently missing from the state of the art. It is precisely these two gaps that this dissertation sets out to close: the architecture presented in Chapter 4 positions a DT as a non-disruptive intelligence layer over the existing CSMS, ingesting live OCPP traffic into a real-time state model, and Chapter 6 validates that pipeline against nine months of genuine production data, addressing the integration and real-world validation gaps that the literature leaves open.

Chapter 4

System Architecture

This chapter presents the architecture of the proposed DT platform, addressing the technical gaps identified in the literature review. Moving EVCI from passive, reactive monitoring to active bidirectional control requires a decoupled, scalable design that must satisfy several constraints simultaneously: continuous high-frequency telemetry ingestion, sub-second end-to-end state synchronisation, and secure command routing back to physical assets, without interfering with the financial and security operations of the legacy CSMS.

This chapter describes the complete target architecture of the platform; the subset realised in the MVP, together with the components deferred to future work, is delineated in Section 5.1.

The chapter is organised in three parts. Sections 4.1 and 4.2 establish the system requirements and the reference architecture; Section 4.3 instantiates that reference into the platform's six-layer architecture and justifies the design decisions behind it, which Section 4.4 then details layer by layer. The remaining sections detail the mechanics of each subsystem: the protocol normalisation pipeline (Section 4.5), the state synchronisation engine (Section 4.6), the closed-loop intelligence and actuation mechanisms (Section 4.7), and the security and compliance framework (Section 4.8).

4.1 System Requirements and Design Principles

The architecture is driven by a set of requirements derived directly from the limitations identified in Chapter 3. The recurring weakness of existing systems, that they operate as passive Digital Shadows, are validated only in simulation, and lack the real-time connectivity needed to act on the infrastructure, translates into a concrete set of functional and non-functional requirements that the design must satisfy. Table 4.1 states these requirements and the section in which each is addressed.

ID	Requirement	Section
Functional		
FR1	The platform must ingest telemetry from heterogeneous physical devices, principally EV chargers speaking OCPP 1.6 and 2.0.1, and normalise it into a single representation independent of protocol version.	4.5
FR2	The platform must maintain a real-time, queryable model of the current state of every entity in the infrastructure: stations, connectors and transactions.	4.6
FR3	The platform must preserve a durable, ordered history of every state transition for audit, fault tracing, and recovery.	4.3.1
FR4	The platform must consume that state to produce predictive and optimisation outputs, such as charging schedules and anomaly signals.	4.7
FR5	The platform must close the control loop by routing commands back to physical devices, so that the DT can intervene rather than merely observe.	4.7.1
Non-functional		
NFR1	A state change at a charger must be reflected in the digital model within a sub-second threshold, the threshold that distinguishes an active DT from a passive shadow.	4.6.2
NFR2	Services must be independently deployable and horizontally scalable: no service shares in-process state with another, and every data domain has a single writer, so services are coupled only through the event bus and explicit read interfaces.	4.4.2
NFR3	No event may be lost, and the live state must be fully reconstructible after the failure of any stateful component.	4.3.1, 4.8.3
NFR4	The platform must coexist with the operator's existing CSMS without assuming responsibility for connection lifecycles or authentication: the legacy system's domains of record.	4.3.3
NFR5	Access must be authenticated and authorised, and data must be isolated between tenants sharing the same deployment.	4.8.1

Table 4.1: Functional and non-functional requirements.

Several of these requirements are met through four cross-cutting principles that recur throughout the architecture. An event-driven backbone decouples producers from consumers and satisfies NFR2. Event sourcing, persisting every transition as an immutable event rather than overwriting state, satisfies FR3 and NFR3, making the live state a disposable, replayable projection. An anti-corruption layer, realised as the adapter pattern, satisfies FR1 by confining all protocol-specific complexity to the ingestion edge. And polyglot persistence, routing current state, durable history, and time-series telemetry to storage engines chosen for each access pattern, resolves the tension between NFR1's latency and NFR3's durability. The remainder of this chapter details each of

these decisions in turn.

4.2 Conceptual Overview

The proposed DT functions as a real-time, bidirectional model of a physical EVCI. It ingests data from every device in the infrastructure, such as chargers, PV arrays, energy meters, and grid connection points, and keeps a synchronised state model of all of them, establishing a closed-loop control mechanism by sending commands back to the physical devices. This system differs from a monitoring system because of its bidirectional communication; it does not only mirror the physical component. As a DT, it presents the current state of the infrastructure, simulates future states, and actively intervenes through the existing CSMS to improve and predict outcomes.

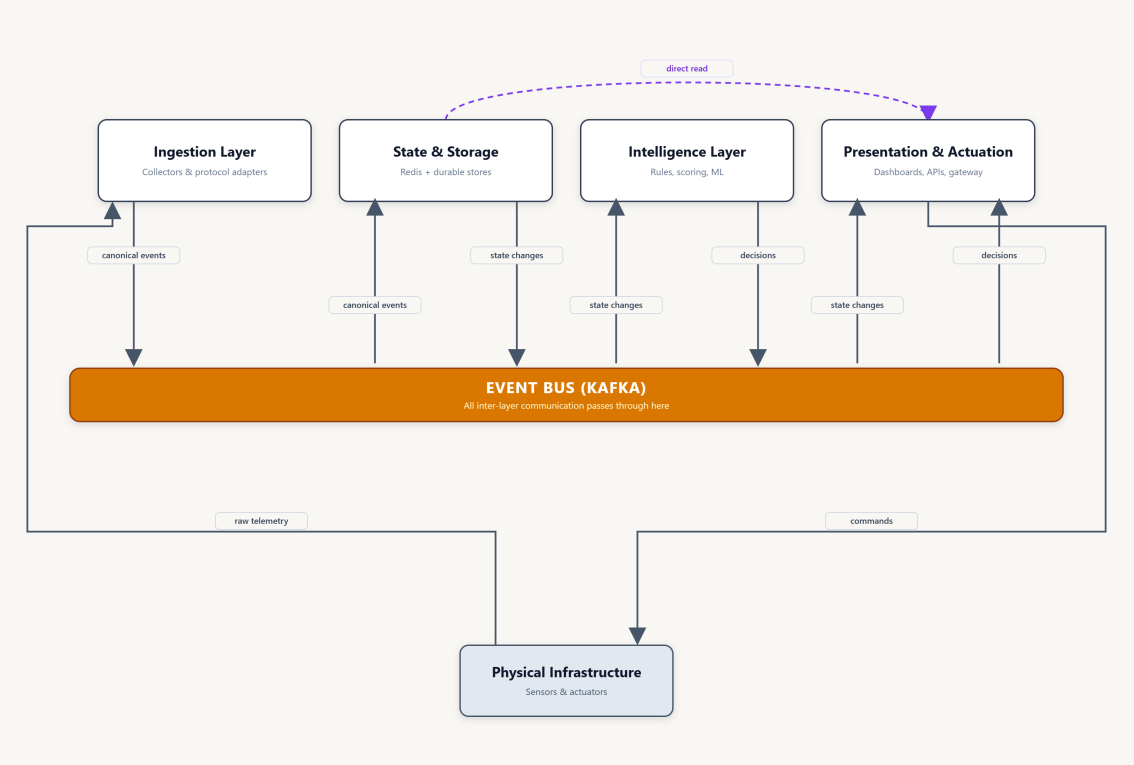

Figure 4.1: Reference architecture of the proposed platform: the closed-loop data flow.

Figure 4.1 presents the reference architecture of the proposed platform. Data originates at the Physical Infrastructure layer and flows into the Ingestion layer, which acts as a strict boundary. This layer absorbs vendor-specific protocols and translates all incoming data into the platform's canonical language, ensuring no downstream services are compromised by hardware complexity. Once normalised, the data enters a central Event Bus. This asynchronous communication backbone is the core of the system, providing critical decoupling between data producers and consumers, and supporting event replay. From the Event Bus, the data flows into the State and Storage layer, which reduces the event stream into a fast in-memory cache of the current physical world, persists every transition to a durable event log, and archives continuous time-series telemetry. The

Intelligence layer, composed of forecasting engines and optimisation schedulers, consumes the digital state to produce charging plans. Crucially, the Intelligence layer never interacts directly with the physical hardware. Instead, it routes its decisions to a Presentation and Actuation layer, which safely translates digital decisions back to physical commands. This establishes a continuous feedback loop where raw telemetry dictates the digital state, the state informs the Intelligence layer, and the Intelligence layer drives physical actuation. Figure 4.1 is deliberately technology-agnostic: it names the six layers, their responsibilities, and the direction of flow between them, but not the services or storage engines that realise them. Section 4.3 instantiates this reference into the platform's concrete six-layer architecture and justifies the design decisions that determine its content; Section 4.4 then details each layer in turn.

4.3 Architecture Overview and Design Decisions

Figure 4.2 instantiates the reference architecture of Figure 4.1 into the platform's concrete six-layer architecture, naming the services and storage technologies that realise each layer. It is presented here, ahead of the decisions themselves, because those decisions are precisely what determines its content: which engines populate the State and Storage layer, which technology carries the event bus, where the boundary with the legacy CSMS falls, and which algorithms the Intelligence layer runs.

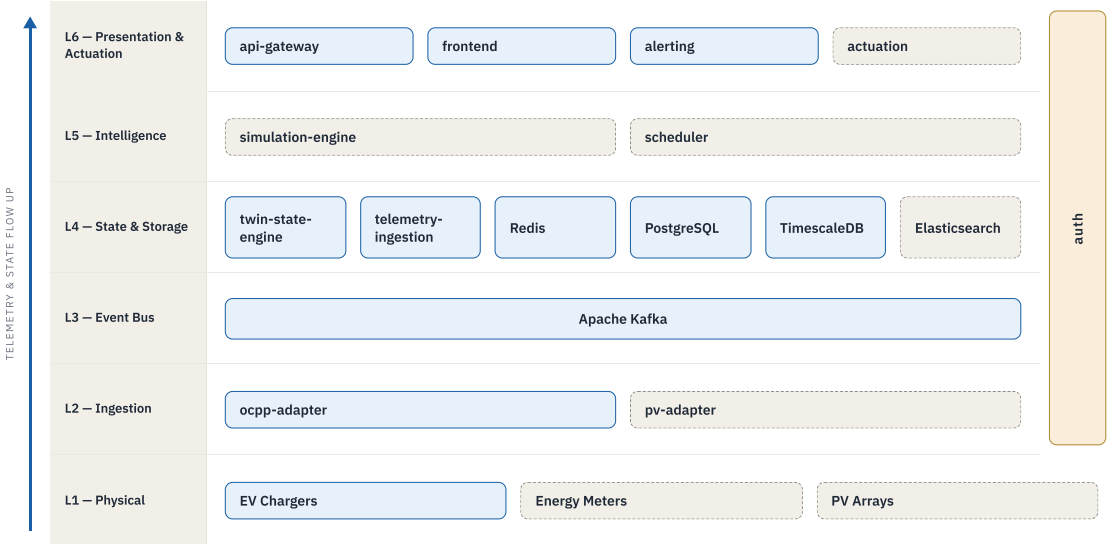

Figure 4.2: The platform's six-layer architecture: the reference architecture of Figure 4.1 instantiated with concrete services and stores.

The figure also delineates the scope realised in this work. Components implemented and evaluated in the MVP are drawn solid; those that belong to the complete target design but are deferred to future work, the `pv-adapter`, the `simulation-engine`, the `scheduler`, the `actuation` service, and the Elasticsearch store, are drawn dashed, and are scoped in detail in

Section 5.1. The `auth` service spans the software layers as a cross-cutting concern, since it secures every one of them rather than occupying a layer of its own.

Translating the conceptual architecture of an active DT into a production system requires significant engineering trade-offs. To satisfy the previously outlined requirements, several conventional architectural patterns were discarded in favour of decoupled solutions. This section details the fundamental architecture design decisions that shape the proposed platform, justifying the selection of the technology stack and integration strategies. Specifically, it explores the adoption of event sourcing for state management (Subsection 4.3.1), the persistence strategy for high-frequency telemetry (Subsection 4.3.2), the integration framework with legacy CSMS infrastructure (Subsection 4.3.3), and the phased algorithmic approach driving the Intelligence layer (Subsection 4.3.4).

4.3.1 Event Sourcing and Event Backbone

Traditional state-based (Create, Read, Update, and Delete (CRUD)) architectures rely on in-place updates, which overwrite prior states and hide the chronological sequence of system transitions. For a DT, where the `twin-state-engine`, a service inside the State and Storage layer, must maintain the real-time state of every infrastructure entity, losing the exact history of state changes renders it impossible to perform historical audits or trace hardware faults.

To satisfy the requirement for high-fidelity historical tracing, the traditional state-based model was discarded in favour of event sourcing. Rather than storing only the current state, every state transition is persisted as an immutable event in an append-only log in PostgreSQL, while the current state is maintained as a materialised view in Redis for fast reads. This paradigm provides two critical capabilities. The first is point-in-time reconstruction: recovering the exact state of any entity at an arbitrary past timestamp by replaying its events up to that point. The second is idempotent replay: although the Redis view is written first and serves as the runtime read source of truth, it can be reconstructed in full from the durable log, so it remains a disposable projection, rebuilt on start-up and after any loss.

Event sourcing governs how state is stored; a second, independent decision governs how events are transported between services. The platform requires a decoupled, durable event bus, for which Apache Kafka was selected for its durability, partition-based ordering, and replay support (Subsection 4.4.2). Partition-based ordering is essential in an EV-charging context: the sequential meter values of a given charger must be processed in order, since processing them out of order would corrupt energy calculations; partitioning the inbound-event topic by charger enforces this. Kafka further ensures events are never lost while a consumer is temporarily offline, and allows a new service, such as a future ML model, to bootstrap by replaying a topic's history from its earliest offset.

Finally, allowing multiple decoupled services to exchange events as untyped payloads introduces a risk of silent data corruption. The target design mitigates this with strict data contracts, enforced by a schema registry built on Protocol Buffers, so that breaking schema changes would

be caught automatically during Continuous Integration (CI) rather than surfacing as runtime deserialisation failures. As detailed in Subsection 4.5.2, the MVP realises these contracts as JSON, with the Protobuf-based registry deferred to future work.

4.3.2 Time-Series Data Storage: TimescaleDB vs InfluxDB

The platform is required to ingest continuous, high-frequency time-series telemetry, including OCPP meter values (power, energy, voltage, current, frequency, and SoC) and PV data (power output, irradiance, and temperature). For a deployment of 1,000 chargers, this results in substantial data volumes, estimated at 10,000 to 50,000 data points per minute. This telemetry needs to be written with low latency, queried efficiently across time ranges, aggregated into time buckets, and crucially, joined with relational data for analytics. These constraints prompted an evaluation of two time-series databases, InfluxDB and TimescaleDB, summarised in Table 4.2.

Criterion	**InfluxDB**	**TimescaleDB**
Relational JOINs	Performed in application code	Native SQL JOINs, indexed on both sides
Efficiency at extreme scale (>1M points/min)	Higher (columnar engine)	Adequate at the target scale
Disk footprint	Lower (native columnar)	Higher (row-based; columnar compression available)
Ecosystem	Purpose-built time-series database	PostgreSQL extension (familiar SQL and tooling)

Table 4.2: Time-series database comparison for the platform's telemetry.

TimescaleDB was selected, deployed as a PostgreSQL extension on a dedicated instance isolated from the operational PostgreSQL that holds entity state and configuration. The decisive factor was native SQL JOIN capability: relational analytics, correlating telemetry with dimension data such as tariffs and topology, are expressed directly in SQL, with indexes on both sides of the join, rather than fetched in separate queries and joined in application code as InfluxDB requires. This advantage is realised within the telemetry instance, so the dimension tables an analytical query needs are co-located with, or replicated into, the telemetry store; the rarer joins that must reach the separate operational instance use PostgreSQL foreign-data wrappers. InfluxDB's columnar engine is more efficient at extreme scale (over one million points per minute) and more compact on disk, but neither advantage binds at the platform's target scale, where TimescaleDB's columnar compression closes most of the storage gap. Reusing the PostgreSQL ecosystem further favoured TimescaleDB, which proved the most suitable solution for the platform's constraints.

4.3.3 The Digital Twin - CSMS Relationship

An existing CSMS manages all charger connections and handles OCPP 1.6 and 2.0.1 compliance. The proposed DT must consume all OCPP traffic, maintain entity state, optimise charging, and issue commands, raising the fundamental question of how the DT and the existing CSMS coexist. Three integration options were evaluated, summarised in Table 4.3:

Option	Mechanism	Key trade-off	Verdict
A - Replace CSMS	Chargers connect directly to the `ocpp-adapter`, part of the Ingestion Layer; the DT replaces the CSMS entirely	Total control, but a high-risk migration that loses existing charger authentication databases and configurations	Rejected
B - Passive Observer	Chargers stay on the CSMS; the DT receives a read-only copy of OCPP traffic	Low risk and faster to build, but read-only: a sophisticated monitoring system that cannot actuate	Rejected
C - CSMS as Transport, DT as Intelligence	Chargers stay on the CSMS; the DT taps inbound traffic and issues commands via the CSMS management API	Non-disruptive and bidirectional-capable	Selected

Table 4.3: Integration options for the DT-CSMS relationship.

To minimise operational risk while supporting an active DT, Option C was selected: it is the only option that is both non-disruptive and bidirectional-capable. It enforces a strict separation of concerns: the legacy CSMS retains responsibility for protocol compliance, connection lifecycles, and authentication, while the DT assumes responsibility for state management, predictive intelligence, and optimisation decisions. Although Option A is rejected as an integration strategy, the adapter still implements direct connection as one of several runtime modes alongside the Option C tap modes. The direct mode, in which CSs connect to the adapter and it acts as their CSMS, is reserved for the PoC evaluation, where no legacy CSMS is present (Subsection 5.4.1).

Depending on the capabilities of the legacy CSMS, the inbound read path can be implemented via native Kafka topic exports, Hypertext Transfer Protocol (HTTP) webhooks, or, as a fallback, a transparent WebSocket proxy. The webhook mode of this read path is exercised against a production CSMS in Section 6.3, validating the integration strategy on nine months of real OCPP 2.0.1 traffic. The outbound write path completes the bidirectional design by routing commands through the CSMS management API but its implementation is deferred to future work (Section 5.1); the MVP therefore validates only the inbound path.

4.3.4 The Optimisation Strategy

The `scheduler` service is the DT component responsible for optimising EV charging across a given site. This is fundamentally a constrained mathematical optimisation problem that must balance competing objectives: maximising the utilisation of local PV generation, minimising grid energy costs, respecting physical grid capacity limits, and meeting driver SoC targets. The `scheduler` belongs to the platform's target architecture and is deferred beyond the MVP (Section 5.1); the design rationale for its optimisation strategy is nonetheless presented here for architectural completeness.

Three algorithmic paradigms were evaluated for this problem, summarised in Table 4.4. A mixed-integer Linear Programming (LP) solver represents the mathematically optimal long-term solution, but it is opaque to operators, and its constraints (site grid limits, available PV generation, tariff structures, and per-vehicle departure deadlines) must be formulated and validated against each site's own behaviour. Deploying such a solver before that site-level operational baseline exists risks silent misconfiguration and hard-to-diagnose infeasibility, so a transparent, predictable engine is the safer starting point.

Paradigm	**Strengths**	**Limitations**	**Role**
Rule-based heuristics	Transparent, fast, predictable	Sub-optimal solutions	Phase 1
Mathematical optimisation (LP)	Mathematically optimal	Opaque; needs baseline data; infeasibility risk	Phase 2
ML (e.g. Reinforcement Learning)	Adaptive	Opaque, data-hungry, hard to validate	Not adopted

Table 4.4: Algorithmic paradigms evaluated for the scheduler's optimisation strategy.

Therefore, a phased approach was adopted to build algorithmic transparency. The first phase specifies a priority-ordered rule-based engine, whose rule set is detailed in Subsection 4.7.2. Once a site has accumulated sufficient operational data to validate the physical constraints, the design transitions to the LP Solver of the second phase, retaining the deterministic rule-based engine as a reliable fallback for solver timeouts or mathematical infeasibility.

Finally, a non-negotiable constraint on this strategy is support for manual override. If facility managers cannot reliably override the autonomous schedule, they will inevitably lose trust and disable the system; the `scheduler` is therefore designed to respect manual configurations, ensuring operational safety and long-term adoption.

4.4 The Six-Layer Architecture

The six layers introduced in Figure 4.2 each carry a single, clear operational responsibility. The figure presents the model as a vertical stack: the physical devices form the base and each succes-

sive layer builds on the one below up to the Presentation and Actuation layer at the top. Telemetry and state flow upward through the stack: raw device data is normalised, reduced to state, persisted, interpreted, and finally presented, while the control commands that close the loop flow back down as actuation commands issued from the Presentation and Actuation layer. Because layers communicate only through the event bus rather than through direct calls, hardware complexity is abstracted at the ingestion edge and never propagates into the platform's core logic.

The following subsections detail the layers in turn, with the Physical Infrastructure and Ingestion layers treated together (Subsection 4.4.1): the Event Bus (Subsection 4.4.2), State and Storage (Subsection 4.4.3), Intelligence (Subsection 4.4.4), and Presentation and Actuation (Subsection 4.4.5).

4.4.1 Layers 1 and 2: Physical Infrastructure and Ingestion

The Physical Infrastructure layer constitutes the physical boundary of the system, composed of the real-world assets this platform models: EV chargers operating via OCPP 1.6 or 2.0.1, PV inverters, energy meters, and grid connection points. This hardware remains outside the platform's control: the platform observes it through the ingestion path and acts on it only through the actuation path.

Directly above it, the Ingestion layer is responsible for protocol normalisation through the adapter pattern. Dedicated services translate the native protocols of each source into the platform's canonical schema (Section 4.5). Consistent with the integration strategy of Subsection 4.3.3, in production the `ocpp-adapter` consumes the OCPP traffic tapped from the CSMS rather than connecting to chargers directly; the `pv-adapter`, part of the target design and deferred beyond the MVP, would interact directly with PV inverters. Consequently, nothing downstream of the Ingestion layer ever deals with hardware complexities, raw OCPP frames, or vendor-specific payloads. This modularity ensures that integrating a new device class, such as a BMS, requires only a dedicated adapter, preserving the integrity of the core platform without spreading protocol-specific logic across other services.

4.4.2 Layer 3: Event Bus

The Kafka event bus is the platform's central asynchronous communication backbone. Producers, such as the ingestion adapters, and consumers, such as the `twin-state-engine` and the `alerting` service, never address one another directly: they exchange messages only through the bus and share no databases. This indirection design keeps the services strictly decoupled, so that the failure of any single service is contained rather than propagated to the others.

The event-driven design yields two principal benefits. First, services can be added or removed without disrupting the rest of the platform, since none depends on another's availability or internal state. Second, because the bus retains recent event streams, a new or recovering consumer can replay them to catch up, while the durable event log (Subsection 4.3.1) preserves the complete history for full state reconstruction and audit. The bus is partitioned by charger, which preserves

the ordering of each charger's events while allowing throughput to scale horizontally across partitions; message delivery is at-least-once, and its behaviour under sustained load is evaluated in Chapter 6. To prevent silent data corruption as messages cross service boundaries, every message carried by the bus conforms to the platform's canonical schema, whose structure and encoding are detailed in Section 4.5.

4.4.3 Layer 4: State and Storage

To prevent any single database from becoming a bottleneck, the State and Storage layer follows a polyglot persistence strategy: each kind of data is routed to a storage technology chosen for its access pattern, as summarised in Table 4.5. Ownership here is defined per data domain rather than per physical engine: each domain has a single writer, the service responsible for it, so write paths stay decoupled. Reads are shared where it is efficient: the Redis live-state view and the time-series store are queried directly by the services that present or evaluate that data, while charger connectivity and operational topology are reached through explicit APIs. No service writes to a domain it does not own, and all cross-service communication flows through the event bus (Subsection 4.4.2).

Three technologies serve the MVP. Redis holds the current state of every entity as a materialised view, reconstructible in full from the log on recovery. PostgreSQL is the system of record: it maintains the durable, append-only event log required by the event-sourcing architecture (Subsection 4.3.1) and, on its operational instance, the relational configuration and topology data. TimescaleDB, deployed as a PostgreSQL extension on a dedicated instance isolated from that operational database (Subsection 4.3.2), holds time-series telemetry optimised for high-throughput ingestion and the time-range aggregation. A fourth store, an Elasticsearch index of raw OCPP frames for full-fidelity audit, belongs to the target design and is deferred to future work (Subsection 4.8.2).

Store	Technology	What lives here	Access pattern
Current state	Redis	Entity current states (connector status, active transactions)	Sub-millisecond key lookup
Event log	PostgreSQL	State-change events (append-only, event sourcing)	Sequential append; point-in-time replay
Configuration & topology	PostgreSQL	Sites, stations, connectors, users, tenants, tariffs	Relational queries
Time-series telemetry	TimescaleDB	Meter values, energy telemetry	Time-range queries, aggregations
OCPP audit log	Elasticsearch	Raw OCPP messages (full fidelity)	Full-text search, forensics

Table 4.5: Polyglot persistence: storage technologies and their roles.

4.4.4 Layer 5: Intelligence

The Intelligence layer will be the autonomous decision-making layer of the DT. It will consume real-time state from the event bus and historical data from the stores to drive its analytical functions. It will comprise two components: the `scheduler`, which produces optimised charging plans (Subsection 4.3.4), and the `simulation-engine`, which produces energy-production and demand forecasts. Beyond these, the layer is the architectural home for the predictive, anomaly-detection intelligence; while an online detector is deferred to future work, this capability is explored within the present work through the offline ML PoC evaluated in Chapter 6.

Consistent with the platform's separation of concerns, this layer will never communicate with physical devices. It will write its decisions as events on the event bus, from which downstream services persist them and perform any physical execution. This keeps the Intelligence layer as a pure producer of analysis, decoupled from both storage ownership and actuation.

4.4.5 Layer 6: Presentation and Actuation

The Presentation and Actuation layer presents the DT to human operators and is designed to act back on the physical infrastructure, closing the system's control loop. Its presentation side is realised in the MVP, while its actuation side belongs to the target design and is deferred to future work. On the presentation side, the `api-gateway` serves the operator UI and third-party integrations, while operator awareness is completed by the `alerting` service, which continuously monitors the event stream and notifies operators when telemetry crosses critical thresholds, so that degraded conditions surface without anyone having to watch the dashboard. On the actuation side, the `actuation` service will consume the optimisation decisions produced by the `scheduler` from the event bus and issue the corresponding commands through the CSMS management API; the CSMS then delivers them to the chargers over its existing connections (Subsection 4.3.3), completing the bidirectional interaction.

4.5 Protocol Normalisation and Ingestion

Introduced as Layer 2 in Section 4.4.1, the Ingestion layer is examined here in detail. Where that section positioned it as the platform's anti-corruption boundary, this section details how the boundary is realised: how raw OCPP traffic is parsed, validated, normalised into the canonical schema, and published to the event bus, before any state synchronisation or optimisation can take place. The following subsections detail the mechanism in turn: the inbound processing pipeline (Subsection 4.5.1), the normalisation of OCPP 1.6 and 2.0.1 into the canonical schema (Subsection 4.5.2), and the modelling of real-time charger connectivity (Subsection 4.5.3).

4.5.1 Inbound Processing Pipeline

As established in Subsection 4.3.3, the platform follows a non-disruptive integration strategy (Option C) in which the legacy CSMS acts as the transport layer and the DT as the intelligence.

Chargers therefore maintain their connection to the CSMS, while the `ocpp-adapter` taps inbound traffic through one of the integration modes enumerated in Subsection 5.4.1, with the direct mode, in which the adapter terminates charger connections itself, reserved for the PoC evaluation where no legacy CSMS is present.

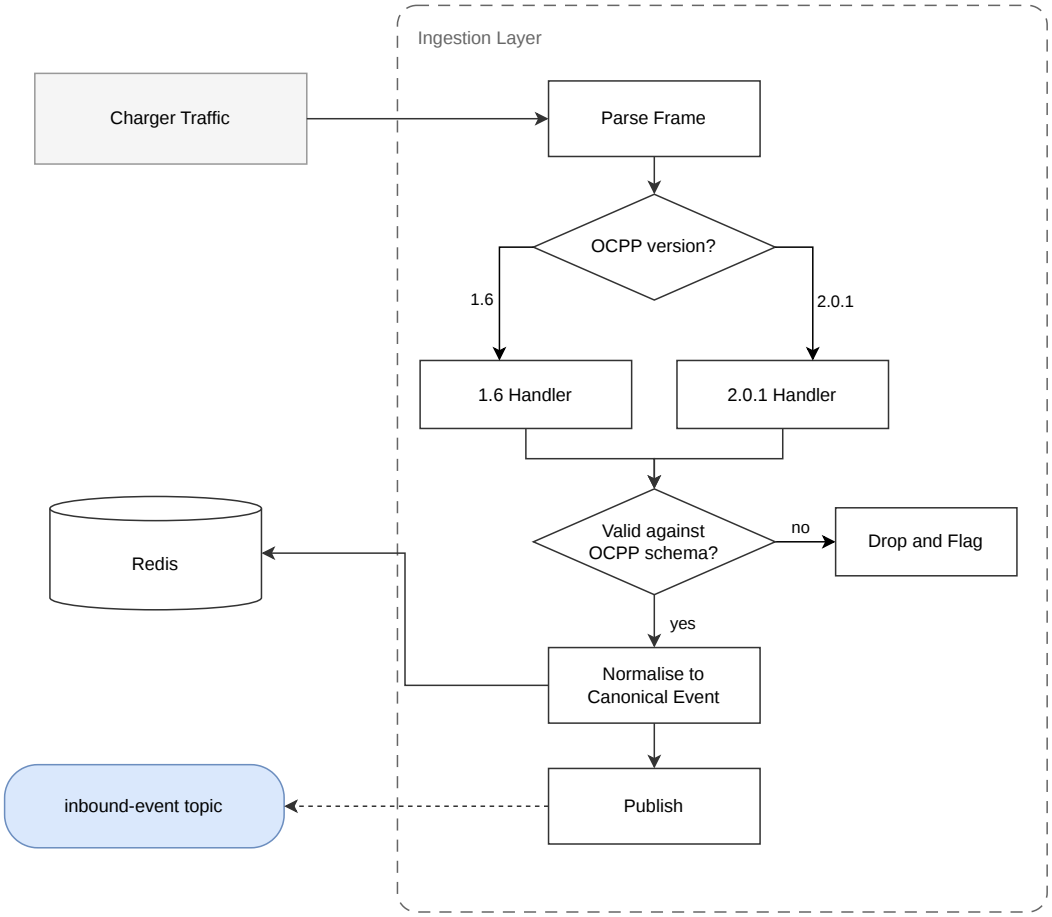

Figure 4.3: The OCPP ingestion pipeline inside the OCPP Adapter service.

Figure 4.3 traces the lifecycle of a message once it reaches the `ocpp-adapter`. The frame is first parsed: the adapter classifies it as a request, a response, or an error and extracts the requested action [43, 45]. It is then routed to the OCPP 1.6 or 2.0.1 handler according to the charger's registered protocol version and the handler validates the payload against the corresponding OCPP schema. Because protocol compliance remains the responsibility of the CSMS under Option C, an invalid frame is flagged and dropped internally rather than answered back to the charger; only the in-line proxy mode is positioned to return a protocol-level rejection at all. A valid message is then translated into the platform's canonical event representation and published to the inbound-event topic, partitioned by charger (Subsection 4.4.2).

A distinction is worth drawing between the two checkpoints in this pipeline. Validation enforces strict protocol validity and discards any structurally invalid frame. Normalisation, by con-

trast, is tolerant: it never discards a structurally valid message, and where an individual field cannot be mapped, an unrecognised status code, for example, it degrades the event rather than failing, a design rule detailed in Subsection 5.4.1.

4.5.2 Unifying OCPP 1.6 and 2.0.1

There is a significant architectural gap between OCPP 1.6 [44] and 2.0.1 [42]: where 1.6 relies on a key-value structure, 2.0.1 uses a richer object-oriented model. Forcing downstream services to handle both logic branches would couple the entire platform to these protocol variations, violating the separation of concerns. The Ingestion layer's normalisation step closes this gap by mapping both structures onto a single canonical event, so that every downstream service consumes one uniform representation regardless of the wire protocol. In the target design this canonical schema is defined in Protocol Buffers and governed by a schema registry; the MVP serialises it as JSON (Section 5.3).

The mapping collapses semantically equivalent messages onto a common event. The command that starts a charging session, for example, is a `StartTransaction` message in OCPP 1.6 but a `TransactionEvent` in 2.0.1; both are normalised to a single canonical event. Table 4.6 generalises this across the principal messages of each version, showing how two structurally distinct protocols converge onto one internal representation. The result is that the `twin-state-engine` operates agnostically: it processes state transitions and maintains the state model without needing to know which OCPP version a given charger speaks.

OCPP 1.6 message	**OCPP 2.0.1 message**	**Canonical event**
StartTransaction	TransactionEvent(eventType=Started)	TransactionEvent(phase=STARTED)
StopTransaction	TransactionEvent(eventType=Ended)	TransactionEvent(phase=ENDED)
MeterValues	MeterValues / TransactionEvent	MeterValues
StatusNotification	StatusNotification	StatusNotification
BootNotification	BootNotification	BootNotification
Heartbeat	Heartbeat	Heartbeat
Authorize	Authorize	Authorize

Table 4.6: Normalisation of OCPP 1.6 and 2.0.1 messages into the platform's canonical events.

4.5.3 Modelling Charger Connectivity

An active DT must maintain a real-time understanding of physical connectivity to interact safely with the infrastructure. Because the chargers remain connected to the CSMS under Option C, the DT does not own those connections; instead, the `ocpp-adapter` models their state from the tapped traffic in a dedicated connection registry. The registry records each charger with its OCPP version, its CSMS session identifier, and the time of its last heartbeat. OCPP chargers emit

periodic heartbeats to signal that they remain reachable, and the registry ties a Time-To-Live (TTL) to this signal: every heartbeat refreshes the entry's TTL, while an absence of heartbeats for longer than twice the configured interval expires the entry, at which point the charger is treated as offline. This registry is a distinct, short-lived use of Redis: unlike the materialised state view of Subsection 4.4.3, it is driven by live heartbeats rather than rebuilt from the event log.

This connection state is a safety-critical input to the rest of the platform. When a charger's entry expires, its transition to offline is published as a device-status event on the event bus. In the MVP, the `alerting` service consumes this event and notifies operators of the disconnection; in the target design, the `actuation` service also consumes it, withholding commands from an unreachable device (Subsection 4.7.1). By making physical connectivity a first-class element of the digital state, the platform avoids a dangerous failure mode: directing a charging command at hardware that is no longer listening.

4.6 Real-Time State Synchronisation

Once the Ingestion layer has normalised physical telemetry onto the event bus, the platform must construct a real-time memory of the physical infrastructure. This section details the `twin-state-engine`, focusing on how the event-sourcing paradigm adopted in Subsection 4.3.1 is realised to deliver sub-second state synchronisation and a durable history. The following subsections trace the synchronisation path from telemetry ingestion to state reduction (Subsection 4.6.1), and then examine the latency budget that holds the platform within its sub-second target (Subsection 4.6.2).

4.6.1 Telemetry Ingestion to State Reduction

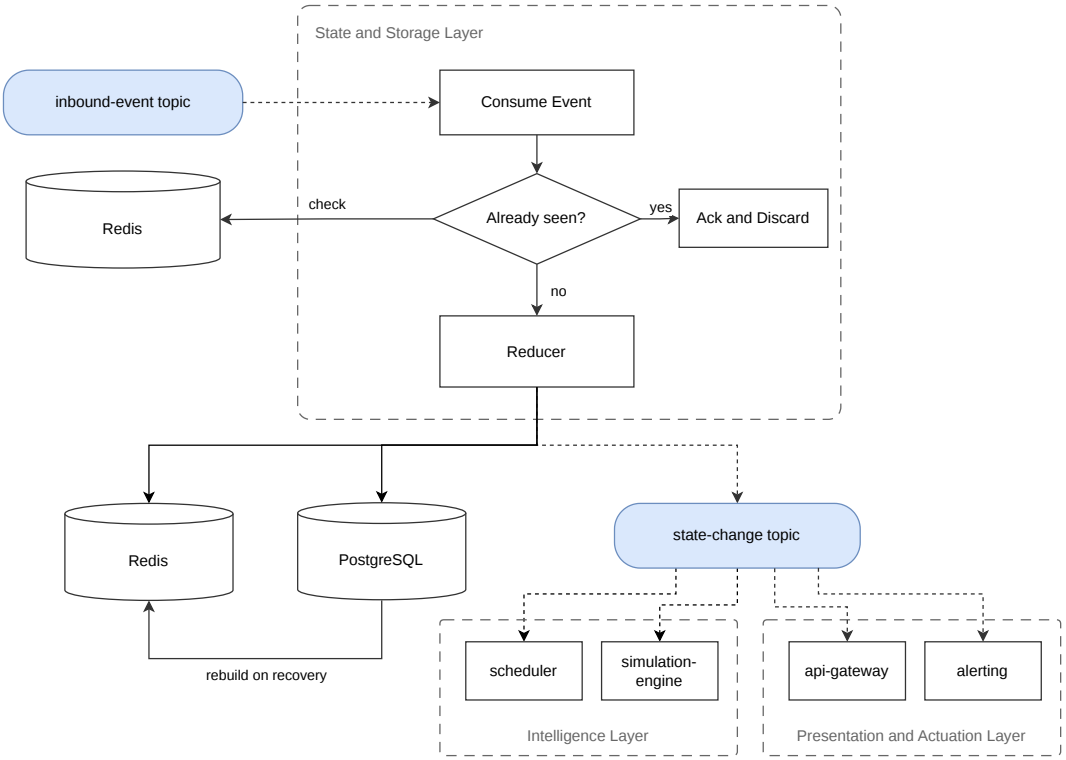

Figure 4.4: The Twin State Engine service.

Figure 4.4 represents the synchronisation process performed by the `twin-state-engine`, which begins by consuming the normalised canonical events from the inbound-event topic. Because the event bus provides at-least-once delivery and chargers retransmit messages they believe were not acknowledged, the same event may arrive more than once, and the engine must process it idempotently. Every event carries a unique identifier; the engine maintains a short-lived deduplication window in Redis and checks each identifier before processing. A recognised duplicate is acknowledged and discarded, while a new event proceeds to the state transition.

Rather than mutating state in place as a conventional CRUD system would, the engine applies the event through a deterministic reducer: a function that takes the current state and the incoming event and returns a new state. A connector in the *Available* state with no active transaction, for example, transitions to the *Preparing* state on receiving a *'preparing'* status update. Because every state is derived by applying an ordered sequence of events, the platform retains an immutable audit trail of every transition. As established in Subsection 4.3.1, this supports point-in-time state reconstruction, recovering an entity's exact state at any past timestamp, together with historical playback for the twin visualisation and the ability for new analytical services to bootstrap by replaying the event log.

Each transition is committed to the platform's stores in a fixed order with deliberate failure semantics. The live state in Redis is written first: because the live twin answers every real-time query, it is the read source of truth, and a failed Redis write aborts the transition. The engine then appends the transition as an immutable record to the append-only event log in PostgreSQL and, for transactional entities, upserts a relational transaction view that the `api-gateway` serves to clients as a query model; finally, it publishes a state-change event to the state-change topic, from which the `api-gateway` pushes live updates to the operator UI and the `alerting` service raises notifications. In the target design the same topic feeds the Intelligence layer: the `scheduler` consumes state changes to trigger re-optimisation (Subsection 4.7.1), and the `simulation-engine` consumes them in its validation mode (Subsection 4.7.3). These durable-store steps are deliberately tolerant of a transient PostgreSQL failure: downstream consumers keep receiving updates, and any row the database briefly misses is reconstructed by replaying the event stream. Durability therefore rests on the event stream as the ultimate record: Redis and the relational view are disposable projections, rebuilt from the log on recovery (Subsection 4.8.3).

4.6.2 End-to-End Latency Budget

To reason about the platform's real-time capability, the lifecycle of an event, from a physical state change to its reflection on the operator UI, can be decomposed into a per-stage budget. Table 4.7 lists the processing cost contributed by each stage of the path. The budget begins once the event reaches the `ocpp-adapter`, and excludes the CSMS's own receive-and-forward latency, which precedes the tap under the Option C integration (Subsection 4.3.3).

Stage	**Component**	**Cost (ms)**
Charger payload reaches the adapter	network	1-5
Validate and normalise to the canonical schema	ocpp-adapter	1-3
Publish event to the broker	Kafka	2-5
Consume, reduce, write live state and append to log	twin-state-engine	5-15
Publish state-change event	Kafka	2-5
Route and push over WebSocket	api-gateway	1-3
Parse and render the interface update	frontend	16-50
Total		**28-86**

Table 4.7: End-to-end latency budget: per-stage processing cost, from the adapter to the operator interface.

These per-stage figures account only for the processing cost of an event at each service, summing to a theoretical best-case of 28-86 milliseconds under ideal, unloaded conditions. The latency a consumer actually observes is substantially larger and dominated by factors absent from

these compute figures: broker transport and consumer scheduling on the event bus, client-side rendering, and, under concurrent load, contention among co-located services. Accordingly, the realised median measured under load is several times this best case (Section 6.2), which is why the architectural requirement is expressed as a threshold rather than a point estimate: end-to-end synchronisation within one second, the bound that distinguishes an active DT, whose state can be acted upon in real time, from a passive Digital Shadow.

4.7 Bidirectional Control and Intelligence

The Ingestion and State layers of preceding sections establish the high-fidelity Digital Shadow. A true DT, however, must also close the control loop, acting on the synchronised state rather than merely mirroring it. This section presents the capabilities that close that loop. Four functions are involved: charging optimisation, produced by the `scheduler` on the strategy established in Subsection 4.3.4; energy and demand forecasting, produced by the `simulation-engine`; anomaly detection over device telemetry; and the execution of physical commands through the `actuation` service.

As established in Subsection 4.4.4, these components belong to the platform's target design and are deferred beyond the MVP (Section 5.1); their design is presented here for architectural completeness. The anomaly-detection capability is explored ahead of its online realisation through the offline ML PoC evaluated in Chapter 6.

4.7.1 The Actuation Loop

The actuation loop would close the control loop, carrying a decision from the Intelligence layer back to the physical infrastructure. It would begin with the `scheduler` emitting a charging decision in response to a change in state; consistent with the separation of concerns (Subsection 4.4.4), the `scheduler` never contacts chargers directly, delegating all physical execution to the `actuation` service.

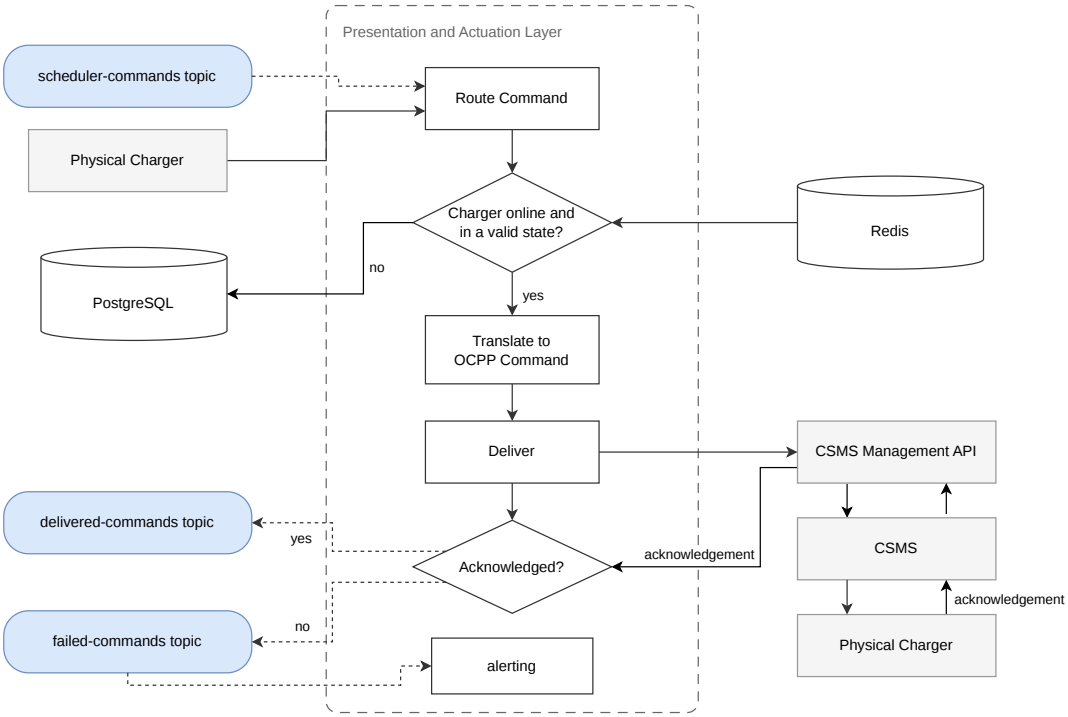

Figure 4.5: The Actuation service.

Figure 4.5 traces the actuation flow as it would operate. An abstract command, whether emitted by the `scheduler` or issued by an operator, would first be routed for execution. Before delivery, a validation stage would confirm, from the connection state (Subsection 4.5.3), that the target charger is online and in a valid operational state; this prevents commands from reaching faulted or disconnected hardware, and a command that failed this check would be rejected and the rejection recorded for audit. Once the preconditions hold, the abstract decision would be translated into a concrete OCPP command. Consistent with the Option C integration strategy (Subsection 4.3.3), the command would then be issued through the CSMS management API, which delivers it to the charger over the connection the CSMS already maintains; in proxy-mode deployments, where the `ocpp-adapter` terminates charger connections directly, it would instead be delivered over that adapter's own WebSocket. Finally, the service would await the physical acknowledgement: on success it would publish a delivered-command event, and on timeout or error a failed-command event accompanied by an operator alert, closing the loop in either case.

4.7.2 Multi-Source Optimisation

The `scheduler`'s objective would be the continuous optimisation of charging across a site. As established in Subsection 4.3.4, computing an optimal charging profile is a multi-objective constraint problem: it would balance maximising PV self-consumption, minimising grid energy costs, and ensuring every vehicle reaches its target SoC by departure. To do so, the `scheduler` would

draw on several live sources, such as current entity state, PV telemetry, energy-tariff configuration, and grid-capacity limits, which is what makes the optimisation multi-source.

Within the phased solver strategy of Subsection 4.3.4, the Phase 1 rule-based engine would evaluate the priority-ordered set of Table 4.8, from the highest priority to the lowest. The Safety rule sits at the top: it would never permit the grid-connection or charger power limits to be exceeded, and no lower rule could override it. At the opposite end, the Minimum Charge rule holds the lowest priority, guaranteeing every connected vehicle a baseline rate only once all higher constraints are satisfied. This ordering is what would make the engine's behaviour predictable: a higher-priority safety constraint always takes precedence over a lower-priority optimisation goal, such as absorbing surplus PV energy, so the system would never trade safety for efficiency.

Priority	Rule	Action
1	Safety	Never exceed the grid connection limit or the charger's rated power
2	Departure Guarantee	Increase power if a session is projected to miss its SoC target at current rate
3	Grid Capacity	Reduce lowest-priority sessions if total site load exceeds 90% of the grid limit
4	PV Excess	Increase charging power to absorb surplus PV production
5	Cost Optimisation	Reduce non-urgent sessions to minimum power during high-tariff periods
6	Minimum Charge	Guarantee every connected vehicle a configurable minimum rate

Table 4.8: Phase 1 scheduler: priority-ordered rule set.

The Phase 2 LP solver (Subsection 4.3.4) would replace this rule set with a rolling horizon optimisation once sufficient operational data exists, falling back to the rule-based engine on solver timeout or infeasibility. Under either solver, the manual operator overrides established as inviolable in Subsection 4.3.4 would still hold: a connector's manually set charging level would remain untouched until the override expires or is cancelled.

4.7.3 Forecasting and Expected vs Actual Monitoring

To operate proactively, a DT must be able to forecast future states and to validate its internal model against physical reality. Both capabilities would be provided by the `simulation-engine`, which would run in two modes. In forecast mode it would produce, on a rolling 15-minute cadence, a four-hour forward view of infrastructure state to feed the `scheduler`. In validation mode it would run continuously, comparing a short-term prediction of each entity's next state against the telemetry that subsequently arrives. The predictions would be grounded in physical

models rather than learned from data: expected PV production, for instance, would be computed from real-time irradiance together with hardware parameters such as panel area, temperature coefficients, and inverter efficiency curves. By comparing this expected state against the observed telemetry on the event bus, the engine would surface a residual; if a charger delivered less power than the `scheduler` requested, or a PV array underperformed its irradiance-based prediction, it would publish an anomaly event. Because this comparison is continuous, it would allow the DT to flag developing hardware anomalies earlier than the threshold-based monitoring of conventional systems.

This physics-based residual monitoring is one of two complementary approaches to anomaly detection that the Intelligence layer is designed to host. The other is data-driven: a supervised-learning model trained on charging-session telemetry. It is the data-driven approach that is explored in this work, as the PoC implemented in Section 5.7 and evaluated in Section 6.4; the physics-based mode described here remains future work.

4.8 System Resilience, Security, and Compliance

A DT is only as robust as its fault tolerance, security, and auditability. Because the platform mirrors live physical infrastructure and is ultimately designed to act upon it, every component must fail safely, and every interaction must be authenticated and traceable. Table 4.9 summarises the principal failure modes the architecture anticipates and the mitigation each relies on; mitigations that depend on the deferred `actuation` and `scheduler` services are marked as part of the target design. The remainder of this section then examines the most significant mechanisms in detail: certificate-based authentication and transport security (Subsection 4.8.1), the immutable raw-message archive that sustains compliance and audit (Subsection 4.8.2), and the event sourcing-based recovery that allows the state engine to rebuild its in-memory state after a crash without data loss (Subsection 4.8.3).

Failure	**Mitigation**
Kafka consumer lag	Events queue durably in Kafka and are never lost; the consumer catches up when capacity recovers
State-engine crash	Redis is rebuilt from the PostgreSQL event log on restart; the service serves traffic only after recovery (Subsection 4.8.3)
Charger disconnect	The connection registry's TTL marks the charger offline, so downstream services avoid commanding unreachable hardware
Actuation timeout *target*	A failed-command event raises an alert and informs the `scheduler` (Subsection 4.7.1)
TimescaleDB slowness	`telemetry-ingestion` buffers and applies back-pressure; telemetry writes are best-effort and never block state updates

Table 4.9: Failure modes and their architectural mitigations.

4.8.1 Authentication and Transport Security

Security spans two distinct domains. The first is the platform itself, secured by the `auth` service: it issues and validates JSON Web Tokens (JWTs), enforces role-based access control, and isolates data between tenants, so that every request between the platform layers, and from external clients, is authenticated and authorised. As a cross-cutting concern, it secures every software layer without occupying one of its own. The second domain is the link to the chargers, which under the Option C integration strategy (Subsection 4.3.3) is owned by the CSMS: the chargers connect to the CSMS, and it is the CSMS that terminates their transport security and authenticates them. The relevant standards are nonetheless part of the platform's design, because the `ocpp-adapter` assumes this role in proxy-mode deployments, where it terminates charger connections directly. The core OCPP 1.6 specification [44] offers only HTTP Basic Authentication, which carries well-known weaknesses: credentials can leak into server logs unless carefully redacted, and the protocol provides neither per-message signing nor certificate-based mutual authentication. Modern deployments therefore target OCPP 2.0.1 Security Profile 3 [42], which secures the channel with Transport Layer Security (TLS) and authenticates the charger with a client-side certificate (mutual TLS). Where the adapter terminates the connection under this profile, charger credentials need never be held locally: on each connection attempt the adapter would validate the certificate against a secure endpoint on the `auth` service, so that a compromised charger can be revoked centrally and locked out in real time.

4.8.2 The Raw OCPP Messages Archive

Disputes over what a charger actually reported are inevitable when operating charging infrastructure: an operator questioning the meter values behind a session, or a regulator requiring evidence of what a device communicated. To support auditing and dispute resolution independently of its own processing, the platform is designed to keep an immutable archive of raw messages. Before the `ocpp-adapter` normalises an incoming OCPP frame (Subsection 4.5.1), it would duplicate the unaltered payload to a dedicated raw-archive topic, which would in turn be indexed into an Elasticsearch store under a retention policy aligned with local regulations. Captured upstream, and so unaffected by the platform's normalisation and state-reduction logic, the result is an immutable record of exactly what each physical device communicated. This archive complements, rather than replaces, the authoritative transaction records held by the CSMS: it provides an independent trail for compliance and for corroborating disputes. As noted in Subsection 4.4.3, the Elasticsearch store belongs to the platform's target design and is deferred beyond the MVP.

4.8.3 State Engine Recovery and Event Replay

Any real-time platform must tolerate component failure: in a conventional system, a cache eviction or a state engine crash means lost state. The DT avoids this because the Redis view is a disposable projection of the durable event log (Subsection 4.6.1). On restart, the `twin-state-engine` does not resume from the live Kafka stream, whose retention is bounded, but first reconstructs

the Redis view from the event log, restoring the latest persisted state of every entity; only then does it mark itself ready and resume consuming live events. Recovery is therefore loss-free and deterministic.

4.9 Summary

This chapter has presented the architecture of the proposed DT as a decoupled, event-driven platform designed to move EVCI from passive monitoring to active, bidirectional control. From the requirements established in Section 4.1, four design decisions follow (Section 4.3): event sourcing makes every state transition an immutable, replayable fact; polyglot persistence routes live state, durable history, and time-series telemetry to storage engines matched to each access pattern; the CSMS-as-transport strategy (Option C) lets the twin coexist with the existing infrastructure rather than replace it; and a phased optimisation strategy begins with transparent rule-based scheduling before introducing mathematical optimisation.

These decisions are organised into six decoupled layers (Section 4.4), connected through a Kafka event bus rather than direct service-to-service calls. The Ingestion layer (Section 4.5) acts as an anti-corruption boundary, normalising both OCPP 1.6 and 2.0.1 into a single canonical schema so that no downstream service depends on protocol details. The `twin-state-engine` (Section 4.6) reduces this event stream into a real-time state model within a sub-second threshold, while remaining recoverable from its durable log. Above it, the intelligence and `actuation` services (Section 4.7) close the control loop, optimising charging, issuing commands through the CSMS, and continuously comparing the twin's expected state against physical reality to detect anomalies. The resilience, security, and audit mechanisms of Section 4.8 support the platform's robustness.

The architecture described here is the platform's complete target design. The remainder of this dissertation concerns the subset realised and validated as an MVP: Chapter 5 details the implementation of the ingestion, state, delivery and PoC intelligence components, while the deferred elements (`scheduler`, `actuation`, and `simulation-engine`) are scoped in Section 5.1 and revisited as future work.

Chapter 5

Implementation

Chapter 4 defined the target architecture; this chapter describes how it is realised as a working MVP. The implementation delivers the platform's ingestion, state-synchronisation, and delivery path, together with a PoC of its intelligence layer, while the active control loop is deferred as future work. The chapter first defines the scope of the MVP (Section 5.1), then sets out the development methodology and the technology stack (Sections 5.2, 5.3). It then documents the six implemented back-end services in turn: from OCPP ingestion and normalisation, through the event-sourced state engine and time-series telemetry persistence (Section 5.4), to the gateway, authentication, and alerting layers (Section 5.5). Section 5.6 details the frontend implementation and Section 5.7 presents the implemented ML PoC. Throughout, the engineering decisions are justified and, where a mechanism specified in Chapter 4 is realised in simplified form, the simplification is stated explicitly.

5.1 MVP Scope and Boundaries

Chapter 4 describes the complete target architecture; this section delimits the subset realised as the MVP. The MVP must get reliable data from the physical infrastructure into a synchronised model, guaranteeing end-to-end propagation within one second, and must produce value operators can act on. Conversely, it is not responsible for production-scale operation, fleet-scale real charger validation, or real-time automated actuation. Scope was deliberately restricted to the components needed to demonstrate those obligations; everything else was deferred to future work under one of four rules:

a) The component would require operator trust before automated operation could be validated;

b) It would require production-scale operational data to be meaningful;

c) It would not be load-bearing for the MVP's empirical claim;

d) Its tooling cost exceeds its contribution to MVP velocity.

Table 5.1 lists the deferred components and the rule under which each falls, while Table 5.2 summarises the services that are delivered.

Deferred Component	**Rule**	**Reason for Deferral**
Scheduler & Smart Charging	(a)	Requires operator trust in the data model before automating decisions. Building it in the MVP would risk operators disabling the DT.
Actuation	(a)	Highest-risk component - wrong decisions affect physical devices. Needs a proven model first.
PV & Energy Device Adapters	(c)	Valuable but not load-bearing for the twin's core proof. OCPP data is the primary entry.
Simulation Engine & Forecasting	(b)	Requires months of historical data to be meaningful. Building it blind now would lead to discarded effort once real telemetry becomes available.
Scenario Runner UI	(b)	Depends on simulation engine.
3D Visualisation	(d)	A well-executed 2D live view is more operationally useful and needs less effort.
Grafana Dashboard Embedding	(d)	Good charts inside the app are sufficient for MVP analytics.
Elasticsearch	(c)	Compliance tool, not a value-proving tool. Structured logs are enough now.
Full Multi-Tenant Management UI	(c)	The data model is already designed for multi-tenancy, but the management UI is not needed in the MVP.
OCPP Credential Management UI	(d)	Handled via `auth` service configuration, not a UI for MVP.
Continuous Aggregate TimescaleDB Views	(b)	Raw table queries are fast enough at MVP scale. Materialised views will be added as future work.

Table 5.1: Deferred components.

Service	Responsibility
auth	Issues and validates JWTs; gates access to the management surface.
ocpp-adapter	Receives OCPP 1.6/2.0.1 traffic, normalises to canonical events, publishes to Kafka.
twin-state-engine	Applies events to entity state, persists the event log, updates Redis.
telemetry-ingestion	Batch-writes meter values to TimescaleDB for time-series analytics.
api-gateway	Serves REST and a Kafka-to-WebSocket bridge for the frontend.
alerting	Evaluates four deterministic rules over the event stream.
frontend	Next.js UI with four views: Login, Live Twin, Transactions, Alerts.
Offline ML PoC	Anomaly-detection classifier trained on TimescaleDB telemetry; validates that the platform's time-series storage supports ML integration.

Table 5.2: The services delivered in the MVP and their responsibilities.

Figure 5.1 shows how these services connect end to end, from the physical charging infrastructure to the operator interface. Data enters the `ocpp-adapter`, which normalises OCPP 1.6 and 2.0.1 messages into canonical events and publishes them to Kafka; both the `twin-state-engine` and the `telemetry-ingestion` service consume from that topic in parallel. The `twin-state-engine` reduces events to per-entity state, writing live state to Redis and appending the durable event log to PostgreSQL, and publishes state changes to a second Kafka topic, from which the `api-gateway` delivers them to connected clients; the `telemetry-ingestion` service batch-writes the raw meter values to TimescaleDB. Each service is independently deployable and owns a single concern.

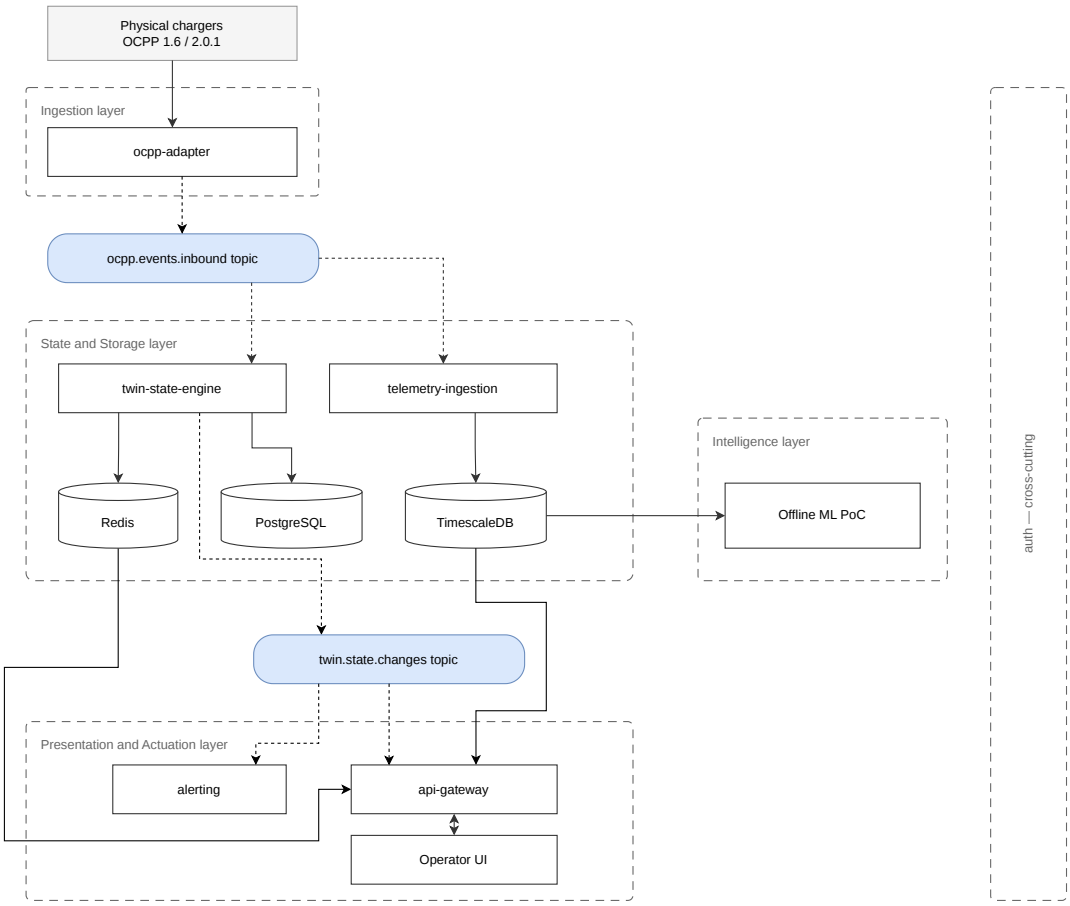

Figure 5.1: The implemented MVP data flow, from charger ingestion to the operator interface.

The remainder of this chapter details each component in turn, beginning with the development methodology and infrastructure choices in Sections 5.2 and 5.3.

5.2 Development Methodology & CI/CD

The project followed a merge request workflow on GitLab: every change reaches the protected integration branch through a merge request opened from a short-lived feature branch, never through a direct push. Direct commits to the protected branch are blocked at three layers: locally through pre-commit hooks, on the server through branch protection, and in the pipeline through a validation job, so the rule cannot be bypassed at any single layer. Branch names and commit messages follow enforced conventions, keeping the history consistent and machine-readable for tooling. Each merge request is gated before it can merge: the pipeline must pass and every review thread must be resolved.

CI runs as a multi-stage pipeline, with each service's jobs scoped to the files that affect it. A lint stage enforces formatting and static type checking; a test stage runs unit tests together with integration tests, the latter exercising service code against real Kafka, PostgreSQL, TimescaleDB,

and Redis instances spun up as containers rather than mocks, so the tests reflect production behaviour; and a build stage produces one container image per service, gated so that images are built only after the tests pass. The same checks run locally as pre-commit hooks, so most failures are caught before they reach CI.

Daily developer operations are unified behind a project `Makefile` that exposes each task: stack setup, service start-up, data seeding, simulation, and load testing, as a single command, with a profile flag that scopes every command to the MVP subset. The same tooling and compose definitions are structured to extend to the full platform, although only the MVP subset is operational; the deferred services have defined slots but are not implemented.

Major architectural decisions were recorded as Architectural Decision Records (ADRs), capturing the context, the alternatives considered, the decision taken, and the consequences accepted; these support the design decisions of Chapter 4, so that every architectural claim is backed by a versioned, dated rationale. Six ADRs were authored across the MVP development: five for decisions realised in the MVP: the monorepo structure, event sourcing, Kafka as the event bus, TimescaleDB for time-series storage, and CSMS integration strategy; and one forward-looking record for the scheduler's solver strategy.

5.3 Core Infrastructure Stack

The platform runs on Docker Compose for both local development and CI testing, orchestrating the backing and application services as a single project; the same definitions are structured to extend to the full-platform, although only the MVP subset is operational. The backing infrastructure follows a polyglot-persistence strategy: an event bus, three storage engines each matched to an access pattern, and an in-memory cache, described in turn below.

Apache Kafka is the event bus. The inbound-events topic is partitioned by charger (Subsection 4.3.1), preserving the per-charger ordering the state engine relies on. Messages are serialised as JSON on the wire; the Protobuf canonical schemas are already defined for the schema registry rollout planned as future work.

PostgreSQL is the platform's system of record, holding three logically distinct concerns: the operational topology (sites, stations and connectors), the append-only event-sourcing log, and the authentication and alerting tables. A tenant identifier is propagated through every table, so the schema is multi-tenant ready although the MVP runs as a single tenant; the schema is created on first start-up from a one-shot initialisation script, with a migration tool configured but left unused at MVP scale, where the init scripts suffice. Time-series telemetry is held in a separate TimescaleDB instance, deliberately isolated from the operational database so that high-frequency meter value writes do not contend with the relational reads of the system of record. Its readings live in a time-partitioned hypertable, indexed for the per-session and per-charger range queries that the transaction view and the ML-PoC rely on; pre-computed downsampled views are defined in the schema but disabled at MVP scale, where raw queries are fast enough. Finally, Redis serves three roles: a sub-millisecond cache of current entity state, the OCPP connection registry (whose

entries expire on a heartbeat time-to-live), and a short deduplication window for idempotent event consumption. Redis is intentionally not persisted to disk: as a materialised view, it is rebuilt from the PostgreSQL event log on restart (Subsection 4.8.3).

This stack is sufficient for the fifty-charger evaluation presented in Chapter 6. Production deployment targets Kubernetes, with the infrastructure-as-code left as future work. Three deliberate simplifications hold at MVP scale: a single Kafka broker, a single PostgreSQL instance without read replicas, and a non-clustered Redis, each a scoped constraint rather than an architectural limit: the design retains the horizontal-scaling paths (broker replication and partition redistribution, PostgreSQL read replicas, a Redis cluster) needed for the full platform, as documented in Section 7.3.

5.4 Ingestion and State Engine Implementation

The protocol path from charger to durable state, represented in Figure 5.1, is handled by three cooperating back-end services, detailed in turn below: the `ocpp-adapter`, which normalises OCPP traffic into canonical events; the `twin-state-engine`, which reduces those events to per-entity state; and the `telemetry-ingestion` service, which writes the raw time-series readings to TimescaleDB. Each is independently deployable and owns a single concern.

5.4.1 OCPP Adapter Implementation

The `ocpp-adapter` is the platform's single entry point for charger protocol traffic. Whatever its source, every OCPP 1.6 and 2.0.1 message leaves the adapter as a normalised canonical event on the inbound-events topic, decoupling every downstream service from the details of the charging protocol. How the adapter acquires that traffic, however, varies by deployment mode (Table 5.3).

Mode	How it works
direct	Chargers connect directly and the adapter is the CSMS. Used in PoC.
proxy	Adapter sits in front of an existing CSMS and taps traffic.
webhook	Existing CSMS posts raw OCPP frames to `POST /ingest`. No WebSocket server.
kafka	Existing CSMS publishes to a Kafka topic. No WebSocket server.

Table 5.3: OCPP Adapter integration modes for acquiring OCPP traffic.

The four modes differ only in how raw OCPP traffic reaches the adapter; the output is always identical. In the `webhook` and `kafka` tapping modes the legacy CSMS retains the charger connection and forwards copies, so the adapter runs no OCPP server and the CSMS owns the protocol handshake. In `proxy` and `direct` modes the adapter instead terminates the charger connection itself, speaking OCPP over WebSocket and handling the request-response handshake; on each connection it negotiates the subprotocol and checks the charger's identity, which in the MVP is a basic

authentication simplification, with validation against a production credential store left as future work. `direct` mode, in which the adapter fully replaces the CSMS, is reserved for the PoC evaluation, and `proxy` for a fallback in front of an existing CSMS. Because `twin-state-engine` and `telemetry-ingestion` consume only the inbound-events topic, they are entirely unaware of which mode produced it.

Where the adapter terminates the charger connection, each OCPP message is handled in two stages. The adapter first performs the minimum work needed to construct a valid acknowledgement and returns it immediately, so the charger never waits on downstream I/O for its response; only after the acknowledgement is sent does it normalise the payload and forward it to Kafka. This keeps all Kafka I/O isolated from the OCPP handshake path: if the broker is temporarily unavailable the publish fails and is logged, but the charger session continues unaffected.

Normalisation is deliberately forgiving. If a message type has no registered handler, or a handler fails, the adapter emits a degraded event (the base event with an empty payload) rather than dropping the message; the design rule is that normalisation never raises, because a downstream consumer can discard unknown fields but cannot reconstruct a missing event. The normaliser also absorbs the structural differences between protocol versions: the connector-status sets of OCPP 1.6 and 2.0.1, which differ in size, are mapped onto a single status space so that the state engine never branches on protocol version, and the differing shapes of the boot-notification payload are handled transparently.

Transaction identity requires careful handling in OCPP 1.6, where the CSMS assigns an integer transaction identifier that every subsequent message in the session references. The DT needs a stable, human-readable, URL-safe string identifier that it can store and use directly in frontend URLs, so the adapter synthesises one from fields the charger supplies at session start. Because the subsequent stop and meter value messages still carry the integer, the adapter keeps a per-session mapping from that integer to the synthesised identifier and rewrites it on the way out; without this, a stop message would reference an identifier the state engine has never seen, and the session would appear never to end. OCPP 2.0.1 removes the problem entirely, carrying a stable string identifier from the outset.

In the `webhook` and `kafka` modes, a second topic also receives a verbatim copy of each OCPP frame before normalisation, a compliance and debugging affordance; indexing this raw stream into a long-term audit store is deferred to future work (Subsection 4.8.2). The terminating modes omit the copy, since there the frames are processed inline rather than arriving as serialised payloads.

Finally, the `ocpp-adapter` owns the connection registry introduced in Subsection 4.5.3. Because TCP does not reliably signal a disconnect, charger liveness is tracked by the heartbeat TTL described there, with a clean disconnect additionally removing the entry immediately. The registry is held in Redis and exposed through a small management API, so other services, including the gateway's twin-view endpoints, can query connectivity without touching the database.

5.4.2 The State Engine

The `twin-state-engine` is the computational core of the DT. It consumes the canonical events published by the `ocpp-adapter` and reduces them to typed per-entity state, maintaining the live state of each entity in Redis, the append-only event log in PostgreSQL, and a stream of state-change notifications on a second Kafka topic for downstream consumers. It models the infrastructure with three entity types: charging station, connector, and transaction (Table 5.4).

Entity	Key fields	Identified by
Charging station	Vendor, model, firmware version, OCPP version, online, last boot, last heartbeat	Charger ID
Connector	OCPP status enum, error code, error info, active transaction ID	Key (ChargerID-EVSEID-ConnectorID)
Transaction	Status, ID, started at, ended at, meter start, meter stop, stop reason	Canonical ID string

Table 5.4: Entity types maintained by the Twin State Engine.

At the heart of the service is a pure-function reducer: given an incoming event and the current state of the entities it touches, it returns a description of what changed and performs no I/O of any kind. This is a deliberate design choice, by keeping all reading and writing out of the reduction logic, the rules that govern how the DT evolves become trivially unit-testable (state in, changes out, no mocks) and the I/O is concentrated in a thin surrounding pipeline. Each change carries the specific fields that differed, so downstream consumers can react selectively without re-inspecting the full state. As in the adapter's normaliser (Subsection 5.4.1), unknown event types and any handler failure yield no changes rather than raising, so a single malformed event can never stall the stream.

Transaction events are the most complex case, because a single OCPP message changes two entities at once. Starting a session creates a transaction entity and, in the same step, records that transaction on the connector; ending one closes the transaction, writing the stop reading, reason, and end time, and clears it from the connector. Both protocol versions route to the same internal start/end logic, so the version is invisible downstream. OCPP 1.6 adds one wrinkle: its stop message omits the connector, so the engine recovers it from the transaction state stored when the session opened. Meter value messages are treated as a pass-through, forwarded to consumers with the latest reading attached, without creating a separate entity or a relational write.

That hidden dependency, needing state to discover which entity a change applies to, is why state is loaded in up to three passes. The entities an event touches are implicit in its payload rather than declared up front, so the pipeline first runs the reducer against empty state purely to learn which entities it reaches, loads those from Redis, and runs it again with real data. This suffices for almost every event; the only exception is the OCPP 1.6 stop case above, where resolving the

connector requires a further load and a third run. Any change that still references an entity with no prior state is silently skipped rather than written.

Once the final set of changes is known, each is applied in a fixed order: Redis, the PostgreSQL event log, the relational transaction view, and finally the state-change topic, with Redis-first failure semantics established in Subsection 4.6.1. One implementation refinement applies here: the transaction-view upsert is skipped for pure meter value updates, so high-frequency telemetry never churns that view.

Every status change passes through a connector state machine before it is written. The machine encodes the connector lifecycle over the single canonical status produced by the adapter (Subsection 5.4.1), so it never branches on protocol version; it knows which transitions are expected, with fault and unavailable states reachable from anywhere, consistent with the specification's treatment of a fault as a global interrupt. Crucially, an unexpected transition is logged but still applied, never blocked. This is the central decision of the module: real chargers routinely skip states, jumping straight from charging to available, or beginning to charge without a preparing step, and refusing such an event would silently desynchronise the live twin from the physical device, whereas recording an unexpected transition is always recoverable.

Two mechanisms make the engine robust to the realities of its inputs. The first is the idempotency guard introduced in Subsection 4.6.1: a single atomic Redis operation records whether an event has already been processed, deduplicating both Kafka's at-least-once delivery and OCPP devices' own retries. Its window is deliberately finite, roughly a day, so that a full replay with reset offsets can rebuild state from scratch. The second is startup recovery: when enabled, the engine rehydrates Redis from PostgreSQL before consuming any new events, fetching only the latest state per entity in a single windowed query rather than replaying history event by event. Recovery therefore scales with the number of distinct entities, not the total volume of events, so Redis is always disposable: a pod restart, an eviction, or a full cache flush is not a data-loss event, because the live twin is restored from the durable log within seconds (Subsection 4.8.3).

5.4.3 Telemetry Ingestion

The `telemetry-ingestion` service is the platform's bulk-persistence layer. While the `twin-state-engine` reduces events to typed entity state, `telemetry-ingestion` has a single concern: writing the raw time-series readings carried by those same events directly into TimescaleDB at high throughput. It maintains no in-memory state model and does not contribute to the live twin view; its output is the durable historical record that downstream analytics and the ML PoC query as their input dataset.

The service is organised as two parallel ingestion pipelines, one per stream it consumes, each pairing a parser with a target table. A single consumer subscribes to both streams under one consumer group and dispatches each message to the right parser by source, simpler than running two independent consumers, with one offset cadence and one configuration to reason about. The first pipeline extracts meter readings from the same canonical event stream the state engine consumes, since OCPP messages embed telemetry alongside protocol state; the second is reserved for raw

measurements from non-OCPP devices such as PV inverters, battery storage, and grid meters. Only the first pipeline is exercised end to end in the MVP: the second is fully implemented and unit-tested but no producer is wired up, present in the codebase as the structural foundation for the bidirectional loop described in Section 4.7.

Most of the parser's complexity comes from OCPP delivering meter readings through three different payload shapes rather than one: the standalone periodic meter values message in OCPP 1.6, the terminal readings embedded inside a 1.6 session-end message, and the readings carried inside the unified transaction event in OCPP 2.0.1. The 2.0.1 case is further complicated by inconsistent field-naming conventions in the upstream payloads, so the parser probes the plausible variants defensively and falls back to an empty list rather than failing. From every shape it walks the same nested structure and emits one row per sampled value, applying the same forgiving rule established in the adapter: a malformed timestamp falls back to the receipt time and an unparseable or unlabelled value is dropped, because losing one reading from a 500-row message is far cheaper than losing the whole message. Each row is typed and ordered to match its target table, so the writer can hand the rows to the database with no further translation, and each measurement column is nullable, since different device types report different subsets of measurands.

The persistence layer is the architectural centrepiece of `telemetry-ingestion`. The naïve approach, one insert per row, would not survive even moderate load, because TimescaleDB writes are bursty and per-row prepared-statement overhead dominates at high throughput. The service instead writes through PostgreSQL's COPY, which transmits an entire batch as a single typed stream and bypasses the parser, planner, and executor work that insert repeats per row. On the development hardware this is roughly a twenty-five-fold speedup: about 2 ms versus 50 ms for a 500-row batch. At the MVP's expected steady-state load this margin is not needed; the COPY choice exists so that the same persistence architecture scales to ten times the load without revisiting the writer.

In front of the writer sits a batching layer that decouples ingestion from persistence. Incoming rows are appended to per-pipeline in-memory buffers (kept separate so a slow flush on one pipeline cannot stall the other) by non-blocking calls, so the consumer keeps fetching from Kafka while a flush is in progress and the writer drains at its own pace. Because every buffer is touched from a single event loop, no locking is required. A background task flushes on a fixed interval, draining each buffer in batch-sized chunks and issuing one COPY call per chunk against a pooled connection; if a buffer has grown large, a single interval can issue several COPY calls. The connection pool is sized to overlap COPY calls across concurrent flushes, so the writer keeps up during bursts rather than serialising on one connection. Error handling is intentionally lenient: a failed COPY is logged and that batch is dropped while the loop continues, on the same reasoning as elsewhere in the platform: a single 500-row batch is recoverable from the Kafka topics, whose retention covers any plausible recovery window, whereas crashing the writer would lose far more and demand operator intervention.

Without a brake, a slow TimescaleDB would let the in-memory buffers grow unboundedly until the service exhausted its memory budget. The Kafka consumer prevents this with pull-side

backpressure: when pending rows cross an upper threshold it stops fetching and polls until the background flush has drained the buffers below a lower one. This is loss-free because Kafka offsets are committed only after a successful poll: any message not fetched during a pause simply waits on its partition. The gap between the pause and resume thresholds is deliberate hysteresis, preventing the consumer from flickering between paused and active on every flush under sustained near-saturation load. The same pending-row signal also drives the service's health endpoint, which reports a degraded status once the backlog builds, so external monitoring can detect a sustained backpressure event even with no error in the logs.

Two design choices follow from the nature of the data. Unlike the `twin-state-engine`, this service maintains no idempotency guard: time-series rows are idempotent by nature (a duplicate reading at the same time, charger, and measurand is either an exact replica, which is harmless, or a near-duplicate that downstream queries can deduplicate cheaply), so the consumer runs with periodic auto-commit and at-least-once delivery is sufficient. Layering an explicit deduplication check on top of bulk COPY writes would add latency to every batch and partly defeat the point of using COPY at all. Finally, the service shuts down cleanly: on stop, the background task is cancelled and a final flush drains both buffers before the pool is closed, with the running totals recorded in the shutdown log.

5.5 Real-Time Delivery and Access

The three services described in this section sit at the boundary between the back-end pipeline and the end user. The `api-gateway` is the sole entry point for browser clients, exposing a REST API for configuration data and historical queries and a WebSocket endpoint for live state updates. The `auth` service issues and verifies the JWTs that authenticate every request to the gateway. The `alerting` service consumes the same state-change Kafka topic that drives the WebSocket bridge and applies a small set of rules to detect operationally significant conditions (faulted connectors, offline chargers, stalled sessions) emitting durable alert records and email notifications. Two patterns recur across the layer: every service verifies JWTs locally against `auth`'s public key, and the `api-gateway` and `alerting` both read live state from Redis and consume the state engine's output topic, while `auth`, the platform's trust anchor, stays off both.

5.5.1 API Gateway

The `api-gateway` is the only service browser clients communicate with directly. It exposes a REST API for configuration data and historical queries and a single WebSocket endpoint for live state, and internally bridges several back-end stores (the PostgreSQL configuration database it owns, the Redis live-state view written by the `twin-state-engine`, the time-series readings in TimescaleDB, and the state-change Kafka topic), presenting them as one unified, authenticated surface. Table 5.5 lists the REST routers.

Router	What it serves
Sites	Site configuration
Charging stations	Stations and live state from Redis
Connectors	Connectors and live state from Redis
Transactions	Session history from PostgreSQL, meter samples from TimescaleDB
Alerts	Alert records proxied from the `alerting` service's API

Table 5.5: API Gateway REST routers.

The gateway's signature responsibility is merging configuration from PostgreSQL with live state from Redis at request time. Neither store alone is sufficient for the UI: PostgreSQL holds the static topology (which stations exist, where they are located, what hardware they advertise) but knows nothing about whether a connector is currently charging, while Redis holds the live state written by the `twin-state-engine` but contains no provisioning data. The gateway joins them per request, returning a fully hydrated entity under a single call rather than forcing the frontend to make two requests and merge client-side. The connector listing adds one deliberate subtlety: it also surfaces entities that exist in Redis but not yet in PostgreSQL (a race that occurs right after a charger's first boot, when the `twin-state-engine` has written connector state but an operator has not yet provisioned the topology), so the UI never briefly hides a freshly booted charger. The transactions endpoint extends the same idea across three stores at once, federating the session row from PostgreSQL, the meter samples from TimescaleDB, and the active session's last reading from Redis into a single object that the UI can render as one session timeline.

Every request to the `api-gateway` authenticates by validating its bearer JWT locally, against the `auth` public key loaded once at startup and held in memory, rather than calling the `auth` service on each request. An invalid or expired token is rejected without revealing why it failed, and administrative routes are additionally gated on the token's role claim. This local verification, repeated across every JWT-consuming service, pays off twice: verification capacity scales with the number of gateway replicas rather than concentrating on a single `auth` service, and brief `auth` outage does not break in-flight requests, since already-issued tokens keep verifying while the in-memory key is valid. The `auth` service is therefore only on the critical path for login, refresh, logout, and the rare key-rotation event (Subsection 5.5.2), which occur on a far slower timescale than per-request authentication.

WebSocket connections cannot reuse that mechanism, because browsers do not allow an authorization header to be set on a WebSocket handshake. The endpoint therefore accepts the JWT as a query parameter and validates it before accepting the upgrade; a missing or invalid token closes the connection rather than completing the handshake. On success the connection is registered with the tenant claim extracted from the validated token and stored alongside the socket for its lifetime, so every later routing decision reads that stored value rather than the token, and a token expiring mid-session never reclassifies an existing subscription. A hard cap on concurrent

connections guards the event loop against a misconfigured client.

The connection registry is held in memory, each entry pairing the socket with its tenant and a list of subscriptions. Subscriptions are additive and scoped to everything in the tenant, to a single site, or to a single station, and a newly accepted connection receives nothing until the client explicitly subscribes, so a client that cares about one station is never sent the tenant's entire event firehose. When the bridge hands an event to the registry, two filters apply in order. Tenant isolation comes first and is unconditional: there is no path by which a connection can receive another tenant's events, even with a malformed subscription. Scope matching follows: a station-scoped subscription matches the station and its connectors (Subsection 5.4.2), and a site-scoped one matches any station in that site, resolved through a station-to-site lookup the bridge maintains. Connections that match nothing are skipped, and any socket that fails mid-send is disconnected after the broadcast completes.

The link from Kafka to those sockets is a single background task in the gateway's own event loop, yielding frequently so it never blocks HTTP traffic. It consumes the state-change topic from the latest offset (WebSocket clients need live events, not history, which the REST routers already serve), and a transient Kafka error triggers a bounded exponential back-off rather than crashing the bridge or the HTTP server. The bridge does not forward Kafka payloads verbatim: it transforms each state-change envelope into a compact push carrying only the per-field before/after diff, dropping the always-changing timestamp as pure noise on the wire. A second background task refreshes the station-to-site map roughly once a minute, so site-scoped matching stays an in-memory lookup rather than a per-broadcast database query, and a newly provisioned station becomes routable within a minute without a restart.

Startup and shutdown are orchestrated by an ordered lifespan. On the way up the gateway opens its PostgreSQL pool, then the TimescaleDB pool, connects to Redis, loads the `auth` public key, constructs the connection registry, starts the Kafka bridge, and only then marks itself ready. The health endpoint reports not-ready until that final flag flips, and afterwards includes the current live-connection count so monitoring can watch saturation. Shutdown reverses the order, stopping the bridge before the pools so no in-flight broadcast races against a closing connection.

5.5.2 Auth

The `auth` service issues access tokens after verifying user credentials and exposes the public key used by every other service to verify those tokens. It is the only service in the platform that holds the JWT signing private key; every other service loads the corresponding public key at startup and validates tokens locally. The service is a handful of endpoints (Table 5.6) over a module of pure cryptographic primitives, but it is the single trust anchor for the entire user-facing surface.

Endpoint	Purpose
Login	Verify email and password; issue an access token and a refresh token
Refresh	Exchange a refresh token for a new access token (without rotating the refresh token)
Logout	Revoke a refresh token (idempotent)
Publish	Publish the public key, consumed by every other service at startup
Users	Create and manage users (admin-only)

Table 5.6: Auth endpoints.

The three auth-flow endpoints implement a conventional access-and-refresh-token pattern. On login, the client submits an email and password and the service compares the supplied password against the stored bcrypt hash. A missing user and a wrong password both return the same generic failure, so probing for valid email addresses reveals nothing about which half of the credential was wrong. On a successful match the service issues two artefacts: a short-lived JWT, and a longer-lived refresh token returned to the client once but persisted only as a digest. Refresh exchanges a valid, unrevoked, unexpired token belonging to a still-active user for a fresh access token, without rotating the refresh token itself. That non-rotation is a deliberate trade-off: rotating on every renewal would shorten the window in which a stolen refresh token can be reused, but it would also force the client to persist a new token on every access-token expiry, complicating client-side storage and recovery; for the MVP the simpler design is preferred over the marginal gain. Logout is idempotent and the user management endpoints are gated by the same admin check described in Subsection 5.5.1.

The choice of RS256 for token signing is the architectural decision with the largest downstream consequence. RS256 is an asymmetric algorithm: a private RSA key signs each token and the corresponding public key verifies it. Because the `auth` service is the only process in the platform that holds the private key, every other service receives only the public key and can therefore verify tokens but never mint them. The alternative (the symmetric HS256 variant) would also satisfy the JWT specification and would be simpler to deploy, with one shared secret instead of a keypair, but every service that verifies a token would then also hold the secret that signs it, so compromising any one of them would let an attacker forge tokens. With RS256 the blast radius is far smaller: breaching the `api-gateway` or `alerting` yields only verification capability, useful for replaying tokens an attacker already holds but not for forging new ones; only a compromise of the `auth` service itself, or theft of the private-key file, would permit forgery. Each token also carries a unique identifier so that individual tokens could be added to a revocation list in future without invalidating the entire issued set.

Two further measures protect the credential stores. Refresh tokens are generated from 256 bits of cryptographic entropy, well above the 128-bit threshold considered safe against brute force, and are never persisted in plain text: the raw token is handed to the client exactly once, in the login response, while only its irreversible SHA-256 digest is written to the database, and every later

lookup or revocation hashes the incoming token and queries by digest. The practical consequence is that a stolen database backup cannot be used to authenticate, because the digest column cannot be reversed and only the legitimate client still holds the raw value. Passwords, similarly, are hashed with bcrypt at a deliberate work factor and verified in constant time, so timing analysis cannot distinguish a near-miss from a far-miss. Bcrypt over a more modern function such as Argon2id is a defensible MVP trade-off (battle-tested, available in the Python standard ecosystem, and resistant to GPU-acceleration that broke earlier hash families), with a migration to Argon2id left as an open option.

The JSON Web Key Set endpoint is what stitches the trust model together for the rest of the platform. At startup the service publishes its RSA public key as a standard JWKS document, built once and served from memory, carrying a key identifier so that a client can tell which key signed a token if more than one is ever in rotation, a forward-looking detail, since the MVP only ever holds a single active key. The intended production pattern is for every service to fetch the key set once at startup, cache it, and verify locally for the process lifetime, refreshing only on a rotation signal. The MVP takes a simpler shortcut (services mount the same public-key file from disk, skipping the HTTP fetch), but both paths converge on the end state already exploited in Subsection 5.5.1: every service holds the public key in memory and never calls `auth` on the request path.

5.5.3 Alerting

The `alerting` service is the platform's rule-based fault detector. It consumes the same state-change Kafka topic that drives the WebSocket bridge, applies a small set of detection rules, and emits durable alert records to PostgreSQL plus email notifications over SMTP. Detection runs along two paths (Table 5.7): a reactive path that evaluates one rule on every Kafka event as it arrives, and a polling path that evaluates three rules on a fixed 60-second interval. Two of the four rules need TimescaleDB at evaluation time, since they query the meter value history written by the `telemetry-ingestion` service. The service opens that time-series pool optimistically at startup and, if the connection fails, simply skips those two rules on every poll while the rest continue against Redis and PostgreSQL, degrading gracefully rather than refusing to start.

Rule	Path	Triggers when	Auto-resolves when
Charger faulted	Reactive	A connector enters the faulted state	The connector leaves the faulted state
Charger offline	Polling (60s)	A station's last heartbeat is older than twice the heartbeat interval	The heartbeat is fresh again
Transaction stuck	Polling (60s)	An active transaction has received no meter readings for a configured interval	A fresh meter reading arrives
Grid capacity	Polling (60s)	Aggregated charger power at a site exceeds a configured warning threshold	Aggregate power drops back below the threshold

Table 5.7: Alerting service detection rules.

Every rule, whether reactive or polling, conforms to a single contract: it reads from whatever it needs (the incoming event, Redis, or TimescaleDB) and returns a description of what should fire and what should resolve, with no side effects of its own. A fire specification names the alert, its severity, and the entity it concerns, together with a context record capturing the evidence: the connector's error code, the age of a missing heartbeat, the threshold that was breached. The rule itself never writes back: every side effect is owned by the engine that invokes it. This is the same separation of pure logic from orchestration the `twin-state-engine` applies to its reducer (Subsection 5.4.2), for the same reason: rules become trivially unit-testable, and the engine's apply logic can be exercised independently of the rules it drives.

The engine applies two distinct guards before an alert reaches its destinations, and they solve deliberately different problems. The first is database-level deduplication: before inserting a row, the engine checks whether a non-resolved alert exists for the same tenant, type, and entity, and if so silently drops the incoming one; without this, a polling rule firing on every tick would write a fresh alert record every interval for the same persistent condition, polluting both the alerts table and the operator dashboard. The second is notification-level suppression: even when the row is new and inserted, the engine consults a suppression window before sending the email, and if the window is active the email is skipped while the row is still written. Separating the two means every detection remains auditable in the database even when its notification has been throttled: record duplication and email duplication are independent concerns.

Suppression is a thin wrapper over Redis: a key per tenant-type-entity triple, created with a time-to-live equal to the configured suppression window when the first notification is sent, tested for existence before each email, and deleted on auto-resolve so that a genuine recurrence of the condition sends a fresh notification rather than being absorbed by a stale window. Redis is chosen over an in-process dictionary for two reasons: the suppression state survives restarts and horizontal scaling, and Redis expires keys atomically, removing the need for any cleanup task. This is the

same Redis-as-shared-state pattern the `twin-state-engine` uses for live entity state (Subsection 5.4.2) and the `ocpp-adapter` uses for its connection registry (Subsection 5.4.1): a single coordination substrate every service treats consistently.

Three components close the pipeline. A PostgreSQL store owns the alert lifecycle (firing, acknowledged, resolved) and the same *"is there an active alert for this entity?"* query serves both the dedup guard on fire and the lookup on resolve, so the two share one definition of what active means. An email notifier renders templated messages for the configured operator inbox; when SMTP is not configured it is a silent no-op, which still leaves the alert record intact, so detection is preserved even in environments with no mail server. Finally, the service exposes a small REST surface, listing active alerts and acknowledging a firing one, which the gateway's alerts router proxies (Subsection 5.5.1), so the frontend sees a single authenticated origin even though the alert data is owned by a separate service. Resolution is not a manual REST action: an alert clears automatically when its triggering condition lifts, via the resolve path the rules emit.

The event-driven design keeps this layer open: adding a further consumer, a billing service, for example, would need only its own Kafka consumer group and the same public-key contract, with no change to the upstream state engine.

5.6 Live Twin User Interface

The live twin UI is the visual surface operators use to observe and interact with the charging infrastructure in real time. It is a single-page React application built with Next.js, TypeScript, and Tailwind CSS, served from its own container alongside the back-end services. The frontend holds no business logic of its own: every state-change decision is made upstream by the `twin-state-engine` and the `alerting` service and its job is to merge the REST snapshot returned by the `api-gateway` with the live updates arriving over the WebSocket, presenting the result as an interactive site map, transactions ledger, and alerts feed. The three subsections that follow cover its structure and authentication flow, its API client and state model, and the WebSocket lifecycle that drives the live view.

5.6.1 Application Structure and Routing

The application uses the Next.js App Router, organising routes into two groups that share layouts without adding path segments. The public group holds the login page; the authenticated group holds the twin view, the transactions list and detail pages, and the alerts feed, all wrapped in a common dashboard shell that renders the sidebar, header, persistent WebSocket connection, and a slide-in alert panel (Subsection 5.6.3). Route protection runs as a middleware before any page renders: because the tokens live in JavaScript memory rather than in cookies, the middleware cannot inspect the JWT itself, so it gates on a separate valueless session cookie written at login and redirects unauthenticated requests to the login page.

Token storage is the centre of the authentication design. The access and refresh tokens are held in memory and never written to a cookie, which removes them from the cross-site request forgery

attack surface entirely; the session cookie carries only a valueless marker, so even if it leaked through a misconfigured log or a shared device it would convey no credential. The accepted trade-off is that a hard browser reload wipes the in-memory tokens and pushes the user back through login, which is acceptable for the MVP and the safer choice for a single-page application than persisting tokens to local storage, which would trade the cross-site request forgery risk for cross-site scripting one.

5.6.2 API Client and State Model

The API client is generated from the `api-gateway`'s OpenAPI specification rather than hand-written, so that specification acts as a typed contract between the two sides of the wire: the frontend cannot call an endpoint that does not exist, and a backend signature change, such as a renamed response field or a newly required parameter, surfaces as a compile-time type error at the next build rather than as a runtime failure in the browser. A thin wrapper injects the in-memory access token into every request and refreshes it transparently when it expires, so an expired token never surfaces to the user as a disruption.

Client-side state is intentionally split between two libraries. TanStack Query owns the REST cache (the configured sites, the station and connector topology, and the transaction and alert lists), while Zustand owns the live state, keyed by entity so that a status change on one connector re-renders only that node rather than the whole grid. The split mirrors the data: Query owns *what should exist*, the persistent topology and historical lookups, while Zustand owns *what it looks like right now*. The two initialise each other cleanly: on first load the site map fetches every station and connector through TanStack Query, and because those responses already carry the live field merged in by the `api-gateway` (Subsection 5.5.1), a single round-trip seeds the live store, after which the WebSocket takes over. Returning to the twin view merges rather than overwrites, so live updates that arrived while the user was on another page are never discarded; the REST snapshot is authoritative only until the first live event is seen, and the WebSocket thereafter.

5.6.3 WebSocket Subscription and the Twin View

The WebSocket lifecycle is encapsulated in a single React hook, mounted once in the dashboard shell rather than per page. The consequence is that navigating between the twin, transactions, and alerts pages never tears down the connection: state changes that arrive while the user is away from the twin view are still applied, so the connector colours are already correct the moment the user returns. The hook opens one socket per browser tab, subscribes to the selected site (falling back to the whole tenant before a site is chosen), and dispatches each incoming message by type, applying state-change events to live store, consuming keepalives, and ignoring unknown types so that a new backend event can never break a deployed client. On disconnect it reconnects on a bounded exponential back-off, and two header indicators (a *'Reconnecting'* pill and a *'Live updates paused'* banner) ensure the operator is never left wondering whether what they see is current.

The twin view is a responsive grid of station cards, each showing a row of small circles, one per connector, coloured by the connector's current OCPP status: green for available, blue for charging, red for faulted, and so on. The colour mapping reuses the same status enum the `twin-state-engine` emits (Subsection 5.4.2), so a connector is coloured straight from its status code with no decoding layer in between; charging connectors carry a soft pulse animation and a corner dot marks with an active transaction, and clicking one opens a detail panel with live readings and a link to the session timeline. A relative-time staleness indicator in the header refreshes on every live message, so if the stream goes silent the value visibly drifts upward, answering the operator's implicit question, *is what I am looking at current?*, without a refresh or a guess.

Figures 5.2 and 5.3 show the result: the live twin site map for a development site, with connectors in a mix of live states, and the transaction-detail view, which renders a session's power, energy, and SoC series alongside the raw meter value table. The observed constant power is characteristic of the synthetic generator (Section 5.7).

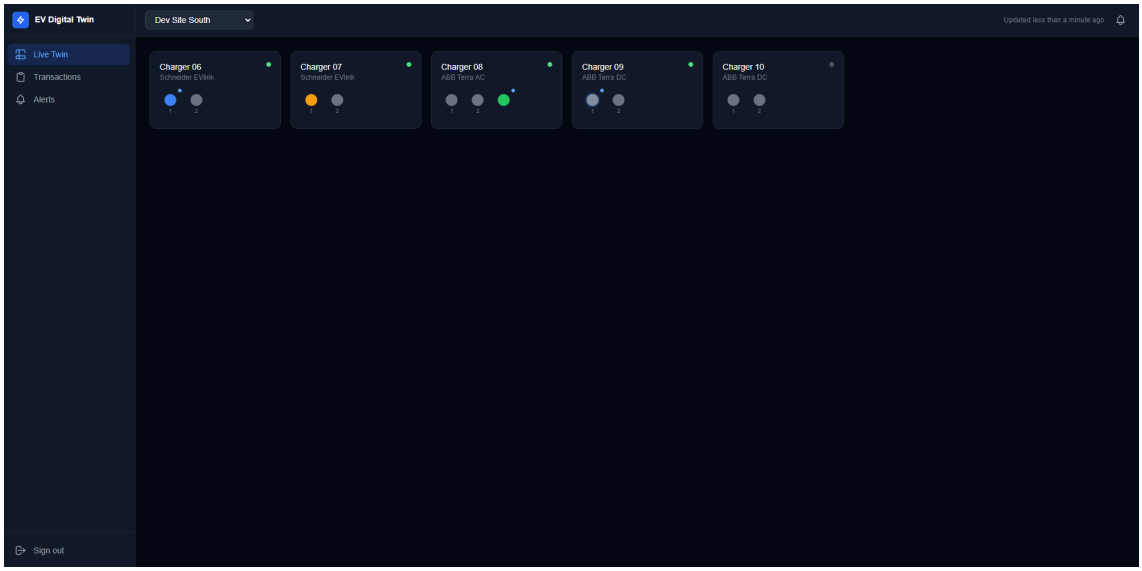

Figure 5.2: The live twin site map.

5.7 Intelligence Layer Proof-of-Concept

The Intelligence layer (Section 4.7) is the platform's path from reactive monitoring (the rule-based `alerting` service), towards proactive, learned fault detection. The MVP delivers this layer only as a PoC: a self-contained training pipeline that consumes telemetry from TimescaleDB, learns to classify charging session anomalies, and writes its results to disk for offline inspection. There is no production inference path: no subscription to live state changes, no scoring of in-progress sessions, and no model-serving endpoint behind the `api-gateway`. What the PoC does demonstrate is that the platform exposes a queryable telemetry surface an ML workflow can target; what it explicitly does not demonstrate is that the resulting model would generalise to real-world charging faults.

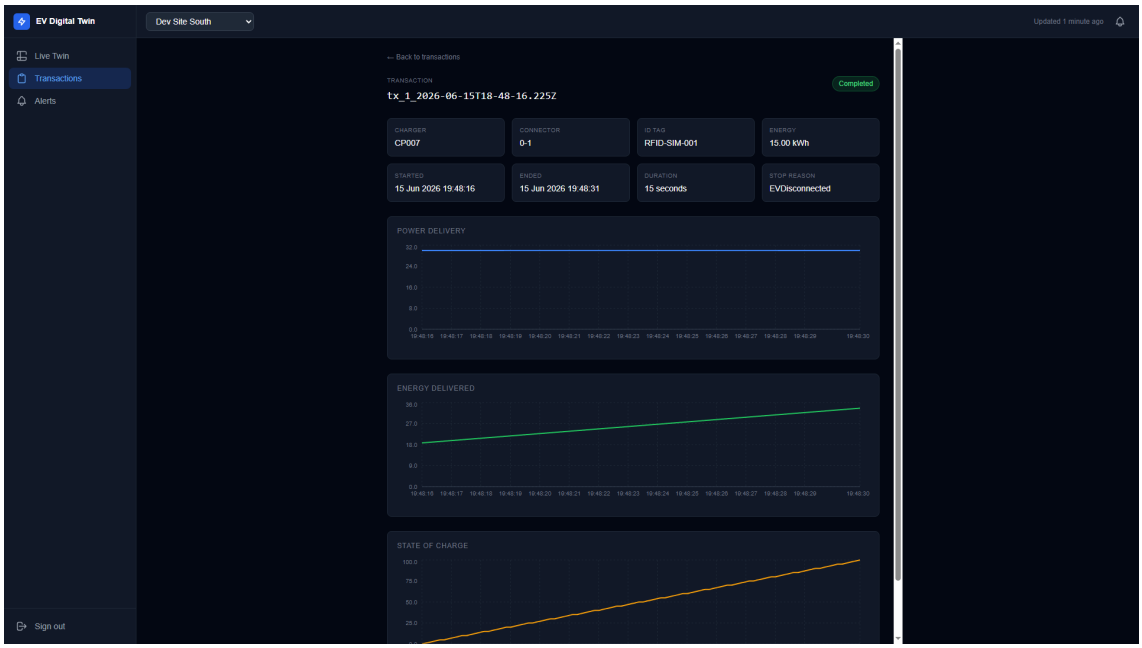

Figure 5.3: The transaction-detail view for a simulated charging session.

No real labelled fault data exists for the MVP: no real charger is connected, and even production deployments rarely produce ground-truth labels at the session level. The PoC therefore uses synthetic sessions, each driven through a mock OCPP charger that runs the full session lifecycle and writes labelled metadata alongside the telemetry. The decisive point is that these synthetic sessions pass through the same `ocpp-adapter`, `twin-state-engine`, and `telemetry-ingestion` services as a real session would (Section 5.4), so the time-series rows they leave behind are structurally indistinguishable from real telemetry. Five session classes are generated: a normal baseline and four fault types, each an injected perturbation of the power, energy, and SoC signals (Table 5.8).

Label	**Meaning**	**Injected pattern**
Normal	Baseline charging session	Power and energy track linearly
Power drop	Inverter fault	Power drops to ≈50% through the session
Power flicker	Loose connection	One or two samples drop to ≈20% at random points
Meter stuck	Meter fault	Energy register stops accumulating while power continues to flow
SoC stagnation	BMS communication fault	SoC stops progressing while power flows

Table 5.8: Anomaly classes.

The pipeline reduces each session's variable-length telemetry to a single fixed-length vector of 37 numeric features, so that sessions of differing length become directly comparable to a clas-

sifier. The feature design is deliberate rather than exhaustive: for each measurand it captures the distribution (central tendency, spread, extremes, and sample count); it splits each session in half and records each half's mean and their ratio, targeting mid-session regime changes such as power drop; it measures how monotonically each signal moves, which captures the stuck and stagnation faults directly; and a cross-measurand feature flags inconsistency between accumulated energy and integrated power. Sessions missing essential power features are dropped and the few remaining gaps are imputed, so the model always receives a complete vector.

Three models are trained on the same stratified train/test split so their results are directly comparable: an unsupervised baseline (an Isolation Forest trained only on normal sessions, representing the strongest fault detector available without labels) and two supervised gradient-boosted-tree classifiers, one discriminating anomaly from normal and the other predicting which of the five classes applies. The pipeline runs reproducibly from a single command with a fixed seed, and its outputs are analysed in Section 6.4, where the contrast between the unsupervised baseline and the supervised classifiers (and why the latter's near-perfect scores reflect the construction of the synthetic data rather than production-grade detection) is reported in Table 6.2.

Whether the learned models would generalise to real charging faults is the separate question examined in Section 6.4. Three concrete extensions towards production detection remain out of scope and are recorded in Chapter 7: an online inference service that scores in-progress sessions and emits predicted-fault alerts; a real-world labelling loop in which operator acknowledgements feed back as training labels; and drift monitoring to detect when the live data distribution diverges from the training distribution.

5.8 Summary

This chapter has documented the implementation of the DT MVP. Section 5.1 delimited its scope against the target architecture, and Sections 5.2 and 5.3 set out the development methodology and the infrastructure stack. Section 5.4 detailed the ingestion path: the `ocpp-adapter` normalises OCPP 1.6 and 2.0.1 messages into canonical events, the `twin-state-engine` reduces those events to typed per-entity state, and the `telemetry-ingestion` service writes the raw time-series to TimescaleDB. Section 5.5 covered the user-facing layer: the `api-gateway` merges live Redis state with PostgreSQL topology over REST and fans state changes over WebSocket; the `auth` service is the single trust anchor for JWT issuance; and the `alerting` service applies four rule-based detectors with database-backed deduplication and Redis-backed suppression. Section 5.6 described the Next.js frontend, whose hybrid model separates *what should exist* from *what it looks like right now*. Finally, Section 5.7 delivered an offline ML PoC that validates the platform's telemetry surface as an ML data source without overclaiming production-grade fault detection. The empirical evaluation of this implementation follows in Chapter 6.

Chapter 6

Evaluation and Results

This chapter evaluates the extent to which the implemented MVP meets the objectives set out in Section 1.3 and the proof obligations established in Section 5.1. The evaluation combines a controlled synthetic load test with a corpus of real production data, and is presented in five parts: the methodology and evaluation criteria (Section 6.1); the platform's performance and scalability under controlled load (Section 6.2); its validation against real-world OCPP 2.0.1 field data (Section 6.3); the evaluation of the intelligence layer PoC (Section 6.4); and the threats to validity that bound these results (Section 6.5). Throughout, results are reported together with the conditions under which they were obtained, so that each claim is reproducible and interpreted within its stated limitations.

6.1 Evaluation Methodology

The evaluation is operationalised through four concrete criteria. Functional correctness (that OCPP messages are normalised, reduced to per-entity state, and persisted across Redis, PostgreSQL, and TimescaleDB without loss) is demonstrated cumulatively by the implementation of Chapter 5 and, on real traffic, by the field validation of Section 6.3. Performance is assessed against the end-to-end latency budget defined in Subsection 4.6.2, measured at the MVP's target scale of fifty concurrent chargers. Real-world applicability is assessed by whether the pipeline handles authentic OCPP 2.0.1 field data, directly addressing the gap identified in Chapter 3, where the existing DTs are shown to be validated almost exclusively in simulation. Intelligence layer feasibility is assessed by whether per-session telemetry can be reduced to features a learned model classifies, established as a PoC rather than a production claim (Sections 5.7, 6.4).

The evaluation draws on two complementary sources of evidence, because neither alone is sufficient. The first is a controlled synthetic experiment: a configurable fleet of simulated chargers driven by a load-test harness under reproducible conditions, permitting precise control of scale, arrival rate, and duration, and yielding a known ground truth against which correctness can be checked; its weakness is realism, since synthetic traffic is more regular and well-formed than real charger traffic. The second is real-world field data: nine months of genuine charging sessions

from a production CSMS, which carry the irregularity, gaps, and protocol detail of authentic OCPP 2.0.1 traffic but offer no controlled load and no fault labels. Used together, the synthetic stream substantiates the performance and scalability claims under reproducible control while the field stream substantiates real-world validity, each covering the other's blind spot, a separation reflected directly in the structure of Sections 6.2 and 6.3.

All measurements were obtained from the MVP stack of Section 5.3, deployed as a single-host Docker Compose stack comprising the six back-end services and their backing infrastructure (Kafka, PostgreSQL, TimescaleDB, and Redis) on an AMD Ryzen 5 4500U (6 cores, 16GB RAM, Windows 10). Running the entire platform on one host is representative of a development and pilot deployment rather than a production cluster; the consequence, that all services, brokers, and databases contend for the same resources, is a recognised threat to validity, discussed in Section 6.5. Configuration was held fixed across runs, and stochastic components were seeded for reproducibility, the ML pipeline using a fixed seed for its train/test split and model initialisation.

The metrics are drawn from the platform's own instrumentation rather than external probes. For the performance evaluation, the load-test harness records end-to-end latency (the interval between a charger emitting an event and the corresponding state change reaching a subscribed client) as a percentile distribution, alongside per-message ingestion latency; Kafka consumer-group lag is sampled from the brokers to confirm the consumers keep pace with ingestion. For the field validation, fidelity is established by comparing row counts in TimescaleDB against the source corpus and transaction counts in PostgreSQL against the number of replayed sessions. For the intelligence layer evaluation, standard supervised-classification metrics, such as accuracy, per-class precision, recall and F1, and ROC-AUC, are computed on a held-out test set (Section 6.4).

6.2 Performance against the Latency Budget

The platform's performance was measured with a controlled load test exercising the complete pipeline under synthetic but realistic conditions. A fleet of fifty CSs, the MVP's target evaluation scale, was simulated for 30 minutes against the containerised stack, each charger connecting over OCPP, completing the boot and authorisation handshake, opening a transaction, and emitting periodic meter values at a 30-second cadence. The harness records two latencies directly: the per-message ingestion latency of meter values, and the end-to-end latency from a charger emitting an event to the resulting state change being delivered to a subscribed WebSocket client, the figure that matters for a live DT. Kafka consumer-group lag was sampled throughout to confirm the downstream consumers kept pace with ingestion.

The runs completed without any failures: all fifty chargers booted, all fifty transactions started and completed, and no connection errors occurred over the 30-minute windows. In each run the engine processed the fleet's full set of lifecycle state changes (the boot, status, and transaction-lifecycle transitions of the fifty chargers) and delivered every one to the subscribed client, establishing that the pipeline sustains the full target fleet end to end without dropped connections or sessions.

Table 6.1 reports the measured latencies, aggregated as the mean and standard deviation across the five runs. The end-to-end state-propagation latency had a median of 438 ms (±85) and a 95th percentile of 810 ms (±92); the maximum observed across all runs was 954 ms, and every run's distribution remained below one second at all percentiles. Figure 6.1 overlays the per-run cumulative distributions, which rise steeply through the 300-500 ms band and sit to the left of the one-second line.

Two points qualify this against the latency budget of Subsection 4.6.2. First, that budget is an estimate of per-stage compute cost under ideal, unloaded conditions; the figure measured here is realised latency under concurrent load on a single host, which additionally bears broker transport, consumer scheduling, and contention among co-located services that the estimate excludes. The realised median sits well above the 28-86 ms best-case, as expected, and reducing it is identified as future work in Section 7.3. Second, the measurement ends at WebSocket delivery: the budget's client-side render stage (16-50 ms) is not instrumented by the load-test client, but adding it to the maximum observed delivery latency still leaves the full path within one second. The sub-second threshold, the architecturally meaningful claim, distinguishing an active twin from a passive one, thus holds end to end at every percentile.

Meter values ingestion latency was an order of magnitude lower, with a median of 18 ms and a 99th percentile of 76 ms. This figure is the adapter's acknowledgement round-trip: because the adapter acknowledges each meter value immediately, before normalising and forwarding it (Subsection 5.4.1), it excludes the downstream propagation that the full event-to-client latency incurs.

Latency	**p50**	**p95**	**p99**	**max**
End-to-end (event → client delivery), ms	438 ± 85	810 ± 92	856 ± 75	862 ± 73
Meter values ingestion, ms	18 ± 3	41 ± 9	76 ± 27	121 ± 31

Table 6.1: Measured latency distributions, mean ± standard deviation across five fifty-charger 30-minute runs.

Consumer-group lag is the decisive indicator of whether the platform keeps pace with its input. Across the five runs, total lag held at zero for the overwhelming majority of samples, rising only to a brief transient of at most fifty messages, one per charger, when the fleet emits its meter values synchronously every 30 seconds; each such backlog drained before the following sample. At no point did the `twin-state-engine` or `telemetry-ingestion` consumers fall behind the rate at which the adapter published events. The pipeline therefore operated in steady state at the target load, and the sub-second propagation latencies are sustained figures, not a transient measured before saturation.

Resource utilisation was sampled alongside latency. The platform's own services held essentially flat memory in a representative run, each growing by under 7 MiB and peaked below 90 MiB, with the backing databases below 50 MiB; none met the leak criterion of growing by more than

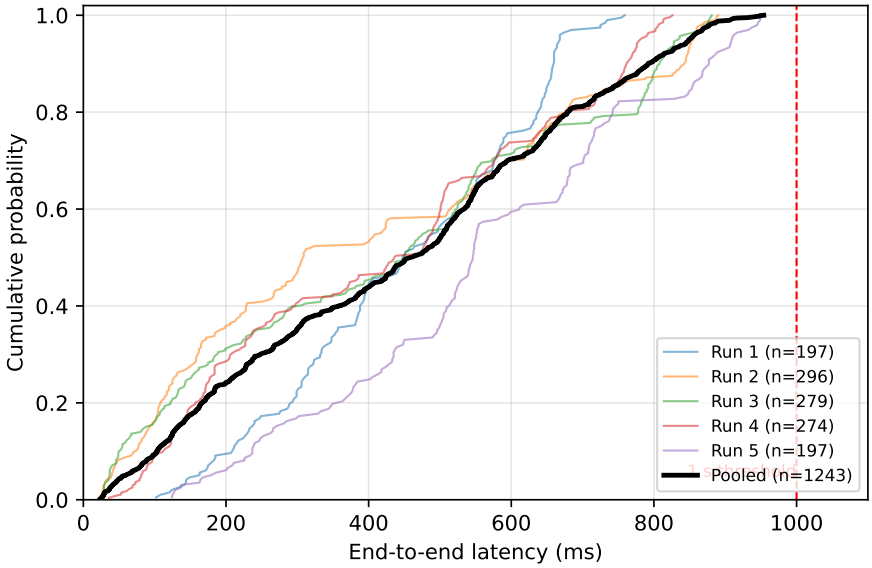

Figure 6.1: Per-run cumulative distribution of end-to-end synchronisation latency under fifty-charger load, across five runs.

50 MiB and more than half its initial footprint. The only component to grow appreciably was the Kafka broker (+12%, from 665 to 744 MiB), behaviour consistent with normal log-segment and buffer allocation under sustained traffic rather than a leak. CPU stayed modest: the heaviest application service, the state engine, peaked at 54%, and the brief Kafka bursts (a 173% per-core peak, roughly 1.7 cores) coincided with log compaction.

These results confirm the MVP meets its performance proof obligation: a fifty-charger fleet is ingested and reflected in the live twin within the sub-second threshold, with zero message loss and no sustained consumer lag, reproduced across five independent 30-minute runs. Two boundaries qualify this claim: the single-host deployment and the characterisation at a single scale rather than a load sweep are examined in Section 6.5.

6.3 Field Validation with Real-World Data

Beyond the synthetic load test of Section 6.2, the platform was validated against real production data. As established in Chapter 3, a recurring weakness of existing charging infrastructure DTs is that they are evaluated exclusively in simulation or on historical datasets in isolation, with little evidence of real-world applicability. To address this directly, a corpus of genuine charging sessions was obtained from the production CSMS operated at INESC TEC and replayed through the DT pipeline. This validates a dimension the synthetic load test cannot: that the ingestion path of Section 5.4 correctly handles authentic OCPP 2.0.1 traffic (the protocol version the synthetic simulator, which speaks OCPP 1.6, never exercised) end to end and without modification.

The dataset comprises 1,431 real charging sessions recorded across 22 distinct chargers over a nine-month period (5 September 2025 to 1 June 2026). The sessions are substantial and varied:

a median duration of 3.9 hours (inter-quartile range 2.2-6.5 h) and a median delivered energy of 10.4 kWh, with the longest session spanning several days; in total the corpus contains 2,712,672 meter value samples. A characteristic of this particular fleet is that every charger reports only four electrical measurands (voltage, current, active power, and frequency) sampled together at each interval. It reports neither the OCPP energy register, which the CSMS instead derives by integrating instantaneous power, nor SoC, which is unavailable; this limitation becomes significant for the ML evaluation in Section 6.4.

The data was extracted read-only through the CSMS's REST export endpoint, which already groups each session's meter values into the per-timestamp sampled-value structure of the OCPP specification. Because the CSMS stores the unit of measure rather than the measurand name, the measurand was inferred deterministically from the unit, since each sample group carries exactly one value per unit. Each session was then reconstructed into its original OCPP 2.0.1 message sequence and submitted to the `ocpp-adapter`'s webhook endpoint. This is the architecturally intended integration for an existing CSMS feeding the DT (Subsection 5.4.1): webhook mode exists precisely so that a third-party CSMS can post raw OCPP frames, which are normalised and published to Kafka exactly as if they arrived over a direct connection. The replayed frames therefore traverse the full production path: normaliser, Kafka, and both the `twin-state-engine` and the `telemetry-ingestion` consumers, with no code specific to the replay.

Ingestion fidelity was first verified on a clean database with a twenty-session subset, which produced 40,332 telemetry rows and twenty corresponding relational transaction records, confirming zero message loss. The complete corpus was then replayed: all 1,431 sessions were accepted (37,462 reconstructed frames, zero failures) and reduced by the state engine to 1,431 relational transaction records spanning all 22 chargers, with live connector and station state maintained in Redis and the full electrical telemetry written to TimescaleDB.

The real sessions are visible in the live twin interface under a dedicated field site (Figure 6.2), indistinguishable in presentation from synthetically generated ones. Their data, however, is visibly authentic: the detail view of one real session (Figure 6.3) shows an active power trace varying continuously across the transaction duration, in clear contrast to the constant synthetic profile of Figure 5.3, with the energy and SoC panels left empty, a direct visual confirmation of the fleet's measurand limitation.

The replay surfaced one integration requirement worth recording. The state engine writes a relational transaction record only when the corresponding station and connector already exist in the operational database, enforced by a foreign-key constraint, whereas the telemetry path has no such dependency. When the real chargers were replayed before being registered in the twin's topology, their telemetry was ingested while the relational transaction upserts were skipped and logged, rather than raised as errors; provisioning the 22 chargers and replaying again resolved this. The asymmetry is by design (the time-series path is deliberately tolerant of unknown devices so that no telemetry is ever lost, while the relational path is strict to preserve referential integrity), and the practical lesson is that a charger must be registered in the twin's topology before its sessions can be reduced to relational state.

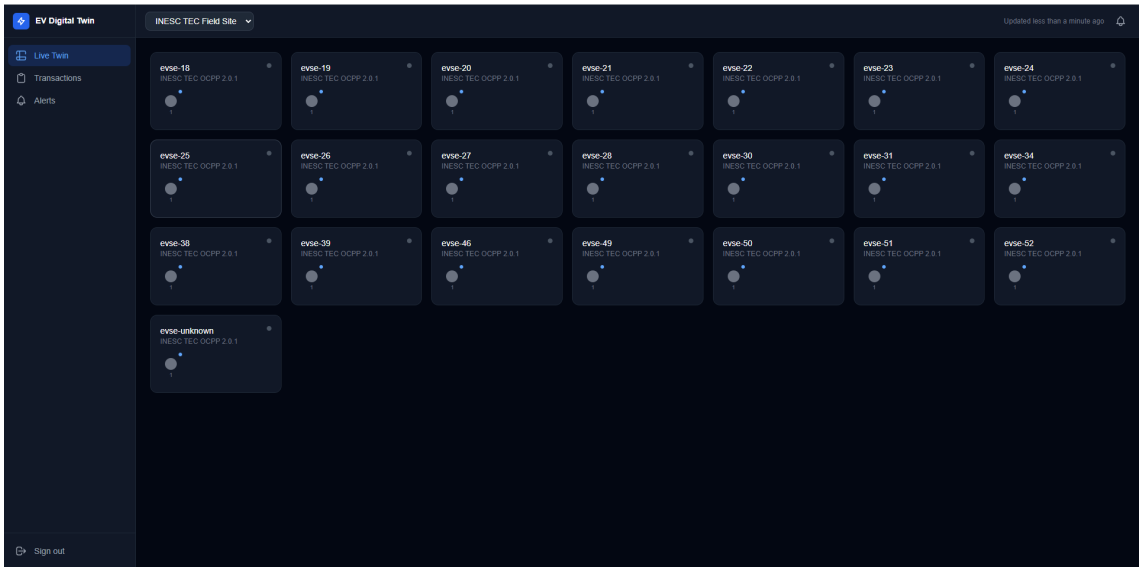

Figure 6.2: The production fleet represented in the Live Twin under a dedicated field site.

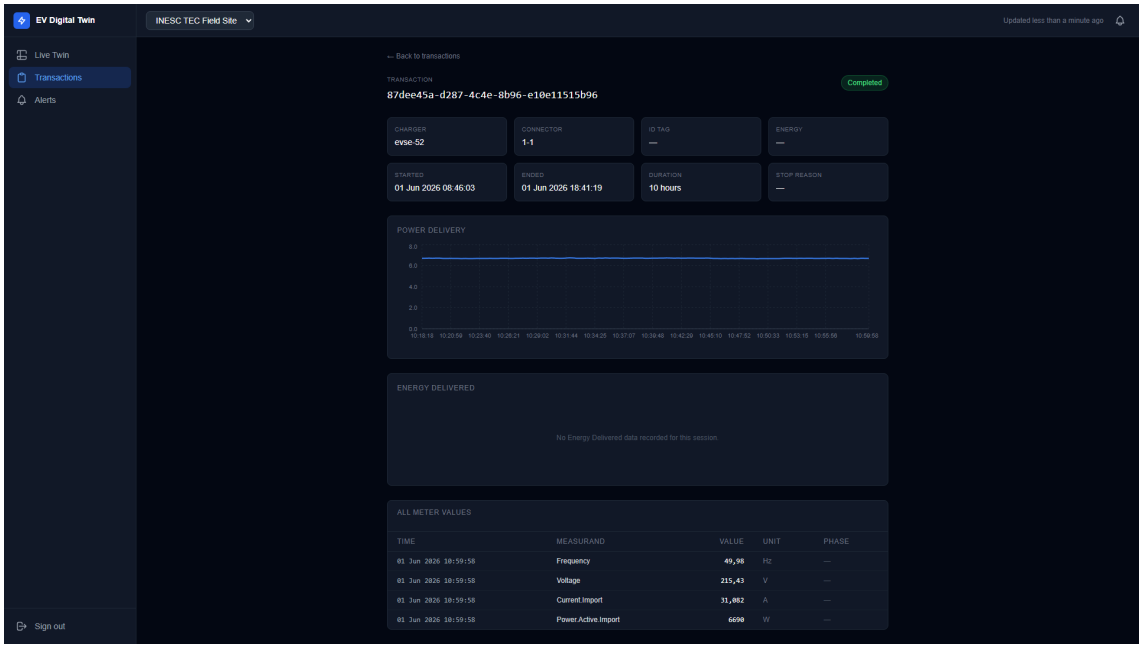

Figure 6.3: Transaction-detail view for a real field session.

Two limitations are specific to this exercise. First, the replay establishes functional fidelity rather than performance (frames were submitted as a burst rather than at a controlled arrival rate) so it complements but does not replace the load test of Section 6.2. Second, of the 1,431 sessions, 35 contained no meter values, opening and closing without reporting telemetry, and so produced a transaction record but no time-series row, a genuine characteristic of field data rather than a pipeline fault. The fleet's restriction to four measurands, noted above, additionally constrains the synthetic-versus-real comparison of Section 6.4, while the broader threats to validity are consolidated in Section 6.5.

6.4 Intelligence Layer Proof-of-Concept Evaluation

This section evaluates the intelligence layer PoC introduced in Subsection 5.7, which tests whether the platform's telemetry can serve as the foundation for learned anomaly detection. The evaluation uses 500 synthetic charging sessions generated through the full ingestion pipeline, each labelled with one of five classes (normal operation and four fault types: power drop, power flicker, stuck meter, and SoC stagnation) and reduced to 37 per-session features summarising the statistical and temporal behaviour of each measurand. Three models were trained on an identical stratified 70/30 train-test split with a fixed random seed: an Isolation Forest as an unsupervised baseline trained only on normal sessions, an XGBoost binary classifier (anomaly versus normal), and an XGBoost multiclass classifier across all five labels. All metrics are reported on the held-out test set.

Table 6.2 summarises the results. The unsupervised Isolation Forest performs at essentially chance level, with a ROC-AUC of 0.50 and a macro-F1 of 0.17: it cannot separate anomalies from normal sessions without labels. The two supervised XGBoost models, by contrast, achieve perfect scores: F1 of 1.00 on both the binary and the five-class problems, with every test session assigned to its correct class and no off-diagonal errors. The gap between the two approaches (a detector no better than random chance against one that separates every class perfectly) is clear; and, as the next paragraph argues, it is this contrast, not the perfect score itself, that is the informative result.

Model	**Accuracy**	**Macro-F1**	**ROC-AUC**
Isolation Forest (Unsupervised)	0.20	0.17	0.50
XGBoost - Binary	1.00	1.00	1.00
XGBoost - Multiclass	1.00	1.00	1.00

Table 6.2: Anomaly-detection model performance on the synthetic test set with 150 sessions.

A macro-F1 of 1.00 must be read with caution: it reflects the nature of the synthetic data, not a production-ready detector. The four fault types are injected by the data generator (Section 5.7) as deterministic perturbations of the power, energy, and SoC signals, and the engineered features were designed to summarise exactly those signals. The supervised model's own feature-importance ranking (Figure 6.4) confirms this directly: the multiclass classifier concentrates the

bulk of its decision gain on a handful of features that each encode one of the injected perturbations: the second-half mean power, which collapses under a power drop; the fraction of increasing energy steps and the energy-to-power ratio, which both register a stuck meter where the energy register stops climbing while power still flows; and the mean SoC, which sits lower when SoC stagnates, with the brief power-flicker dips picked up by the minimum and mean power. Because each fault carries a near-deterministic signature in the feature matrix, a gradient-boosted tree learns axis-aligned decision boundaries that separate the classes trivially. The contrast with the Isolation Forest is the informative result: even when the classes are perfectly separable to a supervised model, an unsupervised detector with no knowledge of which feature axes matter performs no better than chance. The practical conclusion is that supervised labelling, not algorithmic sophistication, is the prerequisite for fault detection on this data, and what the perfect scores genuinely validate is the integration: synthetic OCPP traffic flows through the entire pipeline, lands in TimescaleDB, and is recoverable as a clean, labelled, model-ready feature matrix.

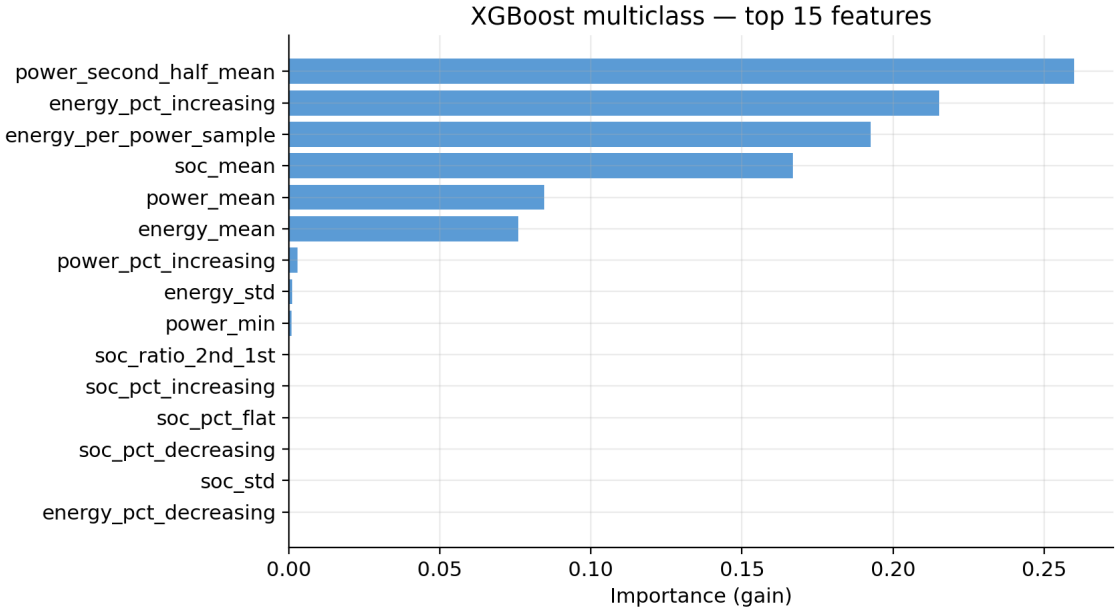

Figure 6.4: Feature importance for the XGBoost multiclass classifier.

To assess how representative the synthetic data is of real charging behaviour, the synthetic normal sessions were compared against the real field corpus of Section 6.3 on the single measurand the two datasets share, active power. Table 6.3 reports the nine per-session power features for 100 synthetic normal and 1,396 real sessions. The contrast is clear: the synthetic normal sessions are a perfectly constant 6 kW signal (every per-session statistic has zero variance, the power is flat across every step, and each session contains exactly 30 samples), while real sessions vary continuously, are flat across a median of only 5.5% of steps, span nearly the full range within a session, and run far longer. Figure 6.5 makes the divergence concrete, and a two-sample Kolmogorov-Smirnov test separates every feature, with a statistic between 0.52 and 0.98.

Feature	Synthetic normal	Real (median)	KS
Mean power (W)	6000	4374	0.60
Std. power (W)	0	636	0.97
Min power (W)	6000	4	0.98
Max power (W)	6000	6741	0.52
Power range (W)	0	5430	0.97
2nd/1st-half power ratio	1.00	0.99	0.62
Flat-step fraction	1.00	0.055	0.98
Increasing-step fraction	0.00	0.49	0.97
Samples per session	30	431	0.94

Table 6.3: Per-session active power feature comparison across all nine engineered features.

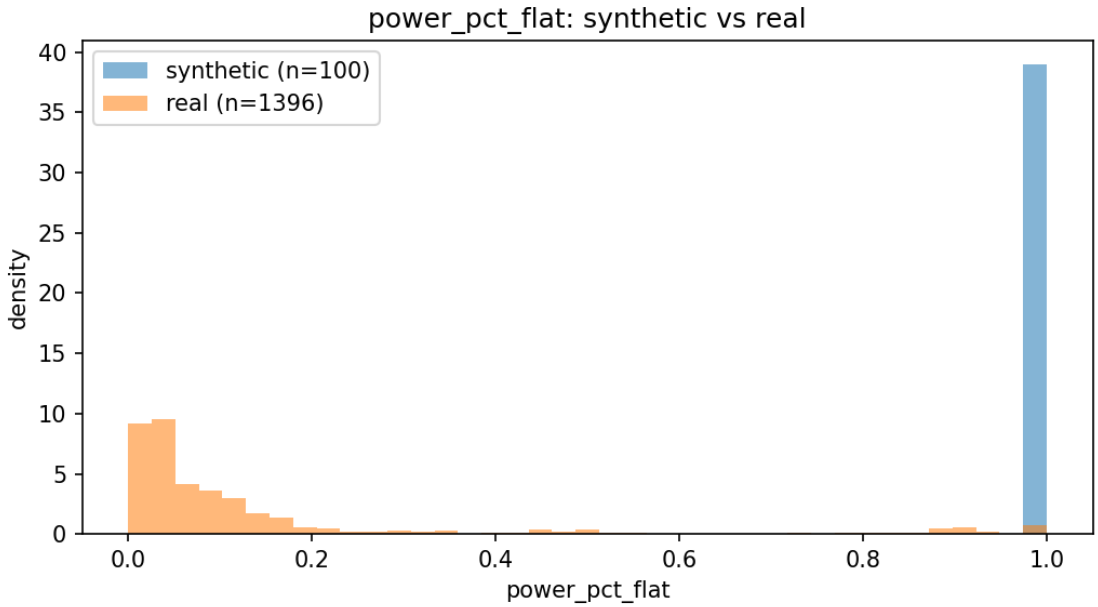

Figure 6.5: Distribution of per-session flat-step fraction of active power.

Read together, the two analyses tell a coherent and honest story. The synthetic normal baseline is a constant line, so any injected deviation is trivially separable, which is precisely why the supervised classifiers reach F1 = 1.00. The simulator reproduces the right power magnitude but not the temporal variability of real charging, where power fluctuates throughout a multi-hour session. The PoC therefore establishes what it set out to: the platform is a viable end-to-end data source for a learned intelligence layer, with telemetry captured, persisted, and reduced to model-ready features on which supervised models train and evaluate successfully. It does not, and was not intended to, demonstrate a detector that would generalise to real chargers.

Closing the gap to a deployable detector requires two things the MVP does not have. First, labelled real data: the field corpus carries no fault annotations, so it supports only the descriptive

comparison above and not supervised training or evaluation; a production path would need an operator-in-the-loop labelling process, for instance feeding acknowledged alerts (Subsection 5.5.3) back as labels. Second, a richer and more realistic generator, or direct training on real telemetry: the field data also exposes voltage, current, and frequency signals that the synthetic generator never produced, and which a real-world detector could exploit. Both are identified as future work in Chapter 7.

6.5 Threats to Validity

The results of this chapter must be read against the limitations of how they were obtained. This section sets out the principal threats to validity, grouped into construct validity (whether the evaluation measures what it claims to), internal validity (whether the observed effects are attributable to the system rather than to artefacts of the setup), and external validity (how far the findings generalise beyond the conditions tested).

6.5.1 Construct Validity

The most significant threat concerns the ML evaluation. Because the four fault types are injected by the same generator whose perturbations the engineered features were designed to detect, the perfect F1 of the supervised classifiers (Section 6.4) reflects the separability of the synthetic data by construction, not a demonstrated ability to detect real faults, a distinction the synthetic-versus-real comparison of Section 6.4 establishes quantitatively. The ML results validate the integrity of the storage-and-feature pipeline, not the generalisability of any detector.

A second construct threat concerns latency. The 28-86 ms figure of Subsection 4.6.2 is an estimated per-stage compute budget, while the 438 ms median of Section 6.2 is the realised end-to-end latency under load; the two quantify different things, and conflating them would misrepresent the system's performance. The realised figure is also measured to WebSocket delivery, and so excludes the budget's client-side render stage, a further reason the two are not directly comparable, though adding the render keeps the full path within one second. The evaluation accordingly treats the sub-second threshold, not the compute budget, as the requirement.

6.5.2 Internal Validity

All measurements were taken from a single-host deployment in which the six back-end services and their backing infrastructure share one machine (Section 6.1); the latency and throughput figures therefore reflect resource contention among co-located components rather than the isolated performance of any one service, and a distributed deployment with dedicated resources could yield materially different numbers. The field validation establishes functional fidelity rather than performance, since frames were replayed as a burst rather than a controlled arrival rate (Section 6.3). Finally, all metrics derive from the platform's own instrumentation rather than independent external probes.

6.5.3 External Validity

Several factors bound how far these results generalise. The synthetic dataset does not represent real charging behaviour, as the active power comparison of Section 6.4 demonstrates quantitatively, so conclusions drawn from it do not transfer to production chargers. The field data, conversely, originates from a single CSMS and a single fleet of 22 chargers operated by one organisation, reporting only four electrical measurands (voltage, current, active power, and frequency); the overlap with the synthetic feature space is limited to active power, and deployments with richer or different telemetry may behave differently. Because the field corpus carries no fault labels, the ML model could not be trained or evaluated on real data, leaving its real-world accuracy untested. Performance was characterised at the MVP's target of fifty concurrent chargers, so behaviour at the platform's longer-term design scale of a thousand chargers, and under multi-host deployment, remains unmeasured. Finally, the field validation exercises the webhook ingestion path with reconstructed OCPP 2.0.1 frames rather than live WebSocket traffic: the architecturally intended integration for an existing CSMS, but a step removed from a charger connecting in real time.

Taken together, these threats define the boundary of the claims the evaluation supports. Within that boundary the results are robust: the platform ingests genuine OCPP 2.0.1 field data end to end without loss, meets its sub-second synchronisation requirement at the target scale, and provides a working data foundation for a learned intelligence layer. Beyond it (production-grade fault detection, performance at large scale, and generalisation across fleets and CSMS vendors), the claims are deliberately not made, and are identified as future work in Chapter 7.

Chapter 7

Conclusions and Future Work

This dissertation set out to design and build a DT for EVCI capable of moving beyond passive monitoring towards proactive, real-time operation, addressing the integration and real-world validation gaps identified in the literature review. This chapter draws the work together: Section 7.1 summarises the contributions, Section 7.2 sets out the limitations that bound them, and Section 7.3 identifies the work that remains to realise the full vision.

7.1 Summary of Contributions

The work makes five principal contributions, each responding to a gap established in Chapter 3.

A scalable, event-sourced DT architecture. The dissertation contributes a decoupled, event-driven architecture (Chapter 4) that separates protocol ingestion, state management, intelligence, and actuation, and is recoverable by construction through event sourcing. Justified through a set of ADRs, it constitutes the complete target design for a proactive DT; the components most critical to the platform's core are realised in this work, while the closed-loop intelligence and actuation layers are specified as future work. Where much of the surveyed literature treats the charger as an interface to be monitored, this architecture treats the infrastructure as a stateful, queryable, recoverable model.

A non-disruptive integration strategy with legacy infrastructure. Rather than replacing the operator's existing CSMS, which is a high-risk migration, the architecture positions the twin as an intelligence layer atop the CSMS as transport (Option C, Subsection 4.3.3). This addresses the integration fragmentation identified in Chapter 3, where the adoption of DT techniques is repeatedly hindered by the assumption that the twin must own the protocol endpoint. The strategy allows a twin to be deployed alongside production infrastructure with minimal operational risk.

A working MVP implementation. The architecture is realised as a working implementation (Chapter 5): six independently deployable back-end services, backed by a polyglot persistence layer, that ingest OCPP 1.6 and 2.0.1 traffic, normalise it into a single canonical representation,

reduce it to a real-time entity-state model through an event-sourcing engine across Redis and PostgreSQL, persist time-series telemetry to TimescaleDB, and drive a live, sub-second twin UI. The implementation meets its sub-second state-synchronisation requirement at the target of fifty concurrent chargers (Section 6.2).

Validation against real-world production data. The most distinctive contribution is empirical. Where the literature is dominated by systems validated exclusively in simulation, this work ingested nine months of genuine OCPP 2.0.1 traffic (1,431 charging sessions across 22 chargers from a production CSMS) through the complete pipeline, end to end and without loss (Section 6.3). This exercised the OCPP 2.0.1 path that the synthetic load test never touched, and demonstrated that the integration strategy works on authentic field data rather than idealised traffic. It also surfaced concrete engineering findings, such as the requirement that a charger must be provisioned in the twin's topology before its sessions can be reduced to relational state.

A clearly scoped PoC for the intelligence layer. Finally, the work establishes that the platform's telemetry is a viable foundation for learned fault detection (Sections 5.7, 6.4). The PoC is presented with deliberate rigour: the perfect classifier scores are shown to reflect the separability of synthetic data by construction rather than a deployable detector, and a quantitative comparison against the real field data exposes the gap between the simulator's behaviour and reality. The contribution is therefore not a fault detector, but a validated, reproducible path from raw OCPP telemetry to a model-ready feature matrix, together with a realistic account of what remains before such a model could be trusted in production.

Answering the research questions. Returning to the three questions posed in Section 1.3, the work answers each as follows.

RQ1 asked how a DT can achieve real-time, bidirectional synchronisation with physical EVSE without disrupting an existing CSMS. The non-disruptive integration strategy positions the CSMS as transport and the twin as intelligence (Option C, Subsection 4.3.3); this answers the inbound direction, and the field validation demonstrates it on real traffic: nine months of OCPP 2.0.1 sessions were ingested through the webhook path with no change to the CSMS's authentication or connection handling (Section 6.3). The outbound direction, by which the twin would issue commands back through the CSMS management API, is specified architecturally (Subsection 4.7.1) but deferred beyond the MVP. The question is answered in full for the read path, designed and demonstrated on real data, which realises the high-fidelity Digital Shadow; the design answers for the write path, whose completion would close the loop into a true bidirectional DT and is the foremost item of future work.

RQ2 asked what architectural patterns and polyglot persistence strategies are required to ingest and reduce high-frequency telemetry into a sub-second state representation while preserving a durable event history. The event-driven, event-sourced architecture answers this: event sourcing maintains a durable, replayable log from which a sub-millisecond Redis view is reconstructed,

while polyglot persistence routes live state, durable history, and time-series telemetry to engines matched to each access pattern. The evaluation confirms the strategy meets its target, synchronising a fifty-charger fleet within the sub-second requirement, with no sustained consumer lag and zero message loss, reproduced across repeated runs (Section 6.2).

RQ3 asked how ML models can be integrated into the twin to exploit historical telemetry for predictive anomaly detection. This is answered at the level of feasibility: the platform's TimescaleDB telemetry is shown to be reducible to a model-ready feature matrix on which both supervised and unsupervised models train and evaluate (Section 6.4). The PoC establishes the integration path, from raw OCPP telemetry to a trained classifier, rather than a deployable detector, and the comparison against real field data makes explicit what remains before such a model could be trusted in production. The question is thus answered constructively, with its production-grade realisation scoped as future work (Section 7.3).

7.2 Limitations

These contributions are bounded in ways that follow directly from the project's scope and the threats discussed in Section 6.5.

The most fundamental limitation is that the active element of the DT, the closed control loop, is designed but not implemented. The `scheduler`, `actuation`, and `simulation-engine` services (Sections 4.3.4, 4.7) are deferred beyond the MVP (Section 5.1). The delivered system is therefore best characterised as a high-fidelity, real-data-validated Digital Shadow with an offline intelligence PoC, rather than a fully active DT that optimises charging and issues commands. The bidirectional vision is architecturally specified, and the read path's validation on real traffic derisks the integration on which the write path depends; its completion nonetheless remains future work.

The empirical evaluation is correspondingly bounded, as set out in Section 6.5: the ML results rest on synthetic data and leave real-world accuracy untested, the performance figures characterise a single scale on a single host, and the field validation, though substantial, draws on a single fleet and CSMS reporting only four measurands. Beyond the evaluation, several production-grade mechanisms specified in the architecture are realised in simplified form in the MVP: the canonical schema is serialised as JSON rather than as Protobuf under a schema registry; charger authentication uses OCPP basic authentication rather than mutual-TLS Security Profile 3; services load a shared public-key file rather than fetching it from the JWKS endpoint; the platform runs single-tenant rather than fully multi-tenant; and the raw-message audit trail is kept as structured logs rather than the Elasticsearch archive. Each is a scoped simplification whose production form is already specified in the architecture. Finally, the ingestion layer is implemented for OCPP alone; its protocol independence is a property of the design rather than one demonstrated empirically.

7.3 Future Work

The natural trajectory of the work follows from these limitations.

Closing the control loop. The foremost priority is to implement the deferred intelligence and actuation services: the rule-based scheduler and its eventual linear-programming successor (Subsection 4.3.4), the `actuation` service issuing commands through the CSMS management API, and the `simulation-engine`'s expected-versus-actual monitoring, which compares the twin's physically modelled expected state against observed reality to flag developing faults earlier than the reactive `alerting` service can. Completing these converts the validated Digital Shadow into the active DT the architecture envisions.

Learning from real data. The field corpus exposed both the limits of synthetic data and an opportunity: real chargers report voltage, current, and frequency signals the simulator never produced. The path to a deployable detector has three steps. First, a labelling process, feeding acknowledged alerts (Subsection 5.5.3) back as ground-truth labels, so the detector learns from production experience rather than a hand-built generator. Second, an online inference service that scores in-progress sessions against the trained model and emits predicted fault alerts. Third, drift monitoring that detects when live telemetry diverges from the training distribution and triggers retraining. Together these turn the offline PoC into a continuously improving production detector.

Scaling and production hardening. The performance evaluation should be extended into a scalability study across load levels and a multi-host deployment, towards the platform's longer-term design target of a thousand chargers. In parallel, the production-grade mechanisms deferred in the MVP (Section 7.2) should be brought into the implemented system.

Broader validation. Finally, the real-world validation should be widened beyond a single fleet and CSMS to multiple operators, vendors, telemetry profiles, and integration modes, establishing that the non-disruptive integration strategy generalises across the heterogeneity of deployed EVCI. A second ingestion adapter, for a different management protocol, would extend this to confirm the protocol independence of the layers above.

In closing, this dissertation has delivered and empirically validated (on nine months of real production traffic) the foundations of a real-time DT for EVCI: a non-disruptive, event-sourced platform that synchronises the physical state into a queryable live model within sub-second latency, together with a clear path from that validated Digital Shadow to the active, bidirectional DT the architecture sets out.

References

[1] Jannavarapu Vani Akhila et al. “Detecting Data Manipulation in Electric Vehicle Charging Stations Using Machine Learning Algorithm”. In: *2024 International Conference on Sustainable Communication Networks and Application (ICSCNA)*. 2024, pp. 899–904. DOI: 10.1109/ICSCNA63714.2024.10864159.

[2] Mohammad Alauthman et al. “Reinforcement Learning-Based Intrusion Detection for Electric Vehicle Charging Stations”. In: *2025 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA)*. 2025, pp. 1–5. DOI: 10.1109/ACDSA65407.2025.11165835.

[3] Ikram Benfarhat et al. “Temporal Convolutional Network Approach to Secure Open Charge Point Protocol (OCPP) in Electric Vehicle Charging”. In: *IEEE Access* 13 (2025), pp. 15272–15289. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3529526.

[4] K C Bishal, Amr Hilal, and Pawan Thapa. “Anomaly Detection in Electric Vehicle Charging Stations Using Federated Learning”. In: *2025 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT)*. 2025, pp. 1–7. DOI: 10.1109/GCAIoT68269.2025.11275550.

[5] Jesus Cumplido, Cristina Alcaraz, and Javier Lopez. “Collaborative Anomaly Detection System for Charging Stations”. In: *Lecture Notes in Computer Science*. Vol. 13555 LNCS. 2022, pp. 716–736. DOI: 10.1007/978-3-031-17146-8_35.

[6] Christos Dalamagkas et al. “The Open V2X Management Platform: An intelligent charging station management system”. In: *Information Systems* 129 (2025). DOI: 10.1016/j.is.2024.102494.

[7] Erika De Bardi and Giambattista Gruosso. “Analysis of a real-time co-simulation framework for smart and secure EV charging infrastructures”. In: *IECON Proceedings (Industrial Electronics Conference)*. 2025. DOI: 10.1109/IECON58223.2025.11221224.

[8] Deeksha Devendra et al. “Electric Vehicle Charging Station using Open Charge Point Protocol (OCPP) and oneM2M Platform for Enhanced Functionality”. In: *TENCON 2021 - 2021 IEEE Region 10 Conference (TENCON)*. 2021, pp. 1–5. DOI: 10.1109/TENCON54134.2021.9707311.

[9] Hongyan Dui et al. “CHART: Intelligent Sensing-Based Fault Diagnosis Method for EV Charging Infrastructure”. In: *IEEE Sensors Journal* 25 (20 2025), pp. 38279–38294. ISSN: 1558-1748. DOI: 10.1109/JSEN.2025.3602689.

[10] André Francisco, Jânio Monteiro, and Pedro Cardoso. “A Digital Twin of Charging Stations for Fleets of Electric Vehicles”. In: *IEEE Access* 11 (2023), pp. 125664–125683. DOI: 10.1109/ACCESS.2023.3330833.

[11] Jenish Gajera, Farhaan Jamal Mohamed, and Akramul Azim. "A Predictable and Real-Time Electric Vehicle Charging Framework with a Dynamic Protection System". In: *2025 IEEE International Conference on High Performance Computing and Communications (HPCC)*. 2025, pp. 1179–1186. DOI: 10.1109/HPCC67675.2025.00168.

[12] Nikolai Galkin, Chen Wei Yang, and Valeriy Vyatkin. "Automatic Generation of Charging Point's Digital Twin for Virtual Commissioning of Their Automation Systems". In: *IEEE Open Journal of the Industrial Electronics Society* 4 (2023), pp. 14–26. ISSN: 26441284. DOI: 10.1109/OJIES.2022.3230214.

[13] Kandarp Gandhi and Walid Morsi. "Impact of the Open Charge Point Protocol Between the Electric Vehicle and the Fast Charging Station on the Cybersecurity of the Smart Grid". In: *Canadian Conference on Electrical and Computer Engineering*. Vol. 2022-September. 2022, pp. 235–240. DOI: 10.1109/CCECE49351.2022.9918406.

[14] Dexin Gao, Xihao Lin, and Qing Yang. "Design and Application of a Fault Diagnosis and Monitoring System for Electric Vehicle Charging Equipment Based on Improved Deep Belief Network". In: *International Journal of Control, Automation and Systems* 20 (5 2022), pp. 1544–1560. DOI: 10.1007/s12555-021-0234-6.

[15] Ximming Gao, Gaoteng Yuan, and Mengjiao Zhang. "Fault Detection of Electric Vehicle Charging Piles Based on Extreme Learning Machine Algorithm". In: *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*. 2020, pp. 849–852. DOI: 10.1109/ICCMC48092.2020.ICCMC-000157.

[16] Zacharenia Garofalaki et al. "Electric Vehicle Charging: A Survey on the Security Issues and Challenges of the Open Charge Point Protocol (OCPP)". In: *IEEE Communications Surveys & Tutorials* 24 (3 2022), pp. 1504–1533. ISSN: 1553-877X. DOI: 10.1109/COMST.2022.3184448.

[17] Rohan Gupta et al. "A Wi-SUN Network-based Electric Vehicle Charging Station using Open Charge Point Protocol (OCPP) and oneM2M Platform". In: *2024 IEEE Applied Sensing Conference (APSCON)*. 2024, pp. 1–4. DOI: 10.1109/APSCON60364.2024.10465759.

[18] Shuai Hao et al. "Design and implementation of multi-protocol EV charging management cloud platform system". In: *2025 7th International Conference on Information Science, Electrical and Automation Engineering, ISEAE 2025*. Institute of Electrical and Electronics Engineers Inc., 2025, pp. 443–447. ISBN: 9798331510381. DOI: 10.1109/ISEAE64934.2025.11041717.

[19] S Hareshma and Parameswaran Sivraj. "Design and Development of Smart Charging Infrastructure for Electric Vehicles". In: *2023 3rd International Conference on Emerging Frontiers in Electrical and Electronic Technologies (ICEFEET)*. 2023, pp. 1–6. DOI: 10.1109/ICEFEET59656.2023.10452215.

[20] Christopher Hecht, Jan Figgener, and Dirk Uwe Sauer. "Protocols and Interfaces for EV Charging". In: *SpringerBriefs in Applied Sciences and Technology*. Vol. Part F2040. SpringerBriefs, 2024, pp. 77–89. DOI: 10.1007/978-3-031-47683-9_7.

[21] Sandeep Hegde et al. "Enhancing Anomaly Detection in Electric Vehicle Supply Equipment (EVSE) Networks Using Classical and Ensemble Learning Approaches". In: *2024 Control Instrumentation System Conference (CISCON)*. 2024, pp. 1–5. DOI: 10.1109/CISCON62171.2024.10696830.

[22] Ritesh Honnalli and Junaid Farooq. "Multimodal LLM-Guided Sequential Detection of Cyber Threats in Electric Vehicle Charging Systems". In: *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. 2025, pp. 1–6. DOI: 10.1109/COINS65080.2025.11125732.

[23] Md Sabbir Hossen et al. "Federated AI-OCPP Framework for Secure and Scalable EV Charging in Smart Cities". In: *Urban Science* 9 (9 2025). DOI: 10.3390/urbansci9090363.

[24] Sara Hsaini, Mounir Ghogho, and My El Hassan Charaf. "An OCPP-Based Approach for Electric Vehicle Charging Management". In: *Energies* 15 (18 2022). DOI: 10.3390/en15186735.

[25] International Electrotechnical Commission. *IEC 63110-1:2022 — Protocol for Management of Electric Vehicles Charging and Discharging Infrastructures. Part 1: Basic Definitions, Use Cases and Architectures*. 1.0. ISBN 978-2-8322-3868-4. International Electrotechnical Commission. Geneva, Switzerland, July 2022.

[26] International Electrotechnical Commission. *IEC 63584:2024 — Open Charge Point Protocol (OCPP)*. 1.0. Based on OCPP 2.0.1 Edition 3, adopted under the IEC Fast Track Procedure. International Electrotechnical Commission. Geneva, Switzerland, Dec. 2024.

[27] Pere Izquierdo Gómez et al. "Data-Driven Thermal Modelling for Anomaly Detection in Electric Vehicle Charging Stations". In: *2022 IEEE/AIAA Transportation Electrification Conference and Electric Aircraft Technologies Symposium (ITEC+EATS)*. 2022, pp. 1005–1010. DOI: 10.1109/ITEC53557.2022.9813767.

[28] Akmal Haris Bin Jasni, Muhammad Ikram Mohd Rashid, and Wan Muhammad Noor Sarbani Bin Mat Daud. "Development of EV Charging Pillar with IoT (Smart Energy Meter) Features". In: *Lecture Notes in Electrical Engineering*. Vol. 1213 LNEE. 2024, pp. 515–524. DOI: 10.1007/978-981-97-3851-9_44.

[29] Vispi Nevile Karkaria et al. "EV Charging Infrastructure Development Using Machine Learning". In: *2023 4th International Conference for Emerging Technology (INCET)*. 2023, pp. 1–6. DOI: 10.1109/INCET57972.2023.10170627.

[30] Venkatasamy Thiruppathy Kesavan et al. "Anomaly detection with grid sentinel framework for electric vehicle charging stations in a smart grid environment". In: *Scientific Reports* 15 (1 2025). DOI: 10.1038/s41598-025-00400-z.

[31] Werner Kritzinger et al. "Digital Twin in manufacturing: A categorical literature review and classification". In: *IFAC-PapersOnLine*. Vol. 51. Elsevier B.V., Jan. 2018, pp. 1016–1022. DOI: 10.1016/j.ifacol.2018.08.474.

[32] Duo Li et al. "Toward Resilient Electric Vehicle Charging Monitoring Systems: Curriculum Guided Multi-Feature Fusion Transformer". In: *IEEE Transactions on Intelligent Transportation Systems* 25 (12 2024), pp. 21356–21366. ISSN: 1558-0016. DOI: 10.1109/TITS.2024.3456843.

[33] Shuangxi Liu et al. "Fault Feature Recognition of Electric Vehicle Charging Infrastructure". In: *2023 8th Asia Conference on Power and Electrical Engineering (ACPEE)*. 2023, pp. 2619–2623. DOI: 10.1109/ACPEE56931.2023.10135690.

[34] Miaoli Lv et al. "Remote Measurement and Detection System for Electric Vehicle Charging Process and Its Application". In: *2025 International Conference on Computing Technologies Data Communication (ICCTDC)*. 2025. DOI: 10.1109/INCSST64791.2025.11210418.

[35] Pradeep Kumar Mallaiah et al. “ML-based Multiclass Anomaly Detection System for Charge Profile Manipulation Attacks in EV Charging Ecosystem”. In: *2025 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. 2025, pp. 1–6. DOI: 10.1109/SmartGridComm65349.2025.11204647.

[36] Jaime Afonso Martins and João Rodrigues. “Intelligent Monitoring Systems for Electric Vehicle Charging”. In: *Applied Sciences (Switzerland)* 15 (5 2025). DOI: 10.3390/app15052741.

[37] Lakshmi Menon and Narayanan Nithin. “Development of a communication simulator for Electric Vehicle charging based on GB/T”. In: *2021 6th International Conference on Communication and Electronics Systems (ICCES)*. 2021, pp. 177–183. DOI: 10.1109/ICCES51350.2021.9489019.

[38] Sagar Babu Mitikiri et al. “Regression based anomaly detection in electric vehicle state of charge fluctuations through analysis of electric vehicle charging infrastructure Data”. In: *Sustainable Energy, Grids and Networks* 42 (2025). DOI: 10.1016/j.segan.2025.101704.

[39] Ernest Fiko Morgan and Mohd Hasan Ali. “Digital Twin-Driven Cybersecurity for 5G/6G-Enabled Electric Vehicle Charging Infrastructure: A Review”. In: *Energies* 18 (22 2025). DOI: 10.3390/en18226048.

[40] Nikhil Sunilkumar Nambiar, Sadhana Jadhav, and Purushottam Balkrishna Ekande. “Software communication interface for OCPP-based DC fast charging stations”. In: Hershey, PA : Engineering Science Reference, 2023, pp. 102–114. DOI: 10.4018/978-1-6684-8816-4.ch007.

[41] Mohamad Syukri Mohamad Nizam et al. “Real-Time Energy Monitoring in Renewable EV Charging Stations: An ESP32-Based System Integrating Modbus, MQTT, and ESP-NOW Protocols”. In: *2024 IEEE 22nd Student Conference on Research and Development (SCOReD)*. 2024, pp. 339–344. DOI: 10.1109/SCOReD64708.2024.10872647.

[42] Open Charge Alliance. *OCPP 2.0.1, Part 2 - Specification*. 4th ed. Dec. 2025. URL: https://openchargealliance.org/.

[43] Open Charge Alliance. *OCPP 2.0.1, Part 4 - JSON over WebSockets implementation guide*. 4th ed. Edition 4, 2025-12-03. 2025. URL: https://openchargealliance.org/.

[44] Open Charge Alliance. *Open Charge Point Protocol 1.6*. 2nd ed. Edition 2 (FINAL), 2017-09-28. 2017. URL: https://openchargealliance.org/.

[45] Open Charge Alliance. *Open Charge Point Protocol JSON 1.6, OCPP-J 1.6 Specification*. 2017. URL: https://openchargealliance.org/.

[46] Subhabrata Pal, Rishiraj Sarker, and Avik Bhattacharya. “Artificial Intelligence-Enabled Intelligent Monitoring for Anomaly Detection in Inductor Current of PSFB Converter for Fast EV Charging Application”. In: *2024 IEEE 7th International Conference on Condition Assessment Techniques in Electrical Systems (CATCON)*. 2024, pp. 260–265. DOI: 10.1109/CATCON60527.2024.10831055.

[47] Konstantinos Papapostolou and Vassilis Kapsalis. “Development of a Smart Electric Vehicle Charging Station Management System”. In: *Proceedings of the 28th Pan-Hellenic Conference on Progress in Computing and Informatics*. 2025, pp. 257–263. DOI: 10.1145/3716554.3716593.

[48] Shaurya Purohit and Manimaran Govindarasu. "FL-EVCS: Federated Learning based Anomaly Detection for EV Charging Ecosystem". In: *Proceedings - International Conference on Computer Communications and Networks, ICCCN*. 2024. DOI: 10.1109/ICCCN61486.2024.10637543.

[49] Hari Raghav et al. "Establishing communication between EV – EVSE – CSMS based on IEC61851-1 and OCPP 2.0.1 Standard". In: *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 2023, pp. 1–6. DOI: 10.1109/ICCCNT56998.2023.10308172.

[50] Milad Rahmati. "Explainable Reliability Modeling and Runtime Monitoring of Software Systems in Electric Vehicle Charging Infrastructure". In: *IEEE Transactions on Engineering Management* 72 (2025), pp. 3667–3677. ISSN: 1558-0040. DOI: 10.1109/TEM.2025.3600381.

[51] R Sanjana et al. "A Proactive EV Charging Scheduler with Real-Time Short Circuit Detection Using IoT". In: *2025 International Conference on Computing Technologies Data Communication (ICCTDC)*. 2025, pp. 1–6. DOI: 10.1109/ICCTDC64446.2025.11157999.

[52] M S Sujatha et al. "RFID-Enabled Electric Vehicle Charging Stations: Enhancing Infrastructure and Communication with OCPP". In: *2025 6th International Conference on Inventive Research in Computing Applications (ICIRCA)*. 2025, pp. 244–252. DOI: 10.1109/ICIRCA65293.2025.11089579.

[53] Mohammadsoroush Tafazzoli et al. "EV Charging Infrastructure Optimization: Overcoming Scalability and Grid Management Challenges". In: *International Conference on Transportation and Development 2025: Transportation Safety and Emerging Technologies - Selected Papers from the International Conference on Transportation and Development 2025*. American Society of Civil Engineers (ASCE), 2025, pp. 343–354. ISBN: 9780784486191. DOI: 10.1061/9780784486191.030.

[54] Tancredi Testasecca et al. "A Digital Twin for Real-Time and Predictive Optimization of Electric Vehicle Charging in Microgrids Integrating Renewable Energy Sources". In: *Energies* 18 (21 2025). DOI: 10.3390/en18215605.

[55] Sunil Thapa, Sushil Poudel, and Mahmoud Abouyoussef. "TinyML-Enabled Intrusion Detection for Securing Electric Vehicle Supply Equipment (EVSE)". In: *2025 1st International Conference on Secure IoT, Assured and Trusted Computing (SATC)*. 2025, pp. 1–5. DOI: 10.1109/SATC65530.2025.11137032.

[56] Hanan Thwany et al. "Machine Learning Approaches for EV Charging Management: A Systematic Literature Review". In: *2023 IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*. 2023, pp. 1–6. DOI: 10.1109/AIBThings58340.2023.10292487.

[57] Viet Ha The et al. "Improving Anomaly Detection for Electric Vehicle Charging with Generative Adversarial Networks". In: *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. Association for Computing Machinery, 2025, pp. 1800–1809. ISBN: 9798400706295. DOI: 10.1145/3672608.3707823.

[58] Mohammad Waseem et al. "State-of-the-Art and Advancement of Charging Infrastructure in Electric Mobility: An Integrated Review". In: *Energies* 17 (23 2024). DOI: 10.3390/en17236137.

[59] Lingxiao Wei et al. “Online Incremental Learning Algorithms for Real-time Fault Diagnosis in Charging Stations”. In: *ICITEE ’24: Proceedings of the 7th International Conference on Information Technologies and Electrical Engineering*. 2025, pp. 201–207. DOI: 10.1145/3717934.3717965.

[60] Lutong Yang, Hui Gao, and Haowei Duan. “Research on intelligent Diagnosis method of Electric vehicle Charging Fault based on artificial intelligence expert system”. In: *Journal of Physics: Conference Series*. Vol. 1848. 2021. DOI: 10.1088/1742-6596/1848/1/012125.

[61] Yong Yang et al. “Electric vehicle charging equipment component abnormal prediction technology”. In: *2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Vol. 5. 2021, pp. 165–170. DOI: 10.1109/IAEAC50856.2021.9391064.

[62] Yue Yang and Jiarou Li. “Electric Vehicle Charging Anomaly Detection Method Based on Multivariate Gaussian Distribution Model”. In: *Proceedings of the Asia Conference on Electrical, Power and Computer Engineering*. Association for Computing Machinery, 2022. ISBN: 9781450396127. DOI: 10.1145/3529299.3531488.

[63] Gang Yu et al. “Digital twin enabled transition towards the smart electric vehicle charging infrastructure: A review”. In: *Sustainable Cities and Society* 108 (2024). DOI: 10.1016/j.scs.2024.105479.

[64] Gang Yu et al. “Towards Cognitive EV Charging Stations Enabled by Digital Twin and Parallel Intelligence”. In: *2021 IEEE 1st International Conference on Digital Twins and Parallel Intelligence (DTPI)*. 2021, pp. 290–293. DOI: 10.1109/DTPI52967.2021.9540103.

[65] Ming Yu. “Attention-Based CNN-BiGRU Hybrid Model for Enhanced Parameter Prediction and Early Fault Warning in Electric Vehicle Charging”. In: *2025 IEEE 5th International Conference on Software Engineering and Artificial Intelligence (SEAI)*. 2025, pp. 345–350. DOI: 10.1109/SEAI65851.2025.11108766.

[66] Weiya Zhang et al. “Research of integrating prior knowledge into abnormal behavior recognition model of EV charging station”. In: *2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*. 2023, pp. 862–867. DOI: 10.1109/EEBDA56825.2023.10090512.

[67] Xiong Zhang, Li Ma, and Xiangyang Zhang. “Exploration of Electric Vehicle Charging State Monitoring and Fire Early Warning Based on Neural Network”. In: *2024 5th International Conference on Clean Energy and Electric Power Engineering (ICCEPE)*. 2024, pp. 1363–1366. DOI: 10.1109/ICCEPE62686.2024.10931513.

[68] Rui Zhao et al. “Fault Diagnosis of Electric Vehicle Charging Station Based on Empirical Wavelet Transform and Entropy”. In: *2021 40th Chinese Control Conference (CCC)*. 2021, pp. 4504–4509. DOI: 10.23919/CCC52363.2021.9549401.

[69] Shah Meeran Zia et al. “Heterogeneous Cloud-Based EV Charging Management System with Rust and Multi-Device Integration”. In: *2025 Energy Conversion Congress & Expo Europe (ECCE Europe)*. 2025, pp. 1–5. DOI: 10.1109/ECCE-Europe62795.2025.11238613.