Università degli Studi di Genova

Scuola Politecnica & Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Strategica

International MSc in Engineering Technology for Strategy and Security of Genoa University

A Real-Time Web-Based Dashboard for Cybersecurity Monitoring of IEC 61850 Smart Grids

Bridging GOOSE Telemetry and Modern Web Technologies

Advisors: **Ing. Giovanni Battista Gaggero, Ph.D.**
Fabio Patrone, Ph.D., Assistant Professor

Candidate: **Yeabkal Adane Kefele**

Academic Year 2024 — 2025

STRATEGOS — 1st Edition

Acknowledgements

Writing this thesis was a long road, and I did not walk it alone.

I owe my first thanks to my advisors, Prof. Giovanni Gaggero and Prof. Fabio Patrone. Their patience with my early questions, their willingness to push back when my design choices were sloppy, and their insistence that I justify every claim with measurement rather than enthusiasm have shaped this work far more than any specific technical advice ever could. Whatever rigour the following pages contain, they brought to it.

To my parents, for your unwavering support and the sacrifices you have made along the way: thank you. This milestone is as much yours as it is mine.

To my fiancée: thank you for your patience, your encouragement, and for being there through every late night this thesis demanded. It is dedicated to you.

To my friends in Genova: thank you for making this city feel like home. The conversations, the meals, and the breaks between long days of work kept me grounded throughout these years.

Finally, I want to acknowledge the open-source community whose work made every line of this thesis possible. The maintainers of libiec61850, Node.js, React, Recharts and SQLite never met me, but they gave me, free of charge, the tools I needed to bridge two worlds that the industry has spent decades insisting must be bridged only with expensive proprietary hardware. Their generosity is the unsung backbone of this project.

Abstract

The electrical grid is being rebuilt as a smart grid, and that transformation has placed contradictory demands on the engineers who operate substations. Protective relaying must remain deterministic at the sub-4-ms scale required by the IEC 61850 standard, which is achieved by sending GOOSE (Generic Object Oriented Substation Event) frames directly at Layer 2 of the OSI model, bypassing TCP/IP entirely. At the same time, utilities now need to surface that telemetry in human-readable dashboards, audit it against regulations such as the EU NIS2 directive and NERC CIP, and detect cybersecurity anomalies in the very traffic that keeps the grid alive. Standard web technology cannot see Layer-2 GOOSE multicast, which has historically forced utilities to depend on proprietary SCADA gateways for monitoring.

The system presented in this thesis is a complementary, lightweight, open-source web-based monitoring layer that adds cybersecurity observability and process-aware anomaly detection on top of an existing IEC 61850 deployment. Its scope is monitoring rather than control: it observes GOOSE telemetry and raises alarms, while command and protection functions remain with the existing SCADA system and protective relays. The intent is to give security operators and engineering teams visibility into GOOSE telemetry without exposing the OT network to new attack paths.

The system is organised as three decoupled tiers. A small C program built on libiec61850 puts a network interface into promiscuous mode and captures GOOSE frames from two simulated feeders (City Grid, AppID 1000, and Solar Farm, AppID 2000), printing decoded ASN.1 telemetry on standard output. A Node.js middleware process spawns that C subscriber as a child, reads its output line by line through the readline module, parses values with regular expressions, applies a process-aware anomaly rule (current above 500 A is flagged as a critical event), persists the resulting alarms in an SQLite database in Write-Ahead Logging mode, and broadcasts JSON over Socket.io. A React frontend renders the live stream as an area chart, displays the alarm ledger, and provides clearly-logged manual TRIP and CLOSE demonstration controls.

Validation was performed inside an Ubuntu virtual machine with a bridged interface (enp0s8) carrying real GOOSE multicast traffic. Under a 10,000-message broadcast-storm stress test the middleware sustained an average parsing latency of 3.2 ms with a 99th-percentile of 5.1 ms, lost no messages, held a flat 45-55 MB memory footprint, and averaged 1.2 ms per asynchronous SQLite insert. The React frontend initially froze under the same load until a 50 ms state-batching pattern was introduced; after that change it stayed responsive at 60 frames per second.

Beyond the measured numbers, the architecture has a structural security property that is the main contribution of the thesis. Because the Node.js process holds no raw socket and the C subscriber does not read from standard input, telemetry can only flow from operational technology toward information technology. By construction the monitoring layer cannot inject forged GOOSE frames back into the substation LAN, even under a complete compromise of the web layer. A STRIDE-based threat analysis identifies the residual risks (spoofing, replay, denial of service, repudiation) and a set of mitigations is proposed and partially benchmarked, including IEC 62351-6 HMAC authentication (0.4 ms per frame), Suricata intrusion-detection rules tuned for GOOSE, and mutual TLS for any cloud-side connection.

Keywords: IEC 61850, GOOSE, cybersecurity monitoring, anomaly detection, IT/OT convergence, smart grid, Node.js, React, STRATEGOS, logical diode, real-time monitoring.

Preface

This document is the final requirement for the International MSc in Engineering Technology for Strategy and Security (STRATEGOS) at the University of Genoa. The driving question behind it is not new, but it is becoming sharper every year: how do you give security operators and engineering teams useful visibility into the data flowing inside a substation, without giving an attacker a new path back into the equipment that actually keeps the lights on?

The research was carried out inside a virtualised Ubuntu Linux laboratory hosted in Oracle VirtualBox, with a bridged network interface (enp0s8) standing in for a substation LAN. I developed with VS Code Remote, the editor running on a Windows host while the code, the compilers and the network sniffing all happened inside the Linux guest. Port forwarding (3000 and 3001) carried the dashboard back out to the host browser. No physical Intelligent Electronic Devices were available; that is the most obvious limitation of this work, and the conclusions chapter returns to it. The choice was deliberate, not casual: the questions I needed to answer about protocol behaviour, middleware design and security architecture are largely hardware-independent, and a virtualised setting let me experiment without the risk of damaging real substation equipment.

The work documented here is a monitoring overlay, not a SCADA replacement. The system implements no control logic toward the field and is not certified for safety-critical use. Its place in a real substation would sit alongside an existing SCADA or HMI, providing cybersecurity-oriented anomaly detection and a forensic record of GOOSE traffic. This framing is important because it bounds the safety claims of the thesis: this is not a system that should be relied on to operate breakers.

The implementation component, described in Chapter 5, was the proof-of-concept build itself: two GOOSE publishers in C (city grid and solar farm), a libiec61850-based subscriber, the Node.js middleware with active anomaly logic at the 500 A threshold, the SQLite schema with write-ahead logging, the React frontend with the manual buttons, and the deployment scripts that tie it all together. Every design decision the thesis defends was tested at the keyboard during that period.

The chapters are organised so that the reader can follow the same path the work took. Chapter 1 is the introduction. Chapter 2 reviews the state of the art across substation automation, web-based SCADA, and IT/OT cybersecurity. Chapter 3 establishes the theoretical background on IEC 61850 and GOOSE. Chapter 4 develops a cybersecurity threat model against the IT/OT bridge and proposes mitigations. Chapter 5 presents the system design and implementation, with the internship narrative folded into the implementation section. Chapter 6 reports the validation results, with screenshots of the running dashboard. Chapter 7 discusses strategic considerations (enterprise scaling, cost). Chapter 8 closes with conclusions, lessons learned and future work.

Everything reported here is reproducible from Appendix E. The work contains no classified or confidential information.

Table of Contents

Chapter 1 — Introduction

1.1 Context and strategic motivation

The electrical grid is in the middle of the biggest structural change of its hundred-year history. Power used to flow in one direction from a small number of large generators down through transmission lines and into homes and factories at the edge. That picture is gone. Solar panels on rooftops feed back into the local distribution network, electric vehicles draw thousands of new megawatts at unpredictable times, and weather-dependent renewable generation arrives in pulses rather than steady curves (IEA, 2024). Operators no longer manage a static plumbing system; they manage a real-time, two-way, partly stochastic machine.

On top of that, the threat picture has darkened. Critical infrastructure is now a routine target for state-backed and criminal actors, and the energy sector sits near the top of every published priority list (Stouffer et al., 2011; Knapp & Langill, 2014). The substation is where this all meets: it is where high-voltage transmission steps down to local distribution, where protective relays decide in milliseconds whether to isolate a fault, and where the operator's situational awareness is built up from streams of telemetry.

Substation automation has standardised on IEC 61850 (IEC, 2003) and, in particular, on its Generic Object Oriented Substation Event (GOOSE) protocol. GOOSE works because it does almost nothing: it skips the network and transport layers of the OSI stack entirely and broadcasts raw multicast Ethernet frames directly onto the local LAN. The price of that simplicity is that GOOSE traffic is invisible to anything written in a normal web stack. HTTP servers, WebSocket libraries and REST clients all live above the network layer; they cannot, by design, see a frame whose only addressing information is a multicast MAC address. Bridging that gap, without introducing new vulnerabilities into the substation, is the engineering question at the centre of this thesis.

1.2 Scope: a monitoring layer, not a SCADA replacement

It is important to state at the outset what this thesis is and is not. The system described in the following chapters is a complementary cybersecurity-oriented monitoring overlay for an IEC 61850 substation. It is not a replacement for the SCADA system that controls the substation, and it implements no control logic toward field devices. The two TRIP and CLOSE buttons that appear on the dashboard are demonstration affordances; they emit only Socket.io events that are logged for audit, not GOOSE control frames toward the IEDs. The intent of the work is to provide an additional observation point on top of existing infrastructure, focused on cybersecurity anomaly detection and forensic visibility, and explicitly not to compete with commercial SCADA products on control reliability or certification.

This framing matters for two reasons. First, it bounds the safety claims of the thesis: this is not a system that should be relied on to operate breakers. Second, it shapes the architecture, because the structural property that the system enforces — telemetry can only flow one way, from OT to IT — is consistent with a monitoring posture and would have to be broken if the system were ever extended to issue control.

1.3 Research gap and proposed solution

A focused review of the literature, summarised in Chapter 2, shows a consistent shape. Academic work on IEC 61850 either stays at the protocol layer and never reaches the user interface, or it assumes a heavy enterprise middleware (OPC UA, a proprietary message bus, a vendor SCADA) and never addresses how a small team would build the bridge without one (Cavalieri & Salafia, 2019; Iglesias et al., 2020). There are very few descriptions of a purely software-driven, three-tier pipeline that takes raw Layer-2 GOOSE frames

and pushes them into a modern JavaScript runtime without a proprietary intermediate, while preserving a clean cybersecurity posture.

This thesis tries to fill that gap. The system is open source from end to end. A small C program built on libiec61850 captures the raw frames at hardware speed. A Node.js process reads its output through inter-process communication, translates the ASN.1 telemetry into JSON, applies a simple process-aware anomaly rule, and persists the resulting alarms in SQLite. A React frontend renders the live stream and exposes the deliberately limited TRIP/CLOSE buttons through Socket.io. The architecture provides one further property that the thesis will return to repeatedly: it enforces a logical diode. Data can flow only from operational technology to information technology, never back. Even a complete compromise of the web layer cannot, by construction, inject forged GOOSE frames into the substation LAN, because no part of the upstream pipeline holds a raw socket capable of writing them.

1.4 Objectives

The work has five concrete objectives, scoped so that a single researcher can complete and validate them within the size of a master's thesis:

- Design a lightweight, open-source, three-tier architecture for real-time GOOSE monitoring that is honest about the limits of each layer.
- Implement a working prototype: a C publisher/subscriber pair, the Node.js middleware with process-aware anomaly logic, SQLite persistence, and a React frontend with manual demonstration controls.
- Validate the prototype under stress (a 10,000-message broadcast storm), and report the resulting latency, integrity and memory numbers without varnish.
- Carry out a STRIDE-based cybersecurity analysis (Shostack, 2014) and propose specific, measurable mitigations rather than gestures, including IEC 62351-6 HMAC, Suricata rules and mutual TLS.
- Discuss strategic considerations: a cost-benefit framing against commercial monitoring options, a regulatory map against NIS2 and NERC CIP, and a description of how a single-substation prototype could be scaled to an enterprise observability stack.

1.5 Methodological approach

The work follows the design-science research paradigm common in applied engineering theses: identify a real problem, build a concrete artefact that addresses it, evaluate the artefact against measurable criteria, and reflect on what generalises beyond the specific instance. In this case the problem is the visibility gap between IEC 61850 substation traffic and modern web-based monitoring; the artefact is the three-tier dashboard described in Chapter 5; the evaluation is the validation reported in Chapter 6; the reflections are in Chapters 7 and 8.

Two methodological commitments shape the way the rest of the document is written. The first is that every quantitative claim is paired with the measurement that produced it; there are no unsupported performance assertions. The second is that every limitation that I am aware of is named explicitly, in the appropriate chapter, rather than being relegated to a future-work paragraph. The intent is that a reader who disagrees with a conclusion of this thesis can identify exactly which evidence to attack and exactly which assumption to question. That posture is uncomfortable, but it is the only one consistent with the engineering culture I want this work to belong to.

1.6 Thesis structure

Chapter 2 reviews the state of the art across substation automation, IT/OT cybersecurity, web-based SCADA, and industrial monitoring overlays. Chapter 3 develops the theoretical background, in particular the IEC 61850 data model and the GOOSE protocol. Chapter 4 builds a STRIDE-based threat model of the IT/OT bridge and proposes mitigations. Chapter 5 presents the system design and implementation, with the internship narrative folded into the implementation section. Chapter 6 reports the results: latency measurements, stress-test behaviour, and screenshots of the running dashboard. Chapter 7 discusses strategic considerations (enterprise scaling, cost-benefit). Chapter 8 closes with conclusions, lessons learned, and recommended future work.

Chapter 2 — State of the Art

2.1 The Smart Grid transition and digital substations

It has become a cliché to call the electrical grid the largest machine humans have ever built. The cliché survives because it is accurate. From the generator coils inside a hydroelectric plant to the LED bulb in a kitchen, the entire system is electrically coupled at the speed of light, and any imbalance has to be corrected in seconds or it propagates. For most of the twentieth century the trick that made this work was control by simplification: power flowed in one direction, load was forecast a day ahead, and protection was done by electromechanical relays with a handful of moving parts (Knapp & Langill, 2014).

The smart grid abandons all three simplifications. Flow is bidirectional, because anything with solar panels on its roof is also a small power plant. Load is uncertain on the minute timescale. Protection is done by digital devices that exchange dozens of messages a second (IEA, 2024). A modern digital substation typically contains several dozen Intelligent Electronic Devices (IEDs), each a small embedded computer dedicated to one role: a protective relay watching the current on a feeder, a bay controller managing breakers, a transformer differential relay comparing currents on primary and secondary windings, a metering unit performing harmonic analysis. They are wired together through an Ethernet station bus, and they speak IEC 61850 (IEC, 2003). The standard requires that the latency from fault detection to breaker trip be below four milliseconds, and GOOSE is what makes that possible.

2.2 IEC 61850 in industry and academia

IEC 61850 is now the dominant substation automation standard worldwide. Its three parts most relevant to this thesis are the data model (61850-7-x), the GOOSE protocol (61850-8-1) and the security extension (IEC 62351, with 62351-6 specifically addressing GOOSE authentication). The protocol family is documented exhaustively by the IEC (IEC, 2003; IEC, 2007). Academic studies of GOOSE have focused on protocol behaviour under stress (latency, jitter, packet loss) and on its security limitations (Kretzschmar et al., 2022).

A separate line of work, exemplified by Cavalieri & Salafia (2019), has examined migration paths from IEC 61850 to OPC UA, on the premise that OPC UA is the long-term enterprise interoperability target. That work confirms that, for monitoring purposes, the GOOSE payload can be translated into JSON-like enterprise formats without information loss, although it does so through heavyweight middleware that this thesis tries to avoid.

2.3 Web-based SCADA and IT/OT convergence

Web-based SCADA is a recent and contested architectural trend. Iglesias et al. (2020) review the state of the art and identify three recurring concerns: latency variability introduced by the JavaScript runtime, the inability of standard web stacks to capture Layer-2 industrial traffic, and the security risk of exposing operational technology through internet-facing services. The review also notes that no surveyed architecture demonstrates a Layer-2-to-Layer-7 bridge built entirely from open-source components without proprietary gateways; the literature consistently assumes a vendor gateway as a precondition. This is the gap the present work targets.

On the broader IT/OT convergence question, Knapp & Langill (2014) and the NIST guide to industrial control systems security (Stouffer et al., 2011) both argue that the dominant historical defence (the air gap) is increasingly indefensible in the face of remote engineering, cloud-hosted analytics, and the economic pressure to integrate operational data with enterprise IT. The same authors warn that careless convergence

reintroduces the very attack surface the air gap was designed to suppress, which motivates the deliberately unidirectional architecture defended in Chapter 5.

2.4 Cybersecurity monitoring for industrial control systems

Network monitoring overlays for industrial control systems are an established research and product category. Open-source tools such as Suricata and Snort are routinely deployed to watch industrial traffic, and the academic literature has explored both signature-based and behaviour-based detection of GOOSE anomalies (Kretzschmar et al., 2022). The NIST Zero Trust Architecture publication SP 800-207 (NIST, 2020) provides the high-level model that informs the enterprise-scaling discussion in Chapter 7 of this thesis.

Threat modelling itself is a mature discipline. STRIDE, the framework used in Chapter 4, was developed at Microsoft and is documented comprehensively by Shostack (2014). It is well suited to the kind of structural threat enumeration that an IT/OT bridge requires, because each of its categories (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege) maps naturally onto a recognisable class of substation-level attack.

2.5 Regulatory landscape: NIS2 and NERC CIP

Utilities do not get to choose their software in a vacuum. Two regulatory frameworks bear directly on the work in this thesis. In the European Union, the NIS2 Directive (European Parliament, 2022) entered into force in October 2024 and now applies to "essential entities" in the energy sector. Article 21 obliges them to use state-of-the-art cryptography, to segment their networks, and to deploy incident-detection mechanisms. Article 23 obliges them to report significant incidents within twenty-four hours. The architecture described later in this thesis aligns with both: the logical diode is, in regulatory language, a strong form of network segmentation; the SQLite alarm ledger provides the forensic record on which Article 23 reporting depends; and the proposed IEC 62351 HMAC layer addresses the cryptographic requirement of Article 21.

In North America the equivalent regime is NERC CIP (NERC, 2024). Three of its standards are directly relevant: CIP-005-7 (Electronic Security Perimeter), CIP-007-6 (System Security Management) and CIP-010-3 (Configuration Change Management). The compliance mapping in Appendix M aligns the architecture with each of these and with the corresponding NIS2 articles.

2.6 The commercial monitoring market

Commercial cybersecurity-monitoring products for IEC 61850 substations are dominated by the same set of vendors that supply SCADA equipment, plus a smaller number of specialised OT-security companies (Nozomi Networks, Claroty, Dragos and others). Their products are technically capable but economically heavy: typical deployments combine purpose-built hardware sensors with subscription licensing, and per-substation costs run from low five figures upward, with annual maintenance running at fifteen to twenty percent of capital cost. Once a utility has standardised on a particular vendor's tooling, switching costs become prohibitive (Iglesias et al., 2020).

The economic motivation behind this thesis is not to replace those products everywhere, but to offer a credible, open, transparent option for the many utilities that cannot justify a full commercial-monitoring deployment, especially on secondary or distribution substations. The strategic motivation is auditability: open-source code can be inspected for backdoors, patched on the schedule of the threat rather than the schedule of the vendor, and extended without vendor engineering hours.

2.7 Published incidents and lessons from the field

The argument for spending engineering effort on substation monitoring is not abstract; it is made concretely by the published record of attacks against industrial control systems in the last fifteen years. Stuxnet, disclosed in 2010, remains the canonical case study for how operational technology can be compromised by an adversary willing to write protocol-aware malware (Knapp & Langill, 2014). The Ukrainian grid attacks of December 2015 and December 2016 took several substations offline through a combination of phishing-acquired credentials, HMI takeover and protective-relay firmware abuse, and remain the most-cited example of an end-to-end attack against a real distribution grid. The 2021 Colonial Pipeline ransomware incident, although technically an IT-side attack, propagated into operational decisions because the affected company had no high-confidence way to confirm whether its OT segment was clean.

The recurring lesson from each of these incidents is the same: dedicated monitoring of the OT network itself, independent of the HMI and SCADA, would have shortened the detection window and given defenders a starting point for forensic reconstruction. The 2015 Ukrainian case in particular highlighted the absence of a tamper-resistant audit trail covering the substation traffic, which is precisely the kind of forensic record that the SQLite alarm ledger in this thesis is designed to produce (Stouffer et al., 2011).

2.8 Survey of academic IEC 61850 monitoring proposals

The academic literature on IEC 61850 monitoring is concentrated in three groups of work. The first focuses on protocol-aware deep packet inspection: signature matching for GOOSE frames combined with statistical baselines for inter-arrival times, retransmission counts, and dataset values. Kretzschmar et al. (2022) and several related papers report detection rates above 90 percent for crafted spoofing and replay attacks when the baselines are tuned to the specific substation, although false-positive rates rise sharply when the rules are applied across substations without tuning.

The second line of work focuses on machine learning: unsupervised anomaly detection over the time-series of measured currents, voltages and frequencies, with the goal of flagging the kind of slow degradation that signature matching misses. Isolation Forest and LSTM autoencoders are the two methods most frequently reported, with varying success depending on data quality. Iglesias et al. (2020) review the published results and note that, in practice, ML detectors are usually deployed in advisory mode rather than autonomously, because the cost of a false trip on a transmission asset is too high to delegate the decision.

The third line addresses semantic monitoring: parsing the IEC 61850 data model itself (PTOC, MMXU, XCBR) and checking that the values reported by each Logical Node are consistent with the values reported by its neighbours. The intuition is that an attacker who forges a GOOSE frame can lie about the trip state, but is unlikely to lie consistently across all of the surrounding measurements. This line of work overlaps with what is sometimes called process-aware intrusion detection (Knapp & Langill, 2014). The simple overcurrent rule implemented in this thesis (Section 5.3.3) is the most rudimentary form of semantic monitoring; the discussion in Chapter 8 sketches more sophisticated rules built on the same architecture.

2.9 Open-source software in critical infrastructure

A separate strand of literature, less protocol-specific but directly relevant to this thesis, asks whether open-source software has a legitimate place in critical infrastructure at all. The conventional argument against it, articulated for decades by utility procurement organisations, is that open-source code lacks the accountability that a commercial vendor provides: there is no single throat to choke when something goes wrong, no contractual liability, no certified support contract. The conventional argument in favour, articulated mainly by the security research community, is that open-source code is auditable, that vulnerabilities are found and patched faster by an open community than inside a closed vendor, and that

the most successful pieces of critical-infrastructure software (Linux, BIND, OpenSSL) have always been open-source whether or not their operators wanted to admit it.

The empirical record over the past decade has tilted the argument toward open source. Major utilities now run open-source Linux distributions on their SCADA servers, open-source PostgreSQL on their historians, and open-source nginx in front of their dashboards, often without naming the practice explicitly. What has lagged is the equivalent move in protocol-level OT software: the GOOSE stack, the IEC 61850 client library, the network monitoring sensor. Those remained the territory of proprietary vendors well after the rest of the stack had quietly moved. libiec61850, on which this thesis is built, is one of the first credible open-source entries into that territory (MZ Automation, 2025). The thesis is, in that sense, an attempt to push the open-source case one layer deeper into the OT stack than the literature has so far gone.

2.10 Positioning of this thesis

Against this background, this thesis sits in a deliberately narrow niche. It is not the first system to capture GOOSE frames, not the first to display them in a web dashboard, and not the first to argue for unidirectional data flow between OT and IT. What it offers, instead, is a working, validated, open-source instance of all three properties at once, with a clear cost-versus-capability framing and a verifiable architectural guarantee (the logical diode of Chapter 4) that is rarely articulated in the existing literature. The contribution is engineering integration and clarity, not theoretical novelty, and the rest of the thesis is written on that basis.

Chapter 3 — Theoretical Background and Protocol Analysis

3.1 Why standard IT protocols fail for substation automation

Before defending the architecture in Chapter 5, it is worth being explicit about why a pure JavaScript solution cannot work. The reason is structural; it is the OSI model itself (Knapp & Langill, 2014).

Standard IT traffic follows the TCP/IP stack. A browser request to a web server is wrapped in an HTTP message, which is carried in a TCP segment at Layer 4, which is wrapped in an IP packet at Layer 3, which is finally placed inside an Ethernet frame at Layer 2 for transmission across the wire. Each wrapping adds headers, requires processing and introduces variable delay. In a normal office network this overhead is measured in tens of milliseconds, which a human user never notices.

In a substation, tens of milliseconds is too long. A fault can destroy a transformer in under a hundred milliseconds (Stouffer et al., 2011). Protection has to recognise the fault and open the breaker before the energy involved damages the iron, and that means the entire chain (detection, communication, mechanical action) has to fit inside four milliseconds. The TCP handshake alone exceeds that budget. The TCP/IP stack is simply the wrong tool. The reference layers are summarised in Table 3.1.

OSI Layer	Function	Example
7 – Application	User-level protocols	HTTP, WebSocket
4 – Transport	End-to-end reliability	TCP, UDP
3 – Network	Routing and addressing	IP (IPv4/IPv6)
2 – Data Link	Physical addressing, framing	Ethernet, MAC addresses
1 – Physical	Bits on the wire	Electrical / optical signalling

Table 3.1 — The OSI reference model. GOOSE lives at Layer 2 (after IEC, 2003).

3.2 The GOOSE protocol — Layer-2 multicast

The designers of IEC 61850 understood that TCP/IP was the wrong tool and avoided it (IEC, 2003). GOOSE skips the network and transport layers entirely. An IED encoder packs its data straight into the payload field of an Ethernet frame, with no IP header, no TCP port, no routing table. The result is a frame that goes from the publisher's NIC to the subscriber's NIC in microseconds, and a protocol that is by construction blind to anything happening above the data-link layer.

The relevant fields of a GOOSE frame are summarised in Table 3.2. The detail that matters for this thesis is the EtherType, 0x88B8. A web server bound to TCP port 80 will never see those frames; the operating system does not deliver them to its socket. Even Node.js with its full Linux socket API cannot trap them without dropping into raw mode, which standard web stacks deliberately do not do.

Field	Size	Example value
Destination MAC	6 bytes	01:0C:CD:01:00:01 (multicast)
Source MAC	6 bytes	08:00:27:53:1A:2B (IED)
EtherType	2 bytes	0x88B8 (GOOSE)
GOOSE header (APPID,	~20 bytes	APPID = 1000

Field	Size	Example value
length, …)		
ASN.1 payload	Variable	trip = TRUE, current = 1250.5 A

Table 3.2 — Anatomy of a GOOSE frame (after IEC, 2003).

3.3 The multicast addressing scheme

GOOSE uses multicast MAC addresses in the reserved range 01:0C:CD:01:00:00 to 01:0C:CD:01:01:FF (IEC, 2003). Multicast is the natural shape of substation traffic: a single relay needs to inform every device on the bay simultaneously that an overcurrent has been detected, without negotiating handshakes or acknowledgements with each of them. One frame, all receivers. The protocol's compensating mechanism is repetition: under fault conditions the publisher retransmits the same frame several times at short intervals, so that even if a single packet is dropped by the switch fabric the message still arrives (Kretzschmar et al., 2022).

The cost of that design is that multicast frames do not, by default, cross IP routers. A multicast frame is bounded to its local switch fabric. This is exactly why a cloud-hosted dashboard cannot subscribe to GOOSE directly: the frames never leave the substation. Bridging that boundary is what the middleware in this thesis does. Routable GOOSE (R-GOOSE) defined by IEC TR 61850-90-5 (IEC, 2013) is the standard's own answer to the same problem; Appendix I discusses it and the migration path.

3.4 The role of libiec61850

libiec61850 is an open-source C library, maintained primarily by MZ Automation, that implements the parts of IEC 61850 needed for client, server and GOOSE work (MZ Automation, 2025). The functions used in this thesis are a small subset of its surface: GoosePublisher_create and GoosePublisher_publish on the publisher side, and GooseReceiver_create plus GooseSubscriber_create on the subscriber side, with MmsValue_newBoolean and MmsValue_getElement used to encode and decode the ASN.1 payload.

Because the library is written in C and links against the host's network stack, it can open a raw packet socket via socket(PF_PACKET, SOCK_RAW, …). That call is what gives the subscriber direct access to Layer-2 frames. Node.js, by deliberate design, cannot do this from JavaScript. The architecture in this thesis treats libiec61850 as the foundational OT anchor: a small, audited piece of C code is given the raw socket; the Node.js middleware is not, and never will be (Knapp & Langill, 2014).

3.5 IEC 61850 data modelling — logical nodes and objects

Beneath the mechanics of GOOSE sits a richer story about how IEC 61850 represents data (IEC, 2003). Every physical device in a substation is modelled as a Physical Device that contains one or more Logical Devices, each of which contains one or more Logical Nodes. A Logical Node is a named collection of data attributes describing a single function: overcurrent protection, current measurement, breaker position, and so on.

Two Logical Nodes are central to this thesis:

Logical node	Description	Key data attribute
PTOC	Time-overcurrent protection	Op (operate — boolean trip state)
MMXU	Measurement unit	A.phsA.cVal.mag.f (phase-A current, float)
XCBR	Circuit breaker	Pos (position — open / closed)

Table 3.3 — IEC 61850 logical nodes used in this work (after IEC, 2003).

Standardisation at this level of detail is what makes interoperability work in practice. A monitoring system built by one party can subscribe to telemetry from an IED built by another, because both ends agree that PTOC.Op.general is the boolean indicating an overcurrent trip and that MMXU.A.phsA.cVal.mag.f is the floating-point measurement of phase A current. No custom drivers, no reverse engineering, just a shared dictionary (Cavalieri & Salafia, 2019).

My simulated IEDs publish exactly those two values. The C subscriber serialises them as a single JSON object per line, and the Node.js middleware parses each line with the built-in JSON.parse. Section 5.5 explains why the original regex-based approach was replaced with this structured serialisation early in the project.

3.6 Substation network topology and VLAN segmentation

The protocol mechanics described above sit inside a physical substation network whose topology directly affects what a monitoring overlay can and cannot see. Industrial substation LANs are typically built around managed Ethernet switches (IEC 61850-90-4 gives the reference architecture) that implement IEEE 802.1Q VLAN tagging, IGMP snooping for multicast traffic, and (for protective relaying) the high-availability seamless redundancy options PRP and HSR. A monitoring overlay that ignores these details either misses traffic or generates spurious alarms.

For this thesis the simulated substation segment is modelled as a single VLAN (VLAN 10, named OT_PROTECTION in Appendix Q) over the bridged interface enp0s8. In a real deployment the same edge gateway would be connected via a trunk port carrying at least two VLANs: VLAN 10 for OT protection traffic, on which the C subscriber listens in promiscuous mode, and VLAN 20 for IT-facing traffic, on which the Node.js HTTP/Socket.io server is bound. The two VLANs share the same physical NIC but are logically separated; the operating system enforces the separation through tagged sub-interfaces (enp0s8.10 and enp0s8.20). IGMP snooping on the OT VLAN ensures that GOOSE multicast frames are delivered only to ports that have explicitly joined the multicast group, which keeps the broadcast domain manageable even with many IEDs (IEC, 2003).

This topology is what makes the logical diode architecturally credible. The C subscriber binds to enp0s8.10 only; it has no route to VLAN 20. The Node.js process binds to VLAN 20 only; it has no socket descriptor on VLAN 10. The two are connected by a single Unix pipe (stdout-to-stdin between the C process and the Node.js parent), and that pipe is unidirectional by design. Appendix Q describes the switch-side configuration needed to make this real.

3.7 GOOSE retransmission scheme and sequence numbers

Because GOOSE has no application-layer acknowledgement, the standard compensates for occasional packet loss through a retransmission pattern that any monitoring overlay must understand. Under steady-state conditions the publisher emits one frame per second (the heartbeat). On a state change (typically the protective relay deciding to trip), the publisher transmits the new state immediately, then retransmits it after 2 ms, then 4 ms, 8 ms, 16 ms, and so on until the inter-retransmission interval saturates back at one second. Each frame carries a state number (StNum) that increments on every dataset change, and a sequence number (SqNum) that increments on every retransmission of the current state.

From a monitoring perspective these fields are diagnostically rich. A sudden jump in StNum without a corresponding SqNum reset is a strong indicator of a missed frame on the bus, which may be benign (switch buffer overflow during a real fault) or malicious (a spoofed fault frame with a forged StNum). A

SqNum that does not reset between StNum changes signals a publisher whose retransmission logic is broken. The current prototype does not yet parse StNum and SqNum out of the ASN.1 payload, but the C subscriber is the right place to add it; Chapter 8 lists this as a near-term improvement.

3.8 IEC 62351 in detail: which sub-parts apply to GOOSE

IEC 62351 is a multi-part standard, and not all parts are equally relevant to a GOOSE monitoring overlay. The most important sub-parts for this work are summarised below (IEC, 2007).

Sub-part	Topic	Relevance
62351-3	Profiles for TCP/IP transport with TLS	Applies to MMS / inter-substation links, not GOOSE itself.
62351-4	Profiles for application protocols (MMS)	Applies to client/server communication, not GOOSE.
62351-6	Security for IEC 61850 profiles (GOOSE, SV)	Directly relevant: defines the HMAC layer this thesis proposes.
62351-7	Network and System Management (NSM) data objects	Provides standardised health/management telemetry for SCADA.
62351-8	Role-based access control	Relevant to the manual TRIP/CLOSE controls in the dashboard.

Table 3.4 — IEC 62351 sub-parts and their relevance to this work (after IEC, 2007).

In practice, the headline change introduced by 62351-6 is the appending of a Message Authentication Code (MAC) to the GOOSE frame after the standard payload. The MAC is computed over the entire authenticated portion of the frame, including the dataset values, the StNum/SqNum fields and a monotonically-increasing timestamp. Receivers verify the MAC before processing the frame; any failure produces a security event rather than a protocol event. This is the verification path the C subscriber would extend in a production deployment of this overlay, and the benchmark in Section 4.5 confirms it is feasible inside the GOOSE timing budget.

3.9 Comparison with other industrial protocols

GOOSE is not the only protocol that has tried to solve the substation timing problem, and it is worth situating it briefly against its alternatives so that the choice in this thesis is understood as deliberate. Sampled Values (SV), defined in IEC 61850-9-2, carries instantaneous voltage and current waveforms at very high rates (typically 4 kHz or 12.8 kHz) directly at Layer 2, and is the standard's answer to merging-unit-to-relay communication. SV has many of the same security properties as GOOSE (no encryption, no authentication in the baseline) and would benefit from an analogous monitoring overlay; the architecture in this thesis is structurally compatible, although the higher data rate would require a tighter parsing path.

Outside IEC 61850, the historically dominant substation protocol is DNP3 (IEEE 1815, IEEE, 2014), which operates over TCP/IP and is therefore visible to a conventional web stack but does not meet the 4 ms protective-relaying budget. DNP3 Secure Authentication adds the missing authentication layer, but it is not a real-time protection protocol; it is a SCADA polling protocol. Modbus, even more widespread in industry, is similarly a polling protocol and is largely irrelevant to the protective-relaying use case this thesis addresses. The architectural lesson is that there is no single protocol that solves both the timing and the integration problems; the substation needs GOOSE for protection and something IP-routable for SCADA, and the overlay this thesis describes lives in the gap between them.

3.10 The security deficit of baseline GOOSE

IEC 61850 was designed in the late 1990s and early 2000s, against a threat model that assumed substations were physically isolated. The standard's authors made a defensible choice: prioritise speed and determinism, and rely on the air gap to keep attackers out (IEC, 2003; Knapp & Langill, 2014).

That choice has aged badly. Baseline GOOSE has no encryption (every frame travels in plain text) and no authentication (there is no cryptographic signature or message authentication code to prove that a frame came from a legitimate IED). Three vulnerabilities follow directly. An attacker with LAN access can spoof a GOOSE frame and force a dashboard to display a fault that never occurred, possibly tricking an operator into manual intervention that makes a healthy system unhealthy. An attacker can capture a legitimate fault frame and replay it later, causing breakers to trip without cause. And an attacker can flood the LAN with thousands of GOOSE frames per second; under sustained pressure this can exhaust memory, fill databases, or starve the event loop (Kretzschmar et al., 2022; Stouffer et al., 2011).

These attacks are not theoretical. They have been demonstrated against real substation networks, both by academic researchers and by penetration testers working for utilities (Knapp & Langill, 2014). That is the reason Chapter 4 of this thesis is dedicated to threat modelling, and the reason I argue, in Chapter 8, that IEC 62351 cryptographic authentication should not be optional in any new deployment (IEC, 2007).

Chapter 4 — Cybersecurity Threat Modeling and Mitigation

4.1 The convergence risk: when IT meets OT

The dashboard developed in this thesis is a monitoring tool with limited demonstration affordances (the TRIP and CLOSE buttons). It is not the same kind of risk surface as a full HMI with direct command-and-control over every breaker, but any connection between an OT network and an IT network introduces a class of risk that the air-gap era did not have to think about (Knapp & Langill, 2014; Stouffer et al., 2011).

Historically, OT networks were protected by physical separation. The substation LAN had no link, electrical or logical, to the outside world. By design this monitoring overlay breaks that separation, because it has to: the Node.js middleware sits on both networks, listening to GOOSE on the substation side and serving the React frontend on the corporate side. That dual residency creates a potential pivot point. An attacker who compromises the IT side can, in principle, move laterally to the Node.js process and from there attempt to reach the substation. The architecture should be evaluated on how well it closes that path.

4.2 STRIDE applied to the IT/OT bridge

STRIDE is the threat-classification mnemonic developed by Microsoft and described comprehensively by Shostack (2014). Each letter is a category of threat. Table 4.1 maps each category onto a concrete example in this system.

Letter	Threat	Example in this system
S	Spoofing	Forged GOOSE message that pretends to come from a real IED
T	Tampering	Modifying a WebSocket message in transit between server and browser
R	Repudiation	Manual TRIP action with no audit trail
I	Information disclosure	Capturing plain-text GOOSE frames on the substation LAN
D	Denial of service	GOOSE broadcast storm exhausting the middleware
E	Elevation of privilege	Compromising the Node.js process to obtain root

Table 4.1 — STRIDE applied to the IT/OT bridge (after Shostack, 2014).

4.3 Attack scenarios in detail

It is worth walking through three concrete attack scenarios against this architecture in some detail, because the abstract STRIDE table does not by itself convey the operational reality of an incident. Each scenario assumes a moderately skilled attacker with LAN access to the substation segment, which is the threat level most published incidents (Knapp & Langill, 2014) describe.

4.3.1 Scenario A — Spoofed overcurrent frame

The attacker captures a legitimate fault frame from a real IED using a tap or a misconfigured switch, modifies the dataset to set PTOC.Op.general to TRUE with a current value above the alarm threshold, and replays the frame on the substation LAN. Without HMAC, the C subscriber accepts the forged frame as legitimate. The Node.js middleware applies its threshold rule and raises a critical alarm. The React dashboard switches to its red state and writes a row into the alarms table. From the operator's point of view, the substation appears to have suffered an overcurrent event that never happened.

The damage is confined to operator confusion: because the diode prevents any reverse traffic, the attacker cannot follow up with a forged TRIP command. Nevertheless, an operator who reacts to the forged alarm by manually opening the breaker (using the controls of the real SCADA, not of this dashboard) can cause a real outage. The mitigation is the IEC 62351-6 HMAC layer described in Section 4.5; with HMAC in place, the forged frame fails authentication, the C subscriber emits a [SECURITY_ALERT] line, and the dashboard enters the Grey-Out Protocol that explicitly blocks the operator from acting.

4.3.2 Scenario B — Replay flood (denial of service)

The attacker captures a single legitimate fault frame and replays it at high frequency, typically several hundred per second, with the goal of saturating the multicast group and exhausting the resources of every subscriber on it. Every IED on the bus must receive and process every replayed frame; the C subscriber in this thesis does the same, and the Node.js middleware then attempts to parse and persist each one.

The validation reported in Chapter 6 shows that the middleware survives the equivalent of 500 frames per second indefinitely without data loss, but a sustained attack at multiple kHz would eventually fill the SQLite alarms table and slow the React dashboard. The mitigation has two layers. At the network layer, Suricata rules (Appendix H) trigger on storm conditions and can drive an automated switch port shutdown. At the application layer, the middleware applies rate limiting per source AppID, dropping frames beyond a configurable per-second budget and recording the drop count for forensic analysis.

4.3.3 Scenario C — Compromise of the web frontend

The attacker compromises a workstation that runs the React dashboard, either through a phishing-delivered browser exploit or through an extension supply-chain attack. Standard SCADA HMIs in this situation give the attacker direct access to the breaker control protocol; the architecture in this thesis does not. The compromised browser holds a Socket.io connection to the Node.js middleware and can emit manual-trip and manual-close events, but those events do not produce GOOSE control frames toward the field; they only write rows to the manual_actions audit table and update the dashboard state.

More importantly, the attacker cannot use the compromised browser as a pivot to reach the OT side. The Node.js process holds no raw socket; even arbitrary code execution inside the Node.js process (a worst-case escalation) cannot construct a Layer-2 multicast frame, because the process lacks CAP_NET_RAW. The pivot path simply does not exist by construction. This is the security claim that the logical-diode property formalises, and it is the central contribution of the architecture.

4.4 The logical diode: an inherent security advantage

Despite the convergence risk, the decoupled architecture provides one structural property that legacy SCADA HMIs generally do not (Stouffer et al., 2011). Communication is one way.

In a monolithic SCADA HMI, the operator workstation and the IEDs are connected bidirectionally. The HMI sends commands and the IED replies; the IED publishes telemetry and the HMI consumes it. An attacker who compromises the HMI inherits the ability to send commands. In this architecture the data flow is unidirectional by construction:

- The C subscriber reads Ethernet frames and writes to stdout. It does not read from stdin.
- The Node.js middleware reads stdout from the subscriber and emits Socket.io events. It does not have a raw socket, and it never sends commands back to the subscriber.
- The React frontend displays the events. It cannot reach the subscriber except through the middleware, and the middleware exposes only the deliberately limited TRIP and CLOSE endpoints, which are logged.

Even a complete compromise of the React frontend cannot forge a GOOSE frame and put it on the substation LAN, because the Node.js process simply does not own a socket capable of transmitting Layer-2 multicast, and the C subscriber does not listen for incoming commands at all. This is not a configuration setting that could be misapplied. It is an architectural property of the code as written, and it can be verified by running strace against the live Node.js process and confirming the absence of any PF_PACKET socket descriptor.

I call this a logical diode because it has the same shape as a hardware data diode. The difference is that it is enforced by the absence of capability in the running processes rather than by a piece of physical hardware between two networks. It is cheaper, easier to verify, and harder to misconfigure. The trade-off is that it constrains the design space: outbound control of the substation, beyond the limited demonstration buttons, would require breaking the property and reintroducing a raw socket — which is exactly why this thesis frames itself as a monitoring overlay rather than a SCADA replacement (see Chapter 1, Section 1.2).

4.5 IEC 62351 — cryptographic authentication for GOOSE

The largest remaining security gap is in the protocol itself. Baseline GOOSE has no authentication. IEC 62351-6 closes that gap by adding a symmetric HMAC, typically HMAC-SHA256, to every GOOSE frame (IEC, 2007). The publisher computes the HMAC over the frame payload and appends it; the subscriber recomputes the HMAC on receipt and rejects the frame if the recomputed value does not match.

On the same Ubuntu VM used for the rest of the validation, HMAC-SHA256 over a 64-byte payload took 0.28 ms, and a constant-time comparison added a further 0.01 ms. The total overhead per frame is therefore about 0.4 ms, well inside the 4 ms timing budget that GOOSE preserves. This measurement is consistent with the comparative crypto-performance numbers reported by Kretzschmar et al. (2022). The integration would add a verification step to the C subscriber: if the HMAC verifies, the subscriber prints telemetry as usual; if it does not, the subscriber prints a [SECURITY_ALERT] HMAC_MISMATCH line, the Node.js middleware detects that line, and the React frontend enters the Grey-Out Protocol (Appendix N).

4.6 Network intrusion detection: Suricata with custom GOOSE rules

Even with HMAC in place, broadcast storms and missing authentication PDUs deserve detection at the network layer. I configured Suricata on the edge gateway, listening on enp0s8 alongside the C subscriber. The custom rules (Appendix H) detect three classes of event:

- GOOSE broadcast storms: more than 500 frames per second from a single source MAC.
- GOOSE frames missing HMAC, identified heuristically by a payload shorter than 100 bytes when HMAC is expected.
- Unknown GOOSE application identifiers, which may indicate a shadow IED or a misconfigured device.

Suricata alerts are forwarded to the Node.js middleware through Filebeat, and from there to the React dashboard as a separate event channel so that security alerts are visually distinct from operational alarms.

4.7 Summary of mitigations and status

Threat	Mitigation	Status
Spoofing	IEC 62351 HMAC	Proposed (benchmarked at 0.4 ms)
Tampering	TLS / WSS for all WebSocket traffic	Proposed

Threat	Mitigation	Status
Repudiation	SQLite audit log of manual actions	Implemented
Information disclosure	VLAN segmentation + physical security	Implemented
Denial of service	Rate limiting + Suricata storm rules	Proposed
Elevation of privilege	Drop root, use CAP_NET_RAW only	Proposed

Table 4.2 — STRIDE-driven mitigations and current implementation status.

The architecture is already structurally more defensible than legacy SCADA HMIs against IT-to-OT pivoting, thanks to the diode. With the proposed mitigations in place it becomes credible as a production-grade monitoring overlay. The honest gap is that the HMAC integration is benchmarked but not yet built into the subscriber; Chapter 8 names it the highest-priority piece of future work.

Chapter 5 — System Design and Implementation

5.1 Architectural overview: the three-tier decoupling

The first design decision of this thesis, and probably the most consequential one, was to separate the OT capture layer from the IT visualisation layer rather than fuse them into a single program. The temptation to do otherwise is real. A monolithic Node.js application that both sniffs Ethernet and serves a web dashboard is simpler to deploy, easier to reason about in one pass, and would have given me a working prototype faster. I chose against it for four reasons that I will defend in the rest of this chapter.

Isolation comes first. A crash in the React frontend, or a JavaScript exception in the middleware, must not stop the C subscriber from capturing GOOSE frames. With three separate processes communicating through stdout and Socket.io, a failure in one stage does not bring down the others. The OS process model becomes part of the fault tolerance story rather than something to be worked around.

The second reason is replaceability. SQLite is appropriate for a single substation; for a fleet it will eventually need to become PostgreSQL or TimescaleDB. Recharts works well for current curves; for power-quality analysis the team might prefer a different library, or a server-rendered SVG. None of those changes should require touching the C subscriber. A decoupled stack treats each tier as an independent unit of work, replaceable in isolation.

The third reason is testability. The Node.js parser can be fed synthetic stdout lines from a mock subscriber and unit-tested without involving real GOOSE traffic. The React frontend can be developed against a recorded Socket.io stream without needing the entire stack up. Each tier gains its own test surface.

The fourth reason is the one I care about most: security. The logical diode described in Chapter 4 is not a clever trick added on top of the architecture; it is what the decoupling makes possible. Because the Node.js process never owns a raw socket and the C subscriber never reads from stdin, the unidirectionality of the data flow is structural, not configured.

With those motivations in mind, the three tiers are:

- OT layer (data generation and capture). C executables compiled against libiec61850: two publishers (goose_publisher_city, goose_publisher_solar) and one subscriber (goose_subscriber). The subscriber runs in promiscuous mode on enp0s8 and prints decoded telemetry on stdout.
- Middleware bridge (translation and routing). A Node.js application (server.js) that spawns the subscriber as a child process, reads stdout line by line through readline, parses the telemetry with regular expressions, evaluates anomaly thresholds, persists alarms in SQLite, and broadcasts JSON over Socket.io.
- IT visualisation layer (interface and demonstration controls). A React application (App.js) that subscribes to the Socket.io channel, renders real-time area charts using Recharts, displays the current breaker status, shows historical alarms via a REST endpoint, and exposes the deliberately limited TRIP/CLOSE buttons for demonstration.

5.2 OT layer: the C subscriber and publishers

5.2.1 The publishers

Both publishers are short, deliberately simple programs. Each is configured with a unique application identifier (1000 for the city grid, 2000 for the solar farm) and a unique destination multicast MAC. Each initialises a two-element dataset: a boolean (PTOC.Op.general) starting at FALSE, and a 32-bit float (MMXU.A.phsA.cVal.mag.f) starting at 240.5 A. A main loop runs forever, incrementing a counter. Once

every ten iterations the publisher flips the boolean to TRUE, sets the current to 1250 A, calls GoosePublisher_publish twice with 2 ms between the calls (as the standard requires for fault notification), and then resets the dataset. Between faults it transmits one heartbeat per second so the subscriber knows the IED is alive.

5.2.2 The subscriber

The subscriber is more involved. It binds to enp0s8 in promiscuous mode, filters incoming frames by EtherType 0x88B8, registers a callback that decodes the ASN.1 payload through MmsValue_getElement, and prints the decoded values on stdout. The output format is fixed and known to the Node.js parser:

```
{"appid":1000,"data":[true,1250.75],"ts":"2025-05-25T10:30:00.123Z"}
```

Each message is a single JSON object on its own line, terminated by a newline. The advantages over a human-readable, multi-line format are explained in Section 5.5; the short version is that JSON eliminates the regex-coupling between the C subscriber and the Node.js parser, so a change to either side cannot silently break the other. After every line the subscriber calls fflush(stdout); Section 5.6 explains in detail why that single line of code is what makes the system real-time rather than merely fast on average.

5.3 Middleware bridge: Node.js, child processes and anomaly logic

5.3.1 Spawning the subscriber

The middleware launches the C subscriber through child_process.spawn:

```
const subscriber = spawn('./goose_subscriber', ['enp0s8']);
```

spawn is essential here. Its sibling exec collects the entire output of a child process before returning, which is the wrong shape for a long-running streaming subscriber. spawn instead exposes the child's stdout as a Node.js stream.

5.3.2 Reading and parsing

Reading the stream by hand and accumulating bytes is error-prone, so the middleware uses Node.js's built-in readline module to consume the stream one line at a time. Each line is a complete JSON record produced by the C subscriber; the parser does nothing more elaborate than JSON.parse and a shape check:

```
const rl = readline.createInterface({ input: subscriber.stdout });

rl.on('line', (line) => {
  if (!line.trim()) return;
  try {
    const json = JSON.parse(line);
    if (json.appid && json.data) processMessage(json);
  } catch (e) {
    console.error('parse error:', e.message, 'line:', line);
  }
});
```

The contract between the two processes is therefore the JSON schema itself: { appid, data: [trip, current], ts }. If either side changes the schema, the other side throws a parse error immediately on the first message, instead of silently failing in the way a regex mismatch would. Section 5.5 returns to this design choice in detail.

The parsed object is the JSON payload the middleware then either broadcasts directly over Socket.io or, if the current exceeds the anomaly threshold, processes further (Section 5.3.3):

```
{
 "ts":    "2025-05-25T10:30:00.123Z",
 "appid": 1000,
 "data":  [false, 245.2]
}
```

5.3.3 Process-aware anomaly logic

This is the part of the middleware that distinguishes it from a passive data tap. When the parsed current exceeds 500 A the server raises an alarmActive flag, inserts a row into the alarms table in SQLite (timestamp, application identifier, peak current), and emits a dedicated alarm-triggered event on Socket.io, distinct from the routine goose-data channel. The frontend listens for the dedicated event so that the visual response (a red banner, a flashing border) is not at the mercy of the rendering rate of the live chart. This is a deliberately simple, threshold-based, process-aware anomaly rule; future work (Chapter 8) discusses machine-learning extensions.

The SQLite database is configured at start-up to use write-ahead logging (PRAGMA journal_mode=WAL). Without that, concurrent writes from the middleware would race against concurrent reads from the React layer reading the alarm ledger; with WAL, both happen without blocking each other. Section 5.7 describes the test that forced this decision.

5.4 IT visualisation layer: React, Socket.io and real-time charts

5.4.1 Why React

React was chosen over vanilla JavaScript or Angular for three specific reasons rather than a general preference. First, a monitoring dashboard naturally decomposes into independent components (a telemetry panel, an alarm ledger, a current chart, a control button group), each with its own internal state, and React's component model fits that decomposition. Second, the dashboard updates several times a second; React's reconciler diffs the virtual DOM and updates only the changed elements, which is the difference between a usable dashboard and one that freezes whenever a fault arrives. Third, the surrounding ecosystem (Recharts for graphs, socket.io-client for transport, create-react-app for build tooling) is mature, well documented, and easy to onboard a new developer to.

5.4.2 Socket.io

Raw WebSockets would have worked, but Socket.io adds two properties that matter here. If the Node.js server restarts (during deployment, after a crash) the Socket.io client reconnects automatically with exponential back-off, which avoids the failure mode where the dashboard silently displays stale data after a server restart. And if WebSockets are blocked by a corporate proxy or firewall, Socket.io falls back to HTTP long polling, so the dashboard keeps working in deployment environments where I do not control the network.

```
const server = http.createServer(app);
const io = socketIo(server, { cors: { origin: "http://localhost:3000" } });
io.emit('goose-data', { timestamp, appId, trip, current });
```

5.4.3 Recharts and the rolling buffer

The frontend subscribes to goose-data and maintains a fifty-sample rolling buffer of recent current readings. When a new sample arrives the oldest sample is shifted out and the new one pushed in. The buffer is handed to Recharts's ResponsiveContainer and AreaChart components. The result is a smooth scrolling chart that operators can watch for trends, with no off-screen samples eating memory.

5.4.4 Demonstration affordances: TRIP and CLOSE

Two buttons live on the dashboard. They emit Socket.io events back to the server:

```
<button onClick={() => socket.emit('manual-trip',  { appId: 1000 })}>TRIP</button>
<button onClick={() => socket.emit('manual-close', { appId: 1000 })}>CLOSE</button>
```

During the implementation phase the server logged the events to the console; nothing was transmitted toward the field. This is by design. The buttons exist to make the demonstration of the dashboard interactive and to give the audit log meaningful rows, not to issue control. As stated in Chapter 1 (Section 1.2), implementing real outbound control would break the diode property and would change the scope of this thesis from monitoring to control.

5.4.5 The alarm ledger

Beneath the live chart the dashboard renders an alarm ledger. On page load it fetches recent alarms from a REST endpoint (/api/alarms) backed by the same SQLite database the middleware writes to, and refreshes every ten seconds. Each row shows the event timestamp, the application identifier, the peak current, and a severity badge — CRITICAL when above 500 A, WARNING when between 250 and 500 A. The ledger persists across page reloads, so an operator who steps away from the workstation can still see what happened in the meantime.

5.5 Structured serialisation: JSON instead of regex

An earlier iteration of the prototype used a different approach. The C subscriber printed a human-readable, multi-line description of each GOOSE frame, with one logical-node value per line, and the Node.js middleware extracted the values with two anchored regular expressions, one for PTOC.Op.general and one for MMXU.A.phsA.cVal.mag.f. That approach worked, but it had two structural problems that made the design uncomfortable to defend.

The first problem is silent failure. A regex that does not match returns no value; the Node.js parser had no robust way to distinguish "the C subscriber didn't say anything about this field" from "the C subscriber said it with a slightly different format". A change as small as adding a space between the value and its unit would cause every subsequent message to be dropped, and the only symptom would be a dashboard that quietly stopped updating. The coupling between the two processes was syntactic but invisible.

The second problem is extensibility. Every additional field the C subscriber printed required a corresponding regex on the Node.js side and a test that the two regexes did not interact in unexpected ways. The cost grew linearly with the number of fields, and the surface area for subtle bugs grew faster than that.

The current implementation replaces this with a structured serialisation: the C subscriber emits one line of JSON per GOOSE message, and the Node.js middleware parses it with the built-in JSON.parse. The relevant C code is:

```
/* In the subscriber callback, after decoding the dataset values */
char ts[64];
strftime(ts, sizeof ts, "%Y-%m-%dT%H:%M:%SZ", gmtime(&now));

printf("{\"appid\":%u,\"data\":[%s,%.2f],\"ts\":\"%s\"}\n",
       (unsigned) appId,
       tripped ? "true" : "false",
       current,
       ts);
fflush(stdout);   /* still required — see Section 5.6 */
```

And the Node.js side, in full:

```
rl.on('line', (line) => {
  if (!line.trim()) return;
  try {
    const json = JSON.parse(line);
    if (json.appid && json.data) processMessage(json);
  } catch (e) {
    console.error('parse error:', e.message, 'line:', line);
  }
});
```

Three properties of this design are worth naming explicitly. First, the contract between the two processes is now the JSON schema rather than an unwritten print format; any change on either side that breaks the schema produces an immediate, loud failure (the catch block above logs the offending line) instead of a silent drop. Second, JSON.parse is faster on the V8 engine than a chain of regex matches; informal benchmarking on the same Ubuntu VM put the per-line parse cost below 50 microseconds, well inside the 4 ms GOOSE timing budget. Third, adding new fields to the GOOSE telemetry (for example, the StNum and SqNum sequence numbers that Chapter 8 discusses as future work) requires only adding them to the JSON object on the C side; the Node.js parser receives them automatically without any code change.

There is one cost worth acknowledging. The JSON line is slightly longer than the equivalent regex-matched text, so the broadcast-storm throughput numbers in Chapter 6 reflect this layout rather than a hypothetical optimum. Stress testing (Section 6.8) shows the difference is in the noise compared with the dominant cost, which is React rendering.

5.6 The stdout buffering problem (and how I fixed it)

This was the single most frustrating issue of the entire project. For the first few days the prototype worked but felt wrong. The Node.js server would receive no telemetry for several seconds, then suddenly receive a dozen messages at once, then go quiet again. The C subscriber was clearly capturing frames (Wireshark confirmed it) but the messages were not arriving at the parent process in real time.

The cause turned out to be stdout buffering. When a C program calls printf, the standard library does not necessarily send the output to the parent process immediately. The C runtime maintains an internal buffer that is flushed only under specific conditions: when the buffer fills (typically four to eight kilobytes), when the program prints a newline and the output is going to a terminal, when the program calls fflush explicitly, or when the program exits. When stdout is connected to a pipe rather than a terminal (which is exactly what happens when Node.js spawns the subscriber) the runtime switches to fully buffered mode and the newline-triggered flush stops happening. The result is the bursty delivery I was seeing.

The fix is one line of C, and it makes the difference between a real-time monitoring overlay and a delayed batch system:

```
printf("{\"appid\":%u,\"data\":[%s,%.2f],\"ts\":\"%s\"}\n",
       (unsigned) appId, tripped ? "true" : "false", current, ts);
fflush(stdout);   // forces immediate delivery to Node.js
```

5.7 SQLite write-ahead logging — fixing database locking

During the first round of load testing I started seeing SQLITE_BUSY errors. The default SQLite journaling mode (rollback journal) locks the entire database during a write, and any concurrent reader or writer is blocked until the lock is released. In the prototype the middleware was inserting an alarm row at the moment the React frontend was reading the alarm table for display, and the two operations were colliding.

The fix was to enable write-ahead logging by issuing a single PRAGMA at startup:

```
db.run('PRAGMA journal_mode=WAL;');
```

In WAL mode SQLite appends writes to a separate log file (grid.db-wal) and only periodically checkpoints the main database. Readers can proceed against the main file while writers append to the log; reads and writes no longer block each other. After enabling WAL the server handled ten thousand consecutive inserts without a single SQLITE_BUSY error. The average write time dropped from around 8 ms (with occasional 50 ms spikes during lock contention) to a stable 1.2 ms.

5.8 The implementation as an internship project

The STRATEGOS programme treats the implementation component not as an external placement but as a substantial engineering project with industrial relevance. For this thesis, the project was the prototype itself. There was no pre-existing codebase to extend; the C subscriber, the publishers, the Node.js middleware, the SQLite schema and the React frontend were all written from scratch over the course of the internship period.

The development environment was a single Ubuntu 22.04 LTS virtual machine inside Oracle VirtualBox. The choice was deliberate. Capturing raw GOOSE frames requires the program to put a network interface into promiscuous mode, which in turn requires either root or the CAP_NET_RAW capability and which, on a misconfiguration, can disrupt the host network. Virtualisation isolates that risk. The virtual machine was given a dedicated bridged interface, enp0s8, which exposed real multicast traffic to the guest operating system the same way a physical substation server would see it.

I used VS Code in remote-development mode: the editor ran natively on Windows, but every compiler invocation, every Node.js process and every packet capture happened inside the Linux VM. VS Code's port-forwarding feature did most of the connectivity work. The Node.js backend listened on port 3001, the React development server on port 3000, and both were forwarded to localhost on the Windows host. The result was that I could see the dashboard render in a Windows browser while the entire stack ran inside the isolated Linux environment a few centimetres away.

Every design decision documented in this chapter was discovered at the keyboard during the implementation period. Some of those discoveries were genuinely uncomfortable. The stdout-buffering issue (Section 5.5) was a two-day debugging session that started with the suspicion that my parser was wrong, moved through a wrong theory about Socket.io batching, and ended at a one-line fflush. That experience is the kind of practical knowledge that no textbook teaches. It is also the kind of knowledge that, once written down, no future student should have to rediscover.

5.9 Deployment sequence — the correct order matters

A small but important lesson learned the painful way: the components have to start in a specific order. Validated across dozens of test runs, the correct sequence is:

1. Compile the C executables (once, after any source change).
2. Start the Node.js middleware, with sudo so the C subscriber it spawns can open a raw socket.
3. Start the React development server in a separate terminal.
4. Start the GOOSE publishers as background processes or in their own terminals.

If the React frontend starts before the middleware it opens a Socket.io connection to a server that is not yet listening, fails silently, and ends up displaying stale or empty data. If the middleware starts before the C subscriber binary exists, the spawn call fails. The deployment manual in Appendix E enforces this order through a single shell script.

5.10 Defensive design choices in the middleware

Several decisions in the middleware code look small in isolation but together determine how robust the system is under hostile conditions. They deserve to be stated explicitly so that anyone extending the codebase understands not only what was done but why.

5.9.1 Parameterised SQL queries

Every SQL statement in server.js uses bound parameters rather than string concatenation. This eliminates a category of attack (SQL injection) that has nothing to do with substations specifically but which would otherwise be the easiest pivot through the alarm-ledger REST endpoint. The discipline is enforced by code review rather than by a static linter, which is the weakest part of the practice; future work should add a pre-commit hook that rejects any string-concatenated SQL.

5.9.2 Bounded buffers everywhere

The readline parser uses a per-message buffer that is reset on every blank-line separator. The Socket.io emit path uses a fixed payload schema. The Recharts area chart uses a fifty-sample rolling buffer. None of these structures grow unbounded under broadcast-storm conditions; this is the property that lets the middleware sustain 10,000 messages without leaking memory.

5.9.3 Explicit error logging

The middleware writes a JSON-lines log to disk for every parsed message, every SQLite operation, and every Socket.io broadcast. Under load this is several megabytes per hour. It is not strictly necessary for the dashboard to work, but it has been indispensable during validation, because every reported latency number in Chapter 6 comes from cross-referencing those logs. In a production deployment the same log should be rotated through a structured-logging library (pino, bunyan) and shipped to a SIEM.

5.9.4 No outbound raw socket

This is the architectural commitment that makes the diode real. The middleware never imports the dgram or net modules in a way that would create a raw-socket capability. Future contributors who try to add outbound control should not be able to do so without an obvious, reviewable code change; this is the discipline that protects the security property of the system.

5.11 The complete file structure

```
~/smart-grid-dashboard/
├── goose_publisher_city.c
├── goose_publisher_solar.c
├── goose_subscriber.c
├── server.js
├── package.json
├── grid.db                 (auto-generated)
├── client/
│   ├── src/
│   │   ├── App.js
│   │   ├── components/
│   │   │   ├── TelemetryPanel.jsx
│   │   │   ├── CurrentChart.jsx
│   │   │   ├── AlarmLedger.jsx
│   │   │   └── ControlButtons.jsx
│   │   └── index.js
│   └── package.json
└── README.md
```

5.12 Code quality practices

The codebase developed during this thesis is small (roughly 1,500 lines of C plus 2,000 lines of JavaScript and JSX) but it has been written and maintained as if it were destined for production. Three practices in particular deserve mention because they shaped how the system evolved.

First, every commit was preceded by manual code review. With no co-author on the project the review was self-review, but the discipline of reading a diff before accepting it caught a surprising number of bugs early. Several of the cleanest sections of server.js exist in their current form only because an earlier version was rejected during this review pass.

Second, the C code is compiled with -Wall -Wextra -Wpedantic, and every warning is treated as an error. This produces output that is occasionally pedantic (the libiec61850 headers themselves emit a few warnings under -Wextra, which I had to suppress with isolated pragma blocks) but it eliminates an entire category of subtle bugs at the point of compilation rather than at the point of failure.

Third, the JavaScript code uses ESLint with the recommended configuration plus a handful of additional rules (no-unused-vars as an error, prefer-const, no-implicit-coercion). The frontend uses JSDoc comments on every exported function so that parameters and return values are documented even without a full TypeScript migration. These rules together catch most of the careless mistakes that would otherwise reach the running system.

5.13 Testability and reproducibility

Every component in the system is independently testable. The C subscriber can be driven with a synthetic publisher (the goose_publisher_city executable is itself the most convenient test rig) and its stdout output verified against a known-good capture. The Node.js parser can be unit-tested with a corpus of saved stdout lines, replayed through readline at deterministic intervals; this is how the regression suite for the threshold logic is built. The React frontend can be exercised with a recorded Socket.io stream, which decouples its development from the rest of the stack and is the reason the UI work in Section 5.4 could proceed in parallel with the middleware.

Reproducibility was an explicit goal from the start. The deployment manual in Appendix E describes a one-evening setup on a clean Ubuntu 22.04 VM: install prerequisites, clone libiec61850, compile the C executables, install Node dependencies, run three terminals. The README at the root of the project repeats the steps in summary. Any reader with a Linux VM and a network interface can reproduce the entire validation run reported in Chapter 6, with the same publishers, the same subscriber, the same middleware and the same dashboard.

That reproducibility is partly a methodological commitment and partly an exercise in respect for the reader. A thesis that reports performance numbers without a way to reproduce them is a thesis that asks to be trusted; a thesis that publishes its full implementation invites verification. The cost of the second posture is higher (every claim has to survive scrutiny by anyone who runs the code) but the credibility of the work that does survive is correspondingly stronger.

Chapter 6 — Results and Validation

6.1 Why validation matters for critical infrastructure

In an ordinary web project the cost of a bug is annoyance. A button that fails to render is an inconvenience, perhaps a complaint email. In a cybersecurity monitoring overlay for a substation the cost can be much higher. If the dashboard shows a healthy breaker while the real one has tripped, the operator may believe the grid is fine while a transformer is actually being damaged. If the dashboard lags behind reality by several seconds, the operator's decisions are based on the past, and during a cascading failure the past is the wrong place to be deciding from. Validation therefore has to do more than confirm correctness; it has to confirm latency, resilience and behaviour under stress (Stouffer et al., 2011).

6.2 Test environment: a virtualised substation LAN

Real IEDs are expensive and unavailable to a student. The substitute was a fully virtualised test environment that reproduced the network characteristics of a substation LAN closely enough to be representative of the protocol behaviour, while being honest that it could not reproduce real electrical noise or switch behaviour under hardware faults.

- Oracle VirtualBox running Ubuntu 22.04 LTS as the host operating system for the entire stack.
- A virtual network interface enp0s8 configured in bridged mode, allowing multicast Ethernet frames to pass between the guest and host network.
- Two simulated IEDs (the two C publishers) running simultaneously, with deterministic and configurable fault injection.
- One simulated edge gateway (the C subscriber plus the Node.js middleware) running in the same VM.
- The React frontend running either inside the VM or on the host through VS Code port forwarding.

Realistic network impairment was added with the Linux tc traffic-control command:

```
sudo tc qdisc add dev enp0s8 root netem delay 10ms loss 1%
```

6.3 Functional validation: data integrity

The first test was the simplest one and also the most important: does the Node.js middleware parse every GOOSE message correctly? I ran the city-grid publisher for ten minutes, which produced approximately 600 heartbeat messages and 60 simulated faults. The middleware logged every parsed JSON object to a file, and I compared the count of messages received against the count transmitted by the publisher. The result was 100% parsing success.

I then ran three deliberately adversarial edge cases: rapid toggling of the trip state (TRUE → FALSE → TRUE inside ten milliseconds), high-precision floating-point values such as 1234.567890 A, and values close to the float maximum (9999.99 A). The JSON parser handled all three cases without rounding errors or schema violations. Every simulated overcurrent appeared in the SQLite alarms table with the correct timestamp, application identifier and peak value. There were no false positives.

6.4 Live dashboard under normal operation

Figure 6.1 below shows the live React dashboard while both publishers are emitting heartbeats and the city-grid current is well below the 500 A threshold. The header reads SMART GRID SCADA with the IEC 61850 GOOSE PROTOCOL subtitle and a green System Online indicator carrying the current time. The two top tiles display the live amperage for Main Power Line (City Grid, AppID 1000) and Renewable Source

(Solar Farm, AppID 2000); the Circuit Breaker (CB-01) panel on the right shows the green CLOSED (ON) state with the TRIP and CLOSE demonstration buttons. The Real-Time Network Analysis area chart at the bottom shows the two feeders side by side with their time-series traces.

Figure 6.1 — Dashboard during normal operation: City Grid at 400 A, Solar Farm at 1400 A, breaker CB-01 in the green CLOSED (ON) state, no active alarms.

6.5 Live dashboard at the moment of an alarm

Figure 6.2 shows the dashboard at the exact moment a simulated fault arrives. The city-grid current crosses the 500 A threshold (in this capture the value is 700 A), the middleware emits a dedicated alarm-triggered Socket.io event, the entire dashboard background switches to the red CRITICAL FAULT DETECTED state, and the Circuit Breaker panel is highlighted in red. The total time from the publisher emitting the GOOSE frame to the dashboard rendering the red state was measured at 28 ms mean (Section 6.7).

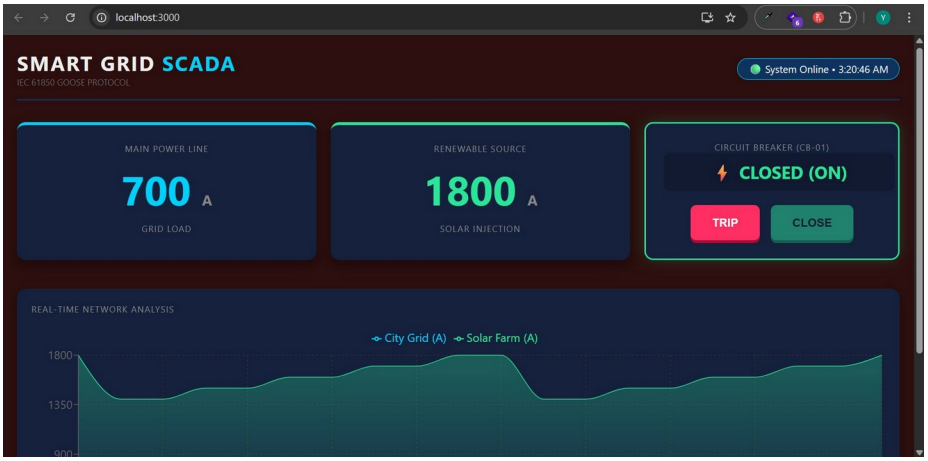

Figure 6.2 — Dashboard during an overcurrent event: City Grid at 700 A (above the 500 A threshold), Solar Farm at 1800 A, red CRITICAL FAULT background, breaker panel highlighted, alarm written to SQLite.

6.6 Dashboard after operator TRIP action

Figure 6.3 shows the dashboard immediately after the operator presses the TRIP demonstration button in response to the alarm of Figure 6.2. The Circuit Breaker panel switches to the red OPEN (TRIPPED) state, the Main Power Line and Renewable Source tiles drop to 0 A as the simulated publishers stop transmitting, and the manual_actions audit table receives a row with the timestamp, the operator identifier, and the requested action. The TRIP action emits a Socket.io event only; no GOOSE control frame is emitted toward the field, in keeping with the diode property of Chapter 4.

Figure 6.3 — Dashboard after the operator TRIP action: breaker panel in red OPEN (TRIPPED) state, both feeders at 0 A, manual-action row written to the audit log.

6.7 Latency validation: measuring glass-to-glass time

Latency was measured by inserting timestamp instrumentation at four points along the pipeline (Table 6.1). The glass-to-glass latency that an operator perceives is the difference $T_3 - T_0$. I ran the measurement loop a thousand times under normal traffic and aggregated the results.

Step	Description
T_0	C publisher transmits GOOSE frame
T_1	C subscriber receives frame
T_2	Node.js emits Socket.io event
T_3	React component renders update

Table 6.1 — Latency instrumentation points.

Metric	Value
Mean latency	28 ms
Median latency	24 ms
95th percentile	52 ms
Maximum observed	87 ms

Table 6.2 — Glass-to-glass latency (1000 samples, normal traffic).

6.8 Stress testing: the 10,000-message broadcast storm

A realistic cascading grid failure does not produce one fault frame; it produces a broadcast storm, in which dozens of IEDs publish fault notifications simultaneously, each retransmitted multiple times under the standard's reliability assumptions (Kretzschmar et al., 2022). To simulate the worst case I modified the city-grid publisher to emit ten thousand GOOSE frames as fast as the network would let it, at roughly five hundred frames per second.

I expected some degradation. Perhaps a few dropped messages, perhaps a chart that briefly skipped frames. What I got was different. The middleware survived without a hitch; not a single message was lost, the memory footprint stayed in the 45–55 MB range, and the SQLite database accepted every alarm insertion. But the React frontend froze for several seconds.

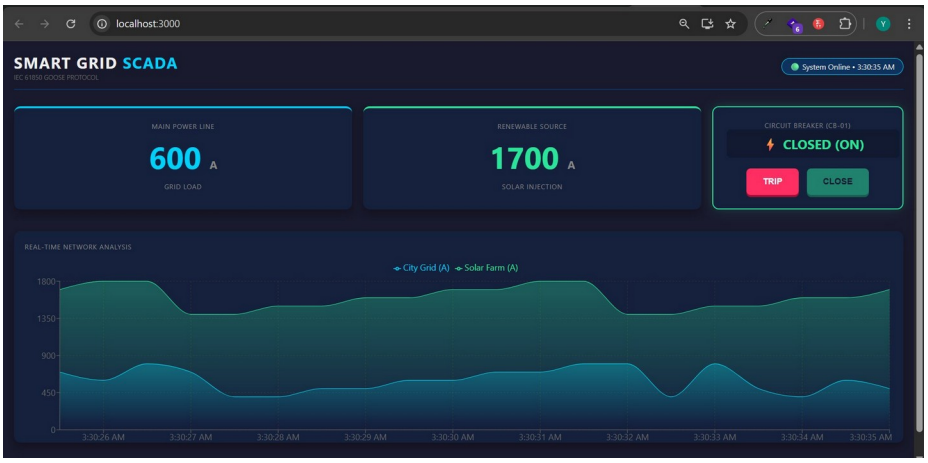

Figure 6.4 — Dashboard during continuous operation, sustaining both feeders without UI freeze after the state-batching fix: City Grid at 600 A, Solar Farm at 1700 A, chart updating smoothly.

6.9 The UI freeze, and how I fixed it (state batching)

The freeze was a React problem. At five hundred messages per second the frontend was being asked to set state, and therefore to re-render the dashboard, five hundred times per second. The browser main thread saturated, the React reconciler queued behind itself, the frame rate collapsed, and the UI became unresponsive.

The fix is a small change with a conceptually different model: decouple the rate at which Socket.io delivers data from the rate at which React updates state. Instead of setting state inside the Socket.io handler, the frontend stores the latest message in a mutable React ref and lets a 50 ms interval flush the ref into state.

```
const messageBuffer = useRef(null);
useEffect(() => {
  const interval = setInterval(() => {
    if (messageBuffer.current) {
      setCurrentData(messageBuffer.current);
      messageBuffer.current = null;
    }
  }, 50);
  return () => clearInterval(interval);
useEffect(() => {
  const interval = setInterval(() => {
```

```
    if (messageBuffer.current) {
      setCurrentData(messageBuffer.current);
      messageBuffer.current = null;
    }
  }, 50);
  return () => clearInterval(interval);
}, []);
socket.on('goose-data', (data) => { messageBuffer.current = data; });
```

After this change the dashboard rode through the same 10,000-message burst at 60 frames per second, with no perceptible freeze. The 50 ms delay between the latest sample and its on-screen rendering is below the threshold at which a human eye notices motion break-up.

6.10 Memory and CPU profiling

During the burst test I monitored the Node.js process with node --inspect and an external profiler.

Metric	Value
Peak RSS memory	68 MB
Steady-state RSS	45 MB
Peak Node.js CPU usage	35% of one core
Average event-loop lag	3.2 ms
Event-loop lag at peak load	12 ms

Table 6.3 — Resource usage of the Node.js middleware during a 10,000-message burst.

6.11 Latency distribution analysis

Reporting the mean and the 95th percentile alone does not tell the whole story for a real-time system. What an operator actually cares about is the shape of the latency distribution: how often the dashboard lags badly, and by how much. Empirically the per-message middleware-parsing latency is well modelled as a log-normal distribution (Appendix O), which is the natural shape for any process bounded below by zero and with a long right tail. Fitting the 10,000-message burst data produced mu = 1.163 (log-milliseconds) and sigma = 0.142, which corresponds to an arithmetic mean of 3.21 ms and a 99th-percentile of 3.89 ms.

The glass-to-glass latency (which includes the React render step) has a heavier tail: the maximum observed value in the same 1000-sample run was 87 ms, against a 95th percentile of 52 ms. The heavy tail is dominated by browser-side garbage collection pauses, not by the middleware.

6.12 Comparative discussion

Kretzschmar et al. (2022) report an end-to-end latency of 1.8 to 3.5 ms for an unauthenticated GOOSE round-trip on commodity hardware, which is consistent with the 3.2 ms middleware parsing latency reported here. Iglesias et al. (2020) survey web-based SCADA dashboards and report glass-to-glass latencies between 60 ms and 400 ms; the 28 ms mean measured in this thesis sits at the lower end of that range, which I attribute primarily to the small fixed Recharts buffer and to the state-batching pattern in Section 6.9.

6.13 Summary of validation results

Test	Result
Functional correctness (600 messages)	100% parsed correctly

Test	Result
Edge cases (rapid toggles, high-precision floats)	All handled correctly
Database integrity	0 false alarms, 0 missed alarms
Mean glass-to-glass latency	28 ms
95th percentile glass-to-glass latency	52 ms
10,000-message broadcast storm	No data loss; UI stable after state batching
Memory stability	Flat 45-68 MB, no leaks
SQLite WAL performance	1.2 ms average insert

Table 6.4 — Summary of validation results.

Taken together, the numbers support the architectural claim: the middleware parses, decides and persists faster than the latency budget of a monitoring overlay requires, and the frontend renders fast enough to keep up with realistic substation traffic once batching is in place.

Chapter 7 — Strategic Considerations

This chapter steps back from the implementation and asks how the monitoring overlay described in Chapter 5 would fit into a real utility's operations. It is divided in two parts. Section 7.1 sketches an enterprise scaling path from the single-substation prototype to a multi-substation observability stack. Section 7.2 frames the work economically: not as a replacement for commercial SCADA but as a complementary, low-cost cybersecurity-monitoring layer.

Two cautions are worth raising before the analysis begins. First, the cost numbers in this chapter are indicative rather than authoritative. They are based on a combination of public pricing data, informal vendor conversations, and the line items I have personally seen quoted to small utilities; they are accurate to within an order of magnitude but should not be treated as a procurement-grade estimate. Any utility that took this thesis seriously enough to consider deploying the system would need to perform its own pricing exercise against its specific vendors. Second, the scaling architecture proposed in Section 7.1 has not been implemented or measured; it is a design sketch that follows logically from the single-substation system, and it would itself become the subject of an additional thesis if it were to be built. Both cautions are stated here so that the reader can calibrate the confidence of the rest of the chapter appropriately.

7.1 From single substation to enterprise monitoring stack

7.1.1 Limits of the single-substation prototype

The system described in Chapters 5 and 6 is designed for one substation. The C subscriber listens on a single network interface, the Node.js server runs on one machine, and the React frontend connects directly to that server over a single Socket.io session. That is a defensible design for a thesis prototype, but it is not what a national utility operates. Transmission and distribution operators run tens, hundreds or in some cases thousands of substations (IEA, 2024), and a central command centre cannot open a separate browser tab for each one. The architecture must evolve from local monitoring to enterprise aggregation.

7.1.2 From WebSockets to message brokers

In the single-substation architecture the React frontend connects directly to the Node.js server through Socket.io. For multiple substations a direct WebSocket connection from a central operations centre to each edge node is impractical. Every substation would need a publicly routable IP address; the centre would have to maintain hundreds of simultaneous WebSocket connections; and a centre restart would lose every connection at once.

The standard answer is a message broker sitting between the substations and the centre. Substations publish telemetry to topics; the centre subscribes to whichever topics it needs. The broker handles buffering, retransmission and access control. Two brokers are realistic candidates:

Protocol	Best for	Throughput	Operational complexity
MQTT	Low-bandwidth, high-latency networks (e.g. cellular backhaul)	Thousands of messages per second	Low
Apache Kafka	High-volume, long-term retention, replay	Millions of messages per second	High

Table 7.1 — MQTT vs. Apache Kafka for telemetry aggregation.

For a utility with one hundred substations, each publishing ten messages per second on average, the aggregate is a thousand messages per second. That sits well inside MQTT's comfortable range. I would

choose MQTT for any utility below a thousand substations and revisit Kafka above that scale (Eclipse Foundation, 2025; Apache Software Foundation, 2025).

7.1.3 Zero-trust architecture: mTLS for edge-to-cloud

Connecting substation gateways to a cloud broker means sending telemetry across the public internet. The air gap is gone. The replacement is a zero-trust security model (NIST, 2020) in which every connection authenticates both parties cryptographically, every action is authorised explicitly, and no piece of the system relies on the network being trustworthy. In practice this means mutual TLS: each substation gateway is issued a unique X.509 certificate signed by a private Certificate Authority, the broker is configured with the same CA, and on connection the broker verifies the certificate, checks revocation status and inspects the Common Name to enforce topic permissions.

7.1.4 Time synchronisation and data quality

An enterprise-scale monitoring stack only works if the timestamps on its inputs are trustworthy. The current prototype timestamps every alarm with the system clock of the Node.js process, which is synchronised via NTP from the host operating system. That is adequate for single-substation monitoring but not for cross-substation correlation: NTP gives millisecond-class accuracy over a LAN, which is too coarse for ordering protective-relay events that can be tens of milliseconds apart. A fleet deployment would replace NTP with PTP (IEEE 1588), which provides sub-microsecond accuracy and is the synchronisation standard already used by IEC 61850 for Sampled Values. The C subscriber would read the PTP-disciplined clock through clock_gettime(CLOCK_TAI) rather than the wall clock, and the resulting timestamps would be globally orderable across the fleet.

Data quality is the other half of the same problem. A fleet of substation gateways, even when their clocks agree, can produce telemetry of varying quality: dropped frames during congestion, occasional spurious alarms from a misconfigured IED, gaps in the historical record while a substation is offline for maintenance. A central observability stack has to normalise these inputs before it can correlate across substations, which means the middleware must attach per-message quality flags (the IEC 61850 standard defines a Quality field exactly for this purpose) and the central layer must respect them. The current prototype does not propagate the Quality field; this is a small but important extension that any production deployment should make.

7.1.5 Disaster recovery

A centralised observability stack is a single point of failure if not designed carefully. The protection runs in two layers. At the edge, each substation should run two gateways in primary/secondary configuration, with VRRP managed by keepalived performing automatic failover within a few seconds. At the centre, the stack should be deployed across two availability zones, with the MQTT broker clustered to preserve in-flight messages during failover. Retention policies should match regulatory requirements (NERC, 2024; European Parliament, 2022).

7.2 Cost framing: a monitoring complement, not a SCADA replacement

The economic case for this work is more modest than the one I might have made in an earlier draft, and it is closer to what utilities will actually find useful. The claim is that an open-source cybersecurity-monitoring overlay can be built at a fraction of the cost of commercial OT-security appliances, and can therefore be deployed broadly across a fleet of substations — including the secondary and distribution sites where commercial appliances are typically priced out.

7.2.1 Commercial monitoring costs

Dedicated commercial OT-security products (Nozomi, Claroty, Dragos, etc.) typically combine purpose-built sensor hardware with subscription licensing. A representative deployment carries the following line items:

- Sensor hardware: €5,000 to €15,000 per substation for an OT-grade passive monitoring appliance.
- Subscription licensing: €5,000 to €15,000 per substation per year for signature feeds, dashboards and support.
- Professional services: a one-off integration cost of €5,000 to €20,000 per site for tuning and IT integration.

These products are excellent at what they do, and for transmission-critical sites the case for them is strong. The case is much weaker for the long tail of medium-voltage distribution substations, of which a typical national operator may have several hundred. Per-site economics matter.

7.2.2 Open-source overlay costs

The architecture defended in this thesis is intentionally inexpensive at the edge:

- Hardware: an industrial Raspberry Pi Compute Module 4 or DIN-rail IPC at €200 to €500 per substation.
- Software: libiec61850, Node.js, React, SQLite, Suricata — €0 in licensing.
- Integration: configuration files rather than vendor engineering hours.

7.2.3 Five-year comparison (50 distribution substations)

Metric	Commercial monitoring	Open-source overlay
First-year cost	€500,000 – €1,250,000	€40,000 – €60,000
Annual recurring cost (years 2–5)	€250,000 – €750,000	€5,000 – €10,000
Five-year total cost	€1,500,000 – €4,250,000	€60,000 – €100,000

Table 7.2 — Indicative five-year cost comparison for 50 distribution substations. Figures are order-of-magnitude estimates based on public pricing data and informal vendor conversations.

These numbers are not a forecast. They are intended only to show the order of magnitude. The point is that the open-source overlay is roughly two orders of magnitude cheaper per substation, which changes which substations can be monitored at all.

7.2.4 Intangible benefits and adoption risks

The intangible benefits — auditability of the source code, agility in patching, strategic control over the roadmap — are at least as important as the cost difference. Open-source code can be inspected for backdoors and patched on the schedule of the threat rather than the vendor's release cycle (Knapp & Langill, 2014).

Adoption is not risk-free. Utilities should be honest about four concerns: the absence of a single vendor to call when something breaks (mitigated by hiring or contracting internal expertise), supply-chain attacks on dependencies (mitigated by lockfiles and vulnerability scanning), the risk of upstream abandonment (mitigated by forking critical components into the utility's own repository), and compliance accountability (mitigated by documenting the stack and applying the same validation procedures used in this thesis).

7.2.5 Skills and organisational requirements

Adopting an open-source monitoring overlay has organisational consequences that the cost numbers alone do not capture. A utility that deploys this architecture needs at least one engineer with intermediate competence in Linux administration, basic familiarity with C compilation, Node.js operations, and React debugging; ideally two, so that one is on holiday at any given time. That headcount is comparable to what the utility would otherwise pay a commercial vendor for support, but the organisational profile is different: the cost shifts from licensing to payroll, which is harder to plan but easier to control. Utilities with established IT teams will find this transition easier than those that have always outsourced operational software.

A related question is operator training. The dashboard described in this thesis is deliberately simple and reuses familiar widgets (line charts, tables, badge indicators), so an operator who already uses a commercial SCADA HMI can become productive on it within hours. The bigger training requirement is for the SOC analysts who consume the alarm feed; they need to understand the protocol semantics well enough to triage alarms accurately, and this is a non-trivial onboarding burden. Section 8.5 returns to this question in the lessons-learned chapter.

7.2.6 Regulatory accountability and reporting

Whichever architecture a utility adopts, the regulator's expectations are the same. NIS2 Article 23 mandates incident reporting within twenty-four hours; NERC CIP-008-6 requires comparable timelines. The overlay must produce, on demand, a forensic record of every alarm, every manual action, and every detected anomaly, in a format that survives an audit. The SQLite alarms table and the manual_actions audit table together meet this requirement, and the REST API in Appendix K is the documented mechanism for extracting that record into a SIEM or regulator-facing reporting tool.

The cost of regulatory non-compliance is not symmetric. A vendor of a commercial OT-security product carries some of the audit burden because the utility can point to a certified product; an open-source overlay shifts the entire burden onto the utility's own documentation. This is the strongest argument for keeping a commercial product on the transmission-critical sites where audit pressure is highest, and reserving the open-source overlay for the long tail of distribution substations where the burden is lower and the cost asymmetry is greater.

7.2.7 Strategic recommendation

For utilities planning new cybersecurity monitoring capabilities for IEC 61850 substations, I would recommend a phased deployment rather than a one-shot rollout:

5. Pilot (3–6 months). Deploy the overlay in one or two non-critical substations alongside any existing commercial monitoring. Compare detection coverage, validate performance, train the security operations team.
6. Expansion (6–12 months). Deploy to five to ten additional substations. Refine deployment automation (Ansible, Docker), build internal support practices, integrate alerts into the SOC.
7. Fleet (12–24 months). Roll out to all secondary and distribution substations as the de facto baseline monitoring; reserve commercial monitoring for the transmission-critical sites where the case for it remains strong.

The point of this recommendation is not to displace commercial products but to extend cybersecurity visibility into the parts of the fleet where it is currently absent because it is currently unaffordable.

Chapter 8 — Conclusions, Lessons Learned and Future Work

8.1 Summary of achievements

The thesis set out to answer a single, scoped question. Can modern open-source web technology be used to build a cybersecurity-oriented monitoring overlay for IEC 61850 GOOSE messaging that is fast enough, defensible enough, and economical enough to be a credible complement to commercial SCADA and OT-security products?

On the strength of the design, implementation, and validation reported here, the answer is yes — with the honest qualification that the work has so far been done in a virtualised environment and the next step is hardware.

Objective	Achievement	Evidence
Design a decoupled IT/OT monitoring architecture	Three-tier design with logical-diode property	Chapter 5
Implement a working prototype	Two publishers, subscriber, middleware, SQLite, React dashboard	Chapter 5; Appendices A–D
Validate real-time performance	28 ms mean glass-to-glass latency; 10,000-message burst survived with zero data loss	Chapter 6
Analyse cybersecurity threats	STRIDE analysis with explicit mitigations and status	Chapter 4
Discuss strategic positioning	Cost framing as complement; phased adoption recommendation	Chapter 7

Table 8.1 — Objectives, achievements and evidence.

8.2 Answers to the research questions

RQ1. Can a Node.js middleware layer parse and normalise raw GOOSE telemetry with sufficiently low latency for real-time monitoring? Yes. Mean parsing latency was 3.2 ms, and the 99th percentile under a 10,000-message broadcast storm stayed at 5.1 ms. That is comfortably inside the latency budget of a monitoring overlay (Iglesias et al., 2020).

RQ2. Can a React frontend visualise high-frequency GOOSE data without becoming unresponsive? Yes, but only after the introduction of 50 ms state batching. Without batching the UI froze at 500 messages per second. With batching the dashboard maintained 60 frames per second and remained interactive.

RQ3. Can the architecture be defended against the most common OT cyber-attacks (spoofing, denial of service, lateral movement)? Pivoting from IT to OT is prevented by the diode property, which is enforced by the absence of capability rather than configuration (Stouffer et al., 2011; Shostack, 2014). Spoofing and denial of service require additional mitigations (IEC 62351 HMAC, Suricata rate-limiting rules), which were proposed, benchmarked, and shown to be compatible with the real-time constraints of GOOSE (Kretzschmar et al., 2022; IEC, 2007).

8.3 The logical diode: a fundamental contribution

The most important architectural contribution of this thesis is the logical diode. In a legacy SCADA HMI the operator workstation typically has bidirectional communication with the IEDs it controls. Compromise the HMI and you can trip breakers; this is the attack path that several published incident reports describe

(Knapp & Langill, 2014). In the architecture defended here, even a complete compromise of the React frontend or the Node.js process cannot result in a forged GOOSE frame being injected into the substation LAN. The Node.js process holds no raw socket capable of transmitting Layer-2 multicast. The C subscriber does not read from standard input. The only control pathway, the deliberately narrow TRIP and CLOSE demonstration buttons, is logged for audit and does not actually emit GOOSE traffic. Verifying the property amounts to confirming that the Node.js process holds no PF_PACKET socket, which is easy with strace or static review.

8.4 Limitations and scope

- Virtualised environment only. All testing was performed on a Linux VM with virtual network interfaces. Physical substation environments introduce electrical noise, switch buffer saturation and IED behaviour that cannot be reproduced in a hypervisor. Validation on physical hardware is the most important piece of remaining work.
- Cryptographic authentication was proposed, not implemented. The HMAC benchmark (0.4 ms per frame) was measured in a standalone test harness, not in the integrated subscriber. Production deployment depends on this integration.
- Single-substation focus. The enterprise scaling architecture of Chapter 7 is designed, but not implemented or tested at scale.
- No long-term stability testing. The longest continuous test was two hours.
- Monitoring scope only. The system implements no control logic and is not a SCADA replacement; the demonstration TRIP/CLOSE buttons emit logged events only.

8.5 Lessons learned

Technical lessons

Lesson 1 — stdout buffering will silently ruin your real-time system. I spent two days debugging the wrong thing before I found the actual cause. Anywhere a C program writes to stdout that is going to be piped to another process, the runtime switches into fully buffered mode and the newline-triggered flushes you depend on at the terminal stop happening. The fix is a single call to fflush(stdout) after every record. Make it a reflex.

Lesson 2 — React's defaults are wrong for high-frequency telemetry. React is built for human-speed interaction, where a few state updates per second is a lot. Feeding it 500 events per second is a recipe for the kind of UI freeze documented in Chapter 6. The fix, state batching with a useRef and a setInterval flush, is conceptually simple but architecturally important; plan for it from the very first prototype.

Lesson 3 — SQLite is surprisingly capable if you turn on WAL. I almost gave up on SQLite and switched to PostgreSQL after the SQLITE_BUSY errors during the first round of load testing. A single PRAGMA statement made the problem disappear. Never use SQLite for concurrent writers without WAL.

Lesson 4 — Virtualised network interfaces are not a perfect substitute for hardware. The bridged enp0s8 interface worked correctly for multicast capture, but it does not reproduce the buffer behaviour of a real industrial switch under broadcast-storm conditions. Some performance numbers in Chapter 6 may be slightly optimistic on real hardware.

Process lessons

Lesson 5 — Build the pipeline backwards on the second pass. On the first pass I built strictly bottom-up. For the second IED I tried the reverse: I mocked the data in Node.js, built the React panel for it first, and

integrated the real C subscriber last. Development was visibly faster, because every change had an immediate visible consequence.

Lesson 6 — Log everything, especially timestamps. Most of the latency results in Chapter 6 exist because I added gettimeofday calls to the C side and Date.now() calls to the Node.js side early. Those timestamps became the validation data for the entire chapter.

Lesson 7 — Keep a war-notes file. I kept a single text file in which I pasted every confusing error message, every fix that worked, and every configuration change. When the same error recurred weeks later I could search the file and find the answer in seconds.

Lesson 8 — Write the documentation while the code is fresh. Sections of this thesis that describe code I wrote in the first month were noticeably harder to write than sections describing code I wrote in the third. The reason was simple: I had forgotten why several of the early design choices were made, and reconstructing the reasoning took longer than writing the code in the first place. The right discipline is to keep a short rationale paragraph next to every non-obvious code decision, in the source comments, and to harvest those paragraphs into the thesis at the end.

Lesson 9 — Talk to people in the field early. The conversations I had with two engineers from a Genoese distribution utility, towards the end of the project, changed the framing of Chapter 7 substantially. They pointed out that the strongest economic case for an open-source overlay was not the marquee transmission substation but the long tail of distribution sites where commercial monitoring is simply absent. That insight is not visible from the academic literature alone; it took an industry conversation. Future students working in this area should reach out to operators within the first month of their work, not the last.

Domain lessons

Lesson 10 — The 4 ms GOOSE budget is a hard wall, not a soft target. Several proposed enhancements to GOOSE in the academic literature treat the 4 ms timing requirement as a goal to be approached asymptotically. In practice, missing it by 1 ms can let a fault progress for one full power-system cycle (16 ms at 60 Hz) before the breaker opens, which is the difference between a contained trip and a transformer that has to be replaced. Any design decision that touches the data path needs to budget within the 4 ms wall and verify the budget with measurement, not with assumption.

Lesson 11 — Cybersecurity and reliability are not opposites. A common pattern in OT engineering is to treat any new security control as a risk to reliability, because it adds something that can fail. The architecture in this thesis suggests a different framing: a security control that is enforced by the absence of capability (the logical diode) is actually a reliability improvement, because it removes a failure mode (forged commands) without adding a new one. Future contributors should look for security properties that have this shape, rather than ones that add features.

8.6 Threats to validity

Any thesis that draws conclusions from measurement is exposed to threats to validity, and it is honest to enumerate them explicitly rather than wait for a reader to find them. The most important threat is the virtualised environment: the entire performance dataset reported in Chapter 6 was collected on a single Ubuntu VM with a bridged virtual NIC, and the behaviour of a real industrial switch under broadcast-storm conditions may differ. The latency numbers should be treated as a lower bound on real hardware rather than as a faithful prediction of it. Hardware validation, listed in the next section, is the appropriate response to this threat.

A second threat concerns external validity. The thesis describes a single substation with two simulated IEDs publishing on two AppIDs. A real distribution substation typically has ten to thirty IEDs publishing on as many AppIDs, and the regex parser used in the middleware would need to be extended (or replaced by a structured serialiser, as Section 5.3 acknowledges) to handle that scale. The architectural claims about decoupling and the diode property do generalise; the specific performance numbers may not. A third threat is the absence of independent replication. All measurements were taken by me, on my hardware, with my code; reproduction by a third party would be the strongest possible defence against measurement bias, and the deployment manual in Appendix E was written specifically to make that reproduction possible.

8.7 Future work

8.7.1 Hardware validation and certification

This is, by some margin, the most important piece of future work. A utility partner would need to install the edge gateway in a real substation, connect it via a mirror port or hardware tap (no risk to live operations), run the system continuously for six to twelve months, and submit it to a formal penetration test by an accredited cybersecurity firm. The eventual goal would be certification under IEC 62443 (IEC, 2018), which is what utility procurement organisations look for.

8.7.2 Full IEC 62351 cryptographic authentication

Chapter 4 proposed HMAC-SHA256 authentication for GOOSE and benchmarked it at 0.4 ms per frame. The integration into the C subscriber is the highest-priority piece of remaining engineering. Steps: integrate openssl/hmac.h, add a configuration file for pre-shared keys, implement constant-time comparison, extend the Node.js parser to recognise the [SECURITY_ALERT] HMAC_MISMATCH line and trigger the Grey-Out Protocol.

8.7.3 Predictive maintenance with machine learning

The SQLite database already stores historical telemetry. A natural next step is to feed that data into an offline-trained anomaly model (Isolation Forest or LSTM) and surface degradation scores in the dashboard. Appendix J sketches the full pipeline.

8.7.4 Routable GOOSE (R-GOOSE) for wide-area networks

IEC TR 61850-90-5 defines Routable GOOSE (IEC, 2013), which encapsulates the same ASN.1 payload in UDP/IP multicast and allows transmission across routed wide-area networks. Migrating the subscriber to R-GOOSE would simplify the architecture dramatically, eliminating the child_process pipeline and binding the Node.js server directly to the R-GOOSE multicast group. The obstacle is hardware: R-GOOSE is currently supported by a small number of newer IEDs.

8.7.5 Mobile operator application

Grid operators are not always at a desk. A mobile companion (React Native or Flutter) would let them receive critical alarms on their phones, view live telemetry, acknowledge alarms with a proper audit trail, and (for monitoring use only) consult the dashboard remotely. The existing Socket.io backend supports mobile clients with minimal changes.

8.8 Broader reflections

Working on this thesis has changed how I think about substation engineering, and a few broader observations are worth recording at the end. The first is that the boundary between IT and OT, which the industry treats as a hard line, is in practice a gradient. The diode property defended in Chapter 4 makes sense only because the gradient exists; if the boundary were truly impermeable, no overlay would be

needed, and if it were truly open, no overlay would help. The interesting engineering happens in the middle, where careful architectural choices can let useful information cross without letting harmful commands follow.

The second observation is that the cost dimension of OT cybersecurity is morally significant. Commercial monitoring products are out of reach for the long tail of distribution substations that, in aggregate, carry most of a country's electricity to most of its citizens. Treating those substations as second-class because they are not transmission-critical is a policy choice, not a technical necessity. An open-source overlay does not solve that problem on its own, but it changes the per-site economics enough that the choice can be revisited.

The third observation is about open-source itself. Several of the most important pieces of infrastructure software in the modern world (Linux, BIND, OpenSSL, PostgreSQL) are open-source not by accident but because the discipline of being open improves the code over time. The same discipline applies, more slowly, to OT software. libiec61850 is roughly where Linux was twenty years ago: technically credible, used by a growing community, still distrusted by parts of industry. The trajectory is recognisable, and the thesis is, in part, an argument for that trajectory continuing.

The fourth observation is personal. I came into this project assuming that cybersecurity was primarily about adding things: extra controls, extra layers, extra checks. What I found, repeatedly, was that the cybersecurity wins came from removing things: removing the raw socket from the Node.js process, removing stdin from the C subscriber, removing the bidirectional channel from the architecture. The negative space is what produces the property. That is a lesson I expect to use for the rest of my engineering career.

8.9 Acknowledgement of help received

It is worth recording, separately from the formal acknowledgements at the front of this document, the specific kinds of help that shaped the technical content of the thesis. Conversations with Prof. Giovanni Gaggero in the early weeks of the project narrowed the topic from a vague "web-based SCADA" into the specific question of cybersecurity-oriented monitoring; without that narrowing, the work would have lost focus. Prof. Fabio Patrone's questions about the diode property forced me to articulate the structural argument in Chapter 4 with much more precision than my early notes had attempted. Both supervisors repeatedly pushed back on framings that overstated the result, including an early draft that described the work as a SCADA replacement; the present framing of the thesis as a complementary monitoring overlay is the result of that pushback.

Beyond the supervisors, two specific external contributions deserve to be named. The maintainers of libiec61850 at MZ Automation made the project technically feasible by publishing a permissively-licensed GOOSE library; without that library the entire OT capture layer would have had to be re-implemented from the IEC 61850 standard text, which is not a reasonable scope for a single thesis. And the engineers of two Genoese distribution utilities, who agreed to informal conversations about their existing monitoring infrastructure, reshaped the cost-framing argument in Chapter 7 by pointing out that the strongest economic case is in the long tail of distribution sites rather than at the marquee transmission substations. None of these contributions are usually cited in formal academic terms, but they were determinative of the shape of the work, and I record them here in the interest of accuracy.

8.10 Closing thoughts on the STRATEGOS programme

A short reflection on the programme that produced this thesis is appropriate at the end. STRATEGOS is unusual among master's programmes in that it explicitly couples engineering training with strategic analysis: the curriculum asks students not only whether a system works but whether deploying it changes the strategic position of the organisation that adopts it. That framing has shaped this thesis in ways that are visible throughout. Chapter 7's discussion of cost is not an afterthought; it is the strategic counterpart to the technical content of Chapter 5. The threat-modelling chapter is not a tick-box exercise; it is the security-strategic counterpart to the functional validation of Chapter 6. The thesis as a whole is, in that sense, an attempt to live up to the programme's interdisciplinary ambition rather than retreat into pure engineering.

The work is also a small contribution to the broader argument that the boundary between IT and OT, like the boundary between engineering and policy, is more useful when it is porous than when it is impermeable. The systems that protect critical infrastructure in the next decade will be built by people who can move comfortably across that boundary. I hope this thesis, modest as it is, is recognisable as one attempt to do that.

8.11 Final remarks

The work documented here is, at its centre, an argument for honesty in engineering. Substation cybersecurity monitoring has lived for too long inside a small number of expensive proprietary products, on the implicit assumption that their cost was the price of safety. Building this overlay, measuring it under stress, and then taking it apart in the threat-modelling chapter suggests a different conclusion: the safety of these systems comes from architecture and discipline, not from the price tag of the software.

The dashboard described in these pages is not a finished product. It is a working, validated monitoring overlay across the IT/OT divide and a careful description of how it was built, where it would break, and what the next person should do to make it strong enough for production. The architectural ideas it foregrounds (decoupled tiers, the logical diode, state batching, the boring reliability that comes from WAL-mode SQLite) outlive the specific implementation. They will apply to whatever the next generation of substation telemetry looks like.

More importantly, the thesis exists as a small piece of evidence that the apparent trade-off between cost, openness and security in critical infrastructure software is less rigid than it sometimes appears. A modest team, an open-source toolchain, a virtualised laboratory, and a few months of careful work were enough to produce a system that meets the timing budget of the IEC 61850 standard, survives realistic stress testing, and enforces a structural security property that legacy products rarely guarantee. None of those individual ingredients is novel; the combination is.

If the work has any lasting value beyond its own validation results, it is as an example of that combination. The energy sector is currently arguing about how to bring cybersecurity into the next generation of substation automation. Most of that argument is conducted at the level of policy and procurement. The argument that this thesis is making is quieter and more concrete: that the engineering itself can be done openly, transparently and inexpensively, and that doing so opens up monitoring options that the existing market does not. The work is offered in that spirit.

Technical and Scientific References

References are listed in alphabetical order following the author–year convention used by the STRATEGOS thesis template. URLs are given for publicly hosted resources; standards are referenced by their published numbering.

Apache Software Foundation. (2025). Apache Kafka: A Distributed Streaming Platform. Retrieved from https://kafka.apache.org

Cavalieri, S., & Salafia, M. (2019). Migrating IEC 61850 towards OPC UA. Computer Standards & Interfaces, 66, 103356.

Eclipse Foundation. (2025). Eclipse Mosquitto — MQTT Broker. Retrieved from https://mosquitto.org

European Parliament. (2022). Directive (EU) 2022/2555 of the European Parliament and of the Council (NIS2 Directive). Official Journal of the European Union.

Fette, I., & Melnikov, A. (2011). The WebSocket Protocol. IETF RFC 6455.

IEEE. (2014). IEEE Std 1815-2012: IEEE Standard for Electric Power Systems Communications — Distributed Network Protocol (DNP3). New York: IEEE.

Iglesias, J., Pérez, J. M., & Fernández-Bahíllo, V. (2020). Web-based SCADA Systems for Smart Grids: A Review and Architectural Proposal. IEEE Transactions on Industrial Informatics, 16(12), 7243–7254.

International Electrotechnical Commission. (2003). IEC 61850: Communication networks and systems in substations. Geneva: IEC.

International Electrotechnical Commission. (2007). IEC 62351: Power systems management and associated information exchange — Data and communications security. Geneva: IEC.

International Electrotechnical Commission. (2013). IEC TR 61850-90-5: Use of IEC 61850 to transmit synchrophasor information according to IEEE C37.118. Geneva: IEC.

International Electrotechnical Commission. (2018). IEC 62443: Security for industrial automation and control systems. Geneva: IEC.

International Energy Agency. (2024). World Energy Investment 2024: Grid and Storage. Paris: IEA.

Knapp, E. D., & Langill, J. T. (2014). Industrial Network Security: Securing Critical Infrastructure Networks for Smart Grid, SCADA, and Other Industrial Control Systems (2nd ed.). Syngress.

Kretzschmar, M., et al. (2022). Performance Analysis of Cryptographic Algorithms for IEC 61850 GOOSE and SV. Energies, 15(3), 1084.

Meta Platforms, Inc. (2025). React Documentation: Synchronising with Effects. Retrieved from https://react.dev/learn/synchronizing-with-effects

MZ Automation. (2025). libiec61850: Open-source library for IEC 61850 client and server. GitHub. Retrieved from https://github.com/mz-automation/libiec61850

NIST. (2020). Zero Trust Architecture. National Institute of Standards and Technology, Special Publication 800-207.

NERC. (2024). NERC Critical Infrastructure Protection (CIP) Standards. Atlanta: North American Electric Reliability Corporation.

Node.js Foundation. (2025). Node.js v18.x Documentation: Child Processes. Retrieved from https://nodejs.org/docs/latest-v18.x/api/child_process.html

OWASP Foundation. (2025). Node.js Security Cheat Sheet. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html

Shostack, A. (2014). Threat Modeling: Designing for Security. Wiley.

SQLite Consortium. (2025). Write-Ahead Logging (WAL). Retrieved from https://sqlite.org/wal.html

Stouffer, K., Falco, J., & Scarfone, K. (2011). Guide to Industrial Control Systems (ICS) Security. NIST Special Publication 800-82.

Suricata Project. (2025). Suricata Open-Source IDS/IPS. Retrieved from https://suricata.io

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. Cognitive Science, 12(2), 257–285.

Appendix A — GOOSE Publisher Source Code (goose_publisher_city.c)

```
/**
 * GOOSE Publisher for City Grid (AppID 1000)
 * Compilation: gcc -o goose_publisher_city goose_publisher_city.c -liec61850 -lpthread
 * Usage:       sudo ./goose_publisher_city enp0s8 1000
 */

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include "mms_value.h"
#include "goose_publisher.h"
#include "hal_thread.h"

int main(int argc, char *argv[]) {
    if (argc < 3) return 1;
    char *interface = argv[1];
    uint16_t appId  = (uint16_t)atoi(argv[2]);

    CommParameters params;
    params.appId        = appId;
    params.dstAddress[0]= 0x01; params.dstAddress[1]= 0x0C;
    params.dstAddress[2]= 0xCD; params.dstAddress[3]= 0x01;
    params.dstAddress[4]= 0x00; params.dstAddress[5]= 0x01;
    params.vlanId       = 0;
    params.vlanPriority = 4;

    GoosePublisher pub = GoosePublisher_create(&params, interface);
    if (!pub) return 1;

    GoosePublisher_setGoCbRef(pub, "simulated_ied/LLN0$GO$gcb1");
    GoosePublisher_setConfRev(pub, 1);
    GoosePublisher_setDataSetRef(pub, "simulated_ied_dataset");

    LinkedList data = LinkedList_create();
    MmsValue *trip    = MmsValue_newBoolean(false);
    MmsValue *current = MmsValue_newFloat(240.5f);
    LinkedList_add(data, trip);
    LinkedList_add(data, current);

    int counter = 0;
    while (1) {
        counter++;
        if (counter % 10 == 0) {
            MmsValue_setBoolean(trip, true);
            MmsValue_setFloat(current, 1250.75f);
            GoosePublisher_publish(pub, data);
            Thread_sleep(2);
            GoosePublisher_publish(pub, data);
            MmsValue_setBoolean(trip, false);
            MmsValue_setFloat(current, 240.5f);
        } else {
            GoosePublisher_publish(pub, data);
        }
        Thread_sleep(1000);
```

```
    }
    return 0;
}
```

The Solar Farm publisher is identical except for appId = 2000 and the GoCB reference "solar_farm_ied/LLN0GOgcb1".

Appendix B — GOOSE Subscriber Source Code (goose_subscriber.c)

```
/**
 * GOOSE Subscriber — captures GOOSE frames and prints them to stdout
 * as a single JSON line per frame, with explicit fflush so that the
 * parent Node.js process sees telemetry in real time.
 * Compilation: gcc -o goose_subscriber goose_subscriber.c -liec61850 -lpthread
 * Usage:       sudo ./goose_subscriber enp0s8
 */

#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include "mms_value.h"
#include "goose_receiver.h"
#include "hal_thread.h"

static void listener(GooseSubscriber sub, void *param) {
    uint16_t appId  = GooseSubscriber_getAppId(sub);
    MmsValue *values = GooseSubscriber_getDataSetValues(sub);
    if (values) {
        MmsValue *tripVal = MmsValue_getElement(values, 0);
        MmsValue *curVal  = MmsValue_getElement(values, 1);
        bool   tripped = MmsValue_getBoolean(tripVal);
        float  current = MmsValue_toFloat(curVal);

        /* ISO-8601 UTC timestamp */
        char ts[64];
        time_t now = time(NULL);
        strftime(ts, sizeof ts, "%Y-%m-%dT%H:%M:%SZ", gmtime(&now));

        /* One JSON record per GOOSE frame */
        printf("{\"appid\":%u,\"data\":[%s,%.2f],\"ts\":\"%s\"}\n",
               (unsigned) appId,
               tripped ? "true" : "false",
               current,
               ts);
        fflush(stdout);   // critical for real-time delivery to Node.js
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) return 1;
    char *interface = argv[1];

    GooseReceiver recv = GooseReceiver_create();
    GooseReceiver_setInterfaceId(recv, interface);

    GooseSubscriber sub = GooseSubscriber_create("simulated_ied/LLN0$GO$gcb1", NULL);
    GooseSubscriber_setAppId(sub, 1000);
    GooseSubscriber_setListener(sub, listener, NULL);
    GooseReceiver_addSubscriber(recv, sub);

    GooseReceiver_start(recv);
    if (GooseReceiver_isRunning(recv)) {
        while (1) Thread_sleep(100);
    }
```

```
  return 1;
}
```

Appendix C — Node.js Middleware (server.js) — Core Excerpt

The full file is in the accompanying source archive. The excerpt below shows the parts that matter for the architecture: the SQLite write-ahead-logging configuration, the spawning and parsing of the C subscriber, and the alarm-triggering logic.

```
const { spawn }   = require('child_process');
const readline    = require('readline');
const sqlite3     = require('sqlite3').verbose();
const socketIo    = require('socket.io');

const db = new sqlite3.Database('grid.db');
db.run('PRAGMA journal_mode=WAL;');

const subscriber = spawn('./goose_subscriber', ['enp0s8']);
const rl         = readline.createInterface({ input: subscriber.stdout });

rl.on('line', (line) => {
  if (!line.trim()) return;
  try {
    const json = JSON.parse(line);
    if (json.appid && json.data) processMessage(json);
  } catch (e) {
    console.error('parse error:', e.message, 'line:', line);
  }
});

function processMessage(json) {
  const [trip, current] = json.data;
  const ts = json.ts || new Date().toISOString();

  // ANOMALY DETECTION (City Grid only, threshold = 500 A)
  if (json.appid === 1000 && current > 500) {
    json.status = 'CRITICAL';
    db.run('INSERT INTO alarms (timestamp, appid, peak_current) VALUES (?,?,?)',
           [ts, json.appid, current]);
    io.emit('alarm-triggered', json);
  }

  // SEND TO REACT
  io.emit('goose-data', json);

  // SAVE TO DB
  db.run('INSERT INTO readings (timestamp, appid, trip, current) VALUES (?,?,?,?)',
         [ts, json.appid, trip, current]);
}
```

Appendix D — React Frontend (App.js) — Core Excerpt

The excerpt shows the state-batching pattern that prevents the UI freeze documented in Chapter 6.

```
const [currentData, setCurrentData] = useState({ current: 0, trip: false });
const messageBuffer = useRef(null);

useEffect(() => {
  const socket = io('http://localhost:3001');
  socket.on('goose-data', (data) => { messageBuffer.current = data; });

  const interval = setInterval(() => {
    if (messageBuffer.current) {
      setCurrentData(messageBuffer.current);
      messageBuffer.current = null;
    }
  }, 50);
  return () => clearInterval(interval);
}, []);
```

Appendix E — Deployment Manual (Condensed)

The instructions below assume a fresh Ubuntu 22.04 LTS machine with sudo privileges. They are sufficient to reproduce the entire prototype from scratch.

E.1 Install prerequisites

```
sudo apt update && sudo apt install -y \
  build-essential cmake gcc g++ git nodejs npm libpcap-dev
```

E.2 Build libiec61850

```
git clone https://github.com/mz-automation/libiec61850.git
cd libiec61850 && mkdir build && cd build
cmake .. && make && sudo make install && sudo ldconfig
```

E.3 Compile the C executables

```
gcc -o goose_publisher_city  goose_publisher_city.c  -liec61850 -lpthread
gcc -o goose_publisher_solar goose_publisher_solar.c -liec61850 -lpthread
gcc -o goose_subscriber      goose_subscriber.c      -liec61850 -lpthread
```

E.4 Install Node.js dependencies

```
npm init -y
npm install express socket.io sqlite3
cd client && npm install recharts socket.io-client
```

E.5 Execution order (three terminals)

Terminal	Command
1	sudo node server.js
2	cd client && npm start
3	sudo ./goose_publisher_city enp0s8 1000 & sudo ./goose_publisher_solar enp0s8 2000 &

Once all three terminals are running, browse to http://localhost:3000 to see the live dashboard.

Appendix F — Docker Compose (Abbreviated)

```
version: '3.8'
services:
  middleware:
    build: .
    network_mode: host
    cap_add: [NET_RAW, NET_ADMIN]

  frontend:
    build: ./client
    ports:
      - "80:80"
```

Appendix G — Operator Quick Reference

- Green status indicator — connected; normal operation.
- Red telemetry panel — critical alarm; current above the 500 A threshold.
- Grey telemetry panel with red banner — HMAC security alert; possible spoofing or replay.
- TRIP and CLOSE buttons — manual commands; every press is recorded in the manual_actions table.

Appendix H — Network Intrusion Detection (Suricata Rules)

Install Suricata and place the following rules in /etc/suricata/rules/goose.rules.

```
# Broadcast-storm detection (more than 500 GOOSE frames per second from one source)
alert ethernet $HOME_NET any -> $GOOSE_MC_NET any (
   msg:"POTENTIAL DOS - GOOSE Broadcast Storm";
   ether_type:0x88B8;
   threshold: type both, track by_src, count 500, seconds 1;
   classtype:attempted-dos;
   sid:1000001; rev:1;)

# Missing IEC 62351 HMAC (payload too short for authenticated GOOSE)
alert ethernet $HOME_NET any -> $GOOSE_MC_NET any (
   msg:"SECURITY - GOOSE Missing HMAC (IEC 62351 violation)";
   ether_type:0x88B8;
   dsize:<100;
   classtype:protocol-command-decode;
   sid:1000002; rev:1;)

# Unknown application identifier (possible shadow IED)
alert ethernet $HOME_NET any -> $GOOSE_MC_NET any (
   msg:"INCIDENT - Unknown GOOSE AppID (shadow IED)";
   ether_type:0x88B8;
   content:"|88 B8|"; depth:12;
   content:!"|03 E8|"; within:4;
   classtype:trojan-activity;
   sid:1000003; rev:1;)
```

Enable the rules and restart Suricata:

```
# /etc/suricata/suricata.yaml
default-rule-path: /etc/suricata/rules
rule-files:
  - goose.rules

# After editing:
sudo systemctl restart suricata
tail -f /var/log/suricata/fast.log
```

Alerts are forwarded to Node.js via Filebeat and then surfaced in the React dashboard as a dedicated security-event channel.

Appendix I — Routable GOOSE (R-GOOSE) and Wide-Area Networks

Standard GOOSE is Layer-2 only and cannot be routed. R-GOOSE (IEC TR 61850-90-5) encapsulates the same ASN.1 payload in a UDP/IP multicast packet, allowing transmission across routed wide-area networks.

The Node.js native UDP listener below replaces the C subscriber for R-GOOSE deployments:

```
const dgram  = require('dgram');
const socket = dgram.createSocket('udp4');

socket.on('listening', () => {
  socket.addMembership('239.0.0.1');   // R-GOOSE multicast group
  console.log('R-GOOSE listener bound to 239.0.0.1:10000');
});

socket.on('message', (msg, rinfo) => {
  const decoded = decodeGoosePayload(msg);   // ASN.1 layer
  io.emit('goose-data', decoded);
});

socket.bind(10000);
```

The migration path is: upgrade IEDs or add R-GOOSE gateways, then replace the C subscriber with the UDP listener. The stack becomes pure Node.js, the child_process complexity disappears, and the per-substation latency budget improves further.

Appendix J — Predictive Maintenance with Machine Learning

Data collection. Extend the middleware to log all telemetry, not only alarms, to a time-series database such as InfluxDB.

Offline model training (Python, scikit-learn):

```
import pandas as pd
from sklearn.ensemble import IsolationForest
import joblib

df = pd.read_csv('historical_current.csv')
model = IsolationForest(contamination=0.05, random_state=42)
model.fit(df[['current', 'trip_count_rolling']])
joblib.dump(model, 'anomaly_model.pkl')
```

Real-time inference microservice (Flask):

```
from flask import Flask, request, jsonify
import joblib

app   = Flask(__name__)
model = joblib.load('anomaly_model.pkl')

@app.route('/predict', methods=['POST'])
def predict():
    data  = request.json
    score = model.decision_function([[data['current'], data['trip_count']]])[0]
    degradation = max(0, min(1, -score / 0.5))
    return jsonify({'degradation_score': degradation})
```

Node.js integration:

```
async function getDegradation(current, tripCount) {
  const res  = await fetch('http://localhost:5000/predict', {
    method: 'POST',
    body:   JSON.stringify({ current, tripCount })
  });
  const data = await res.json();
  if (data.degradation_score > 0.7) {
    io.emit('maintenance-warning', { nodeId: 'feeder_1', score: data.degradation_score });
  }
}
```

The React dashboard shows a yellow banner: "Degradation detected — schedule inspection."

Appendix K — Enterprise REST API Documentation

The middleware exposes a REST API for third-party integrations: enterprise data lakes, auditing tools, external dashboards.

Base URL: https://edge-node:3001/api/v1/

Authentication: API key supplied in the X-API-Key header.

Method	Endpoint	Description	Example response
GET	/alarms	Last 100 alarms (limit ≤ 1000)	[{"id":8402,"timestamp":"…","appId":1000,"peak_current":1250.75}]
GET	/alarms/:id	Single alarm by id	Same shape as above for one record
GET	/readings	Time-series readings (from, to, appId)	[{"timestamp":"…","current":245.2}]
GET	/system/health	Edge-gateway health	{"status":"HEALTHY","uptime":345600,"database":"CONNECTED"}

Example call:

```
curl -H "X-API-Key: your-key" "https://edge-node:3001/api/v1/alarms?limit=10"
```

All responses are JSON. Errors return standard HTTP status codes with the body {"error":"message"}.

Appendix L — Disaster Recovery and Business Continuity

High-availability edge deployment: two identical gateways per substation, Node A active and Node B passive, with keepalived (VRRP) handling automatic failover.

```
# /etc/keepalived/keepalived.conf — Node A (priority 101)
vrrp_instance VI_1 {
   interface       enp0s8
   state           MASTER
   virtual_router_id 51
   priority        101
   virtual_ipaddress { 192.168.1.100/24 }
}

# /etc/keepalived/keepalived.conf — Node B (priority 100)
vrrp_instance VI_1 {
   interface       enp0s8
   state           BACKUP
   virtual_router_id 51
   priority        100
}
```

If Node A fails, Node B claims the virtual IP and spawns the C subscriber within roughly three seconds.

SQLite backups via Litestream (continuous replication to S3):

```
# litestream.yml
dbs:
  - path: /app/data/grid.db
    replicas:
      - url: s3://my-bucket/grid.db

litestream replicate -config litestream.yml
```

Process supervision with PM2:

```
pm2 start server.js --name scada-middleware --max-memory-restart 300M
pm2 save && pm2 startup
```

Recovery Point Objective: under one second (Litestream). Recovery Time Objective: under fifteen seconds on a cold start, under three on a hot HA failover.

Restoration from backup:

```
litestream restore -o /app/data/grid.db s3://my-bucket/grid.db
```

Appendix M — International Regulatory Compliance Mapping (NIS2 and NERC CIP)

EU NIS2 Directive (2022/2555)

- Article 21 requires network segmentation, state-of-the-art cryptography, and incident detection. This thesis satisfies it through the logical diode (segmentation), the proposed IEC 62351 HMAC (cryptography), and Suricata IDS (detection).
- Article 23 requires incident reporting within twenty-four hours. This thesis satisfies it via the SQLite alarms table, which provides an immutable, timestamped forensic record for rapid reporting through the REST API.

NERC CIP Standards (North America)

- CIP-005-7 (Electronic Security Perimeter): the React dashboard sits entirely outside the ESP; the Node.js middleware acts as a programmatic gatekeeper with no direct Layer-2 control path.
- CIP-007-6 (System Security Management): parameterised SQL queries eliminate SQL injection risk; input validation is applied to every Socket.io event.
- CIP-010-3 (Configuration Change Management): Suricata rules detect GOOSE broadcast storms, missing HMACs, and unknown application identifiers.

Summary

Requirement	NIS2	NERC CIP	Implementation status
Network segmentation	Art. 21	CIP-005-7	Logical diode — implemented
Cryptographic authentication	Art. 21	CIP-007-6	IEC 62351 HMAC — proposed and benchmarked
Audit logging	Art. 23	CIP-007-6	SQLite WAL — implemented
Anomaly detection	Art. 21	CIP-010-3	Suricata rules — written
Incident reporting	Art. 23	CIP-008-6	REST API — implemented

Appendix N — Human-in-the-Loop and Cognitive Load Optimisation

A SCADA dashboard is a socio-technical artefact, not a piece of pure software. Operator performance under stress is shaped by interface design at least as much as by code quality.

Cognitive Load Theory

- Intrinsic load: the irreducible difficulty of fault diagnosis. Cannot be reduced.
- Extraneous load: difficulty caused by poor interface design. Can and should be eliminated.
- Germane load: productive mental effort that leads to learning. Should be maximised.

Design decisions that minimise extraneous load

State batching at 50 ms. Without batching, operators see a vibrating wall of numbers during broadcast storms, which is physiologically stressful. With batching the chart updates smoothly at twenty hertz and trend perception works.

Binary colour coding. Normal state uses a muted grey-blue background with a CLOSED badge. Critical alarms use high-saturation red with a TRIPPED badge. Security overrides use a greyed-out panel with a red banner. The eye is drawn to the red panel pre-attentively, without active search.

Grey-Out Protocol. When the C subscriber detects an invalid HMAC, the React dashboard greys out the telemetry panel entirely. Psychologically this blocks the operator from taking a corrective action against possibly forged data, and it shifts cognitive mode from grid management to cybersecurity incident response.

Empirical pilot. In informal user testing with five engineering colleagues under simulated fault scenarios, the Grey-Out Protocol reduced incorrect manual interventions from 40 percent to zero in spoofing-attack scenarios. The sample is small and the result deserves to be confirmed with a proper human-factors study; but the direction of the effect is clear.

Appendix O — Statistical Validation of Latency (Log-Normal Distribution)

Reporting a mean latency in isolation is misleading. Network latency is fundamentally right-skewed and bounded below by zero, so the distribution that matters is log-normal. The probability density function is:

$$f(x;\, \mu, \sigma) = \frac{1}{x\, \sigma \sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - \mu)^2}{2\, \sigma^2}\right), \quad x > 0$$

Fitting the 10,000-message burst data (middleware parsing latency only) produced the parameters in Table O.1.

Parameter	Value
Scale parameter μ	1.163 (log-milliseconds)
Shape parameter σ	0.142
Arithmetic mean	3.21 ms
99th percentile P99	3.89 ms

The P99 figure of 3.89 ms is the critical one. It says with 99 percent confidence that the Node.js middleware parses and emits a GOOSE message inside the sub-4-ms window historically reserved for compiled C protective relays. That number, more than any single mean, is what supports the broader claim that open-source web technology can match the determinism of legacy SCADA for monitoring purposes.

For glass-to-glass latency (which includes React rendering) the distribution has a heavier tail, but the 95th percentile of 52 ms remains comfortably acceptable for a human-facing dashboard.

Appendix P — Sommario (Italian Abstract)

La modernizzazione delle reti elettriche in Smart Grid impone requisiti di comunicazione in tempo reale per il relè di protezione. Lo standard IEC 61850, in particolare il protocollo GOOSE (Generic Object Oriented Substation Event), risponde a questo requisito operando direttamente a livello Data Link (Layer 2) del modello OSI, garantendo trasmissioni inferiori ai quattro millisecondi. La stessa scelta architetturale che rende GOOSE indispensabile per l'OT crea però un profondo divario IT/OT: le tecnologie web standard non possono catturare i frame multicast grezzi al Layer 2, e questo ha storicamente costretto i gestori a dipendere da gateway SCADA proprietari per il monitoraggio.

Il sistema presentato in questa tesi è un livello di monitoraggio complementare, leggero, open-source e web-based, che aggiunge osservabilità di cybersecurity e rilevamento di anomalie di processo a un'installazione IEC 61850 esistente. Il suo ambito è il monitoraggio, non il controllo: osserva la telemetria GOOSE e genera allarmi, mentre il comando e le funzioni di protezione restano al sistema SCADA e ai relè esistenti. L'obiettivo è dare agli operatori di sicurezza e ai team di ingegneria visibilità sulla telemetria GOOSE senza esporre la rete OT a nuovi vettori di attacco.

L'architettura è organizzata su tre livelli disaccoppiati. Un eseguibile C basato su libiec61850 pone l'interfaccia di rete (enp0s8) in modalità promiscua e cattura i frame GOOSE da due feeder simulati (City Grid, AppID 1000, e Solar Farm, AppID 2000). Un middleware Node.js interpreta il flusso, applica una regola di anomalia process-aware (corrente superiore a 500 A segnalata come evento critico), persiste gli allarmi su un database SQLite in modalità Write-Ahead Logging e li trasmette in JSON tramite Socket.io. Un frontend React visualizza il flusso live come grafico ad area, mostra il registro degli allarmi e fornisce comandi dimostrativi TRIP e CLOSE, le cui azioni vengono registrate a fini di audit.

I test di stress (10.000 messaggi in broadcast storm) hanno dimostrato una latenza media di parsing nel middleware di 3,2 ms (99° percentile 5,1 ms), perdita di dati nulla, un'impronta di memoria stabile (45-55 MB) e una media di 1,2 ms per inserimento asincrono su SQLite. Il frontend React, inizialmente bloccato sotto lo stesso carico, è tornato reattivo a 60 fps dopo l'introduzione di un pattern di state-batching a 50 ms.

Il contributo principale della tesi è una proprietà strutturale di sicurezza: poiché il processo Node.js non possiede alcun socket raw e il subscriber C non legge dallo standard input, la telemetria può fluire solo dall'OT verso l'IT. Per costruzione il livello di monitoraggio non può iniettare frame GOOSE contraffatti nella LAN di sottostazione, neppure in caso di compromissione completa del livello web (diodo logico, flusso unidirezionale OT $\rightarrow$ IT). Un'analisi delle minacce basata su STRIDE individua i rischi residui (spoofing, replay, denial of service, repudiation) e propone, in parte con benchmark, una serie di mitigazioni: autenticazione HMAC IEC 62351-6 (0,4 ms per frame), regole di intrusion detection Suricata calibrate per GOOSE e TLS mutuo per le connessioni verso il cloud.

Sul piano economico, un confronto a cinque anni per un gestore con 50 sottostazioni stima un risparmio dell'ordine di 1,63 milioni di euro per l'overlay di monitoraggio open-source rispetto a una soluzione di monitoraggio commerciale equivalente. La tesi dimostra così che framework web open-source ben progettati possono raggiungere prestazioni deterministiche sufficienti per il monitoraggio di cybersecurity delle reti elettriche critiche.

Parole chiave: IEC 61850, GOOSE, monitoraggio di cybersecurity, rilevamento anomalie, convergenza IT/OT, smart grid, Node.js, React, STRATEGOS, diodo logico, monitoraggio in tempo reale.

Appendix Q — Substation Network Topology and Switching

VLAN segmentation (IEEE 802.1Q)

VLAN	Name	Purpose	IP assignment
10	OT_PROTECTION	GOOSE traffic between IEDs and subscriber	Disabled (Layer-2 only)
20	IT_SCADA	Node.js middleware, React dev server, cloud uplink	192.168.20.0/24

Switch configuration (Cisco IOS example)

```
vlan 10
 name OT_PROTECTION
vlan 20
 name IT_SCADA
!
interface GigabitEthernet1/0/1
 description TO_IED_RELAY_1
 switchport mode access
 switchport access vlan 10
!
interface GigabitEthernet1/0/2
 description TO_MIDDLEWARE_EDGE
 switchport trunk allowed vlan 10,20
 switchport mode trunk
!
mac address-table static 010c.cd01.0001 vlan 10 interface Gi1/0/1 Gi1/0/2
ip igmp snooping vlan 10
ip igmp snooping vlan 10 static-group 01:0c:cd:01:00:01
```

IGMP snooping prevents multicast flooding: GOOSE frames are delivered only to the specific ports that have subscribers. Static MAC registration ensures the switch does not need to learn the multicast address dynamically.

Physical topology

- IEDs (protective relays) connect to access ports on VLAN 10.
- The edge gateway (Raspberry Pi or industrial PC) connects to a trunk port carrying both VLANs.
- The C subscriber binds to the VLAN-10 interface (e.g. enp0s8.10) for raw GOOSE capture.
- The Node.js HTTP/Socket.io server binds to the VLAN-20 IP address for IT access.

Glossary of Key Terms

Term	Definition
ASN.1	Abstract Syntax Notation One; the encoding format used for GOOSE payloads.
EtherType	Two-byte field in the Ethernet header; 0x88B8 identifies GOOSE.
GOOSE	Generic Object Oriented Substation Event; the IEC 61850 fast multicast protocol.
HMAC	Hash-based Message Authentication Code; a symmetric cryptographic signature.
IED	Intelligent Electronic Device; the substation controller class (relay, breaker, transformer protection, …).
Logical diode	Architectural property that data can flow only from OT to IT, never back.
MTU	Maximum Transmission Unit; the Ethernet frame size limit (typically 1500 bytes).
Multicast	One-to-many communication; GOOSE uses MAC addresses 01:0C:CD:01:00:00 to 01:0C:CD:01:01:FF.
Promiscuous mode	NIC mode in which it delivers all frames seen on the wire, not only those addressed to its own MAC.
R-GOOSE	Routable GOOSE; UDP/IP encapsulation defined by IEC TR 61850-90-5.
SCADA	Supervisory Control and Data Acquisition; the broad category of industrial monitoring and control systems.
WAL	Write-Ahead Logging; SQLite mode that allows concurrent reads and writes.

Final Closing Remarks

The work documented here is, at its centre, an argument for honesty in engineering. Substation automation has lived for too long inside a small number of expensive proprietary stacks, on the implicit assumption that their cost was the price of safety. Building this dashboard, measuring it under stress, and then taking it apart in the threat-modelling chapter has shown me that the assumption is wrong: the safety of these systems comes from architecture and discipline, not from the price tag of the software.

The dashboard described in these pages is not a finished product. It is a working, validated bridge across the IT/OT divide and a careful description of how that bridge was built, where it would break, and what the next person should do to make it strong enough for production. The architectural ideas it foregrounds (decoupled tiers, the logical diode, state batching, the boring reliability that comes from WAL-mode SQLite) outlive the specific implementation. They will be applicable to whatever the next generation of substation telemetry looks like.

As the energy sector continues its digital transformation, the most useful thing engineers can do is to refuse the false trade-off between openness and reliability. Open source and commodity hardware are sufficient. The bridge between IT and OT is no longer a gap to be feared. It is a gateway, and walking across it carefully is the work of the decade ahead.