POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Distributed Computing for Real-Time Computer Vision in Manufacturing

Design, Study and Implementation of a Distributed Edge Architecture for Real-Time Computer Vision in the Manufacturing Industry

João David Moniz Vieira

School of Management and Technology
Department of Computer Engineering
Master in Mobile Computing

Leiria, July 2026

This page intentionally left blank.

POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Distributed Computing for Real-Time Computer Vision in Manufacturing

Design, Study and Implementation of a Distributed Edge Architecture for Real-Time Computer Vision in the Manufacturing Industry

João David Moniz Vieira

Supervisor: Luis Frazão

School of Management and Technology
Department of Computer Engineering
Master in Mobile Computing

Internship

Leiria, July 2026

This page intentionally left blank.

Distributed Computing for Real-Time Computer Vision in Manufacturing

This page intentionally left blank.

Acknowledgements

"Whatever is received is received according to the mode of the receiver."
—St. Thomas Aquinas

As such, i would first like to address my family for who have been dealing with my academic journey since the beginning, and its support has been constant, wether i flunked, failed to deliver, or just sucuumbed to laziness and anhedonia. My mother Deonemia Vieira for their attention and understanding specially in this latter period of the journey. And my father David Vieira for patiently listening to my developments and problems. Finally my brother, Lucas Vieira, who always found a way to distract me from the stress about deadlines and work in general. I would've loved to see my grandparents, who passed away before i could share with them this journey, but unfortunately, i can only share my achievements with my Maternal Grandpa, whos personality always radiated livelyness. To the remainder, unfortunately i can only dedicate this work to their memory, from the indominatable spirit of Maria Alexandra Vieira, the serene and wise David Lopes Vieira, and lastly to the loving and caring Maria Conceição Vieira, who has recently passed away.

Appart from family support. This work wouldn't be finished without the supervision of Professor Luís Frazão, who guided me through the process and had to deal with not only my procrastinations and lack of results, but also with my incessant talks that sidequested the meetings done for the project.

I also want to thank the Twevo Board, Prof. Monica and Prof. Carlos, who alongside with Prof. João Gil, who hosted my experience and without them this document would've not been possible.

Its not only the board that is important, since my coworkers at Twevo were the fundamental piece for the existance of this work: A special thanks to the eldest of the team: Miguel Silva (Mike) who has been dealing with my jokes and laziness since the beginning of the internship, and who has an unkillable motivation to show results.

To André Neto (Neto) and his taunting for helping me regain some motivation to finish the work, i hope after you read it (or not) you will find it "TOP! está TOP",

Daniel Carreira (Cabo Pires) for making me ever so modest with every decision i implement, “vais fazer sqlite e vais gostar”.

Daniel Carias (Querias), who has the biggest pacience i’ve ever seen on an engineer, and who has been a great help to me in the development of the work, specially in the actuator/controller part and kafka. I know its not fair that i always grab you to the weirdest and annoying tasks i owe you a year of shitty work for it.

Eurico Almeida (Euboro07), for stealing the ignorance i stubornly didn’t want to lose. I cant say you weren’t right about nvidia and docker containers.

Xavier Bento (Xavi) for all the valuable insights and help across the day to day work. Ill give you permission to sleep over the boring details of the work.

To the Tomas Schroder who had to listen to my rants about how XYZ in his project would solve it, when XYZ was already off the table. To Bruno Correia (Chibanga), for having the patience to offload work from me, you were my glue “está tudo partido, mas isto com cola funfa”. Ricardo Camarate (Camarate), I guess once in formula student, always in formula student, i hope you didnt cringe at my “axanatos”, i did my part of getting revenge for all those times powertrain stole things from the eletrical department. This new Lab setup is too clean for my eletronic afflictions And lastly to Beatriz Ribeiro, being new doesnt mean that the atmosphere didnt improve significantly. Jerusha, which the language barrier wasnt an obstacle when it came to teaching about server maintenance and proper resource allocation policies. I would also like to extend my gratitude to a colleague , José Areia, who let me try his template which im now unsure if it has or not become mainstream. For close friends i would like to thank Diogo Alpendre, Duarte Dias and Carolina, Im sorry if i have cutted myself from social life too many times, but the stress and procrastination cycle was a hard one to break. I wanted to thank everyone from the Formula Student of IPLeiria (LART), for the unforgettable experience and the friendships that came with it. I hope for a T28, T30, T50. Im sorry for not simplifying the ACU when it came to periferals, looking back some flexibility was too much, Bruno hopefully you dont keep a grudge. “pops and bangs” to you too Candido. I want to see an eFused Chassis José I hope the arch linux lessons weren’t boring Chen, hope to see both of you on the stars Perdu e Diogo. You should’ve patented the burning mosfets Fadigas. I want to see Linux on your pc Sousa, If i catch you not using a vLLM i will trash your sprint Tojal, and to the rest of the LART family, thanks for the good times, and for the good vibes. Leaving the LART was one of the hardest decisions i had to make, and i know both the bittersweet consequences my departure brought and that I didnt make my narrative any better but i truly gave it my best (you wont see me learning obscure Can Bus settings in this work). Lastly a big thanks to Triton-Maceira and its employees every saturday its a blast from the past, where issues were a screwdriver away from being solved, a special academic hug to you Alex, i hope i get to call you professor one day.

And a thank you to everyone that is reading this work.

Ad Majorem Dei Gloriam

Abstract

Modern manufacturing increasingly depends on distributed, software-based systems in which computer vision and inference workloads must operate under strict latency, availability and resource constraints. In such environments, end-to-end responsiveness is determined not only by model inference, but also by the supporting messaging, storage and orchestration layers, whose design becomes a cross-cutting concern for both system architects and Development and Operations (DevOps) practitioners.

This work was carried out during an internship at Twevo S.A., a Portuguese company that develops AI-powered edge computer vision solutions for the manufacturing sector, and is framed within the evolution of the company's *VIEXPAND AI* platform. The objective was to study, design a distributed edge architecture capable of keeping end-to-end latency low while improving availability and lowering the cost of developing new components, while leveraging industry-standard DevOps workflows to ensure long-term maintainability.

Four workstreams were implemented. A lightweight storage subsystem based on SQLite was designed to replace a legacy text log based persistence mechanism, with tuned journalling modes and a refactored worker-thread concurrency model. A containerisation strategy for the graphical components of the platform was developed, combining Docker images with a remote-desktop-based display layer to enable reproducible, isolated and securely exposed deployments. A Kafka-based messaging backbone was introduced to replace a point-to-point Transmission Control Protocol (TCP) messaging system, including topic and partition layouts, producer and consumer tuning for low latency, and a thin `librdkafka` wrapper to support deterministic benchmarking and integration. Finally, a deterministic perception-to-action controller was implemented to translate vendor-agnostic orders into machine-executable instructions.

The artefacts were assessed through targeted validation activities, including database latency and resource measurements, Kafka broker and integration tests, a container display-latency comparison, and actuator functional and latency tests. The remaining contribution consists of a set of integrated, production-oriented components and a body of design principles and best practices for deploying low-latency, resource-efficient edge inference systems in manufacturing environments.

Keywords: Edge Computing, Low Latency, Stream Processing, Real-Time Data Processing, Kafka, DevOps, Computer Vision.

This page intentionally left blank.

Contents

List of Figures xiii

List of Tables xvii

Acronyms xxi

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Internship Context 3

1.4 Objectives 5

1.5 Research Approach 6

1.6 Main Contributions 6

1.7 Document Structure 6

2 Methodology **8**

2.1 Research Paradigm 8

2.2 Methodology Development 8

2.3 Validation, Methods and Tools used 10

2.3.1 Methods 10

2.3.2 Tools 11

2.4 Summary 12

3 State of the Art **13**

3.1 Wired Communication 13

3.1.1 Fieldbus, Real time Control systems 13

3.1.2 RDMA 15

3.1.3 Ethernet 17

3.2 Application Layer Messaging Protocols 18

3.2.1 MQTT 19

3.2.2 RabbitMQ 19

3.2.3 Kafka 20

3.2.4 Comparison Table 21

3.3 Low Latency Storage Solutions 22

3.3.1 Relational Databases 22

3.3.2 Non-relational Databases 24

3.4 Summary 24

4 Design, Implementation and Evaluation **26**

4.1 Lightweight Storage Subsystem 26

4.1.1 Design Rationale 26

4.1.2 Implementation 27

4.1.3 Evaluation 28

4.2 Secure GUI Exposure through Containerisation 34

4.2.1 Design Rationale 34

4.2.2 Implementation 34

4.2.3 Evaluation 36

4.3 Messaging Backbone Design 38

4.3.1 Design Rationale 38

4.3.2 Implementation 38

4.3.3 Evaluation 41
4.4 Actuator / Controller (Order Translation) Component 43
4.4.1 Design Rationale 43
4.4.2 Implementation 43
4.4.3 Evaluation 45
4.5 Summary 50

5 Conclusion **52**
5.1 Summary of Work 52
5.2 Revisiting the Objectives 53
5.3 Main Contributions 54
5.4 Limitations and Future Work 55

References 59

Appendices

A Actuator Test 1 — Per-Channel ON-Time Statistics **66**

B Tools and Software **71**

C Hardware Specifications **73**

D Kafka Broker Benchmark Data **74**

This page intentionally left blank.

List of Figures

1.1 A diagram illustrating the system architecture of VIEXPAND AI (source: Twevo S.A.) 5

3.1 An example of an Ethercat layout. 14

3.2 An example of a Kafka layout. 21

4.1 A simplified view of the aforementioned system architecture with remote desktop integration. 35

4.2 A screenshot of the remote desktop session, showing the application running inside the container in a production environment. 36

4.3 Latency measurement under the control setup (direct host launch, no containerisation). 37

4.4 Latency measurement under the containerised remote-desktop setup. . 37

4.5 Wrapper integration test — Machine 1 publish-to-consume latency distribution. 42

4.6 Wrapper integration test — Machine 2 publish-to-consume latency distribution. 42

4.7 Test 1 Subtest 1.1 (PoE supply): per-channel ON-time violin plot. 46

4.8 Test 1 Subtest 1.2 (9 V supply): per-channel ON-time violin plot. 47

4.9 Test 1 Subtest 1.3 (12 V supply): per-channel ON-time violin plot. . . . 47

4.10 Test 1 Subtest 1.4 (24 V supply): per-channel ON-time violin plot. . . . 47

4.11 Test 3 Subtest 3.1 (9 V supply): actuation-latency violin plot. Both runs exhibited consistent failures during the sweeping phase due to insufficient power budget. 48

4.12 Test 3 Subtest 3.2 (12 V supply): actuation-latency violin plot. 49

4.13 Test 3 Subtest 3.3 (24 V supply): actuation-latency violin plot. 49

4.14 Test 3 Subtest 3.4 (PoE supply): actuation-latency violin plot. 50

This page intentionally left blank.

List of Tables

2.1 Operationalisation of the DSR framework across the four workstreams. 9

3.1 Comparison of MQTT, RabbitMQ and Kafka 22

4.1 Database profile common to all benchmark runs. 29

4.2 Analytical query latency (ms) on the trimmed 1.19 GB database, two consecutive runs per journal mode. 30

4.3 Read latency under live CTC traffic. 31

4.4 Per-statement (`db_op`) latency inside batched write transactions. 31

4.5 End-to-end transaction (`write`) latency. 31

4.6 Process RSS at key lifecycle points during the real-time sessions. 32

4.7 Distinction between the three timing metrics reported in the actuator evaluation. 46

A.1 Test 1 Subtest 1.1 (PoE) — ON-Time Duration Statistics (ms) 67

A.2 Test 1 Subtest 1.2 (9 V) — ON-Time Duration Statistics (ms) 68

A.3 Test 1 Subtest 1.3 (12 V) — ON-Time Duration Statistics (ms) 69

A.4 Test 1 Subtest 1.4 (24 V) — ON-Time Duration Statistics (ms) 70

C.1 Hardware specifications for Kafka wrapper integration testing. 73

D.1 Broker saturation throughput — 1 MB batch, 100 KB socket buffers, `acks=1`. 74

D.2 Tuned configuration — 16 KB batch, async producer, varying record size. 75

D.3 Realistic-load latency — 100 records per run, varying record size. 75

D.4 Under-provisioned buffers — 1 KB socket buffers, 2 KB batch, `acks=0`. . 76

This page intentionally left blank.

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. (*p. 23*)
ADAS Advanced Driver-Assistance Systems. (*p. 13*)
AI Artificial Intelligence. (*p. 3, 4, 15, 71*)
AMQP Advanced Message Queuing Protocol. (*p. 19*)
API Application Programming Interface. (*p. 4, 12, 39, 43, 72*)

CI/CD Continuous Integration/Continuous Deployment. (*p. 4, 56*)
CRUD Create, Read, Update, Delete. (*p. 23*)
CTC Camera-to-Display Latency Test Controller. (*p. xvii, 29, 31–33*)

DevOps Development and Operations. (*p. iv, 2, 4, 48*)
DNS Domain Name System. (*p. 17*)
DSR Design Science Research. (*p. xvii, 6, 8, 9*)

E2E End-to-End. (*p. 20*)
ECN Explicit Congestion Notification. (*p. 16*)

FDW Foreign Data Wrapper. (*p. 23*)
FPGA Field-Programmable Gate Array. (*p. 3*)

GPIO General-Purpose Input/Output. (*p. 11, 55*)
GPU Graphics Processing Unit. (*p. 12, 71*)
GUI Graphical User Interface. (*p. 7*)

HDR High Data Rate. (*p. 15*)
HPC High-Performance Computing. (*p. 15*)
HTTP Hypertext Transfer Protocol. (*p. 17, 18*)

IETF Internet Engineering Task Force. (*p. 16, 17*)
IoT Internet of Things. (*p. 4, 19, 38*)
IP Internet Protocol. (*p. 16–18*)
IRT Isochronous Real-Time. (*p. 14*)

ISO International Organization for Standardization. (*p. 15, 72*)
iWARP Internet Wide Area RDMA Protocol. (*p. 15, 16, 24*)

JSON JavaScript Object Notation. (*p. 44*)
JVM Java Virtual Machine. (*p. 38*)

LLM Large Language Model. (*p. 12, 71*)

MCC Monitoring Control Center. (*p. 10*)
MVCC Multi-Version Concurrency Control. (*p. 23*)

NAT Network Address Translation. (*p. 18*)
NDR Next Data Rate. (*p. 15*)
NIC Network Interface Controller. (*p. 16, 73*)
NoSQL Not Only SQL. (*p. 24*)
NVMe Non-Volatile Memory Express. (*p. 29, 55, 73*)
NVR Network Video Recorder. (*p. 18*)

OLTP Online Transaction Processing. (*p. 23*)
ONVIF Open Network Video Interface Forum. (*p. 18*)
ORM Object Relational Mapping. (*p. 27*)
OS Operating System. (*p. 28, 29*)
OSI Open Systems Interconnection. (*p. 13*)

PFC Priority Flow Control. (*p. 16*)
PI Profinet IRT. (*p. 14*)
PoE Power over Ethernet. (*p. xiii, xiv, xvii, 6, 11, 45, 46, 48, 50, 51, 53, 54, 67*)
PTP Precision Time Protocol. (*p. 55*)

QoS Quality of Service. (*p. 19, 22*)
QUIC Quick UDP Internet Connections. (*p. 17, 19*)

RAII Resource Acquisition Is Initialization. (*p. 39*)
RAM Random Access Memory. (*p. 11, 27, 29, 73*)
RDBMS Relational Database Management System. (*p. 23*)
RDMA Remote Direct Memory Access. (*p. ix, 7, 15, 16, 24*)
RFC Request for Comments. (*p. 16–18*)
RoCE RDMA over Converged Ethernet. (*p. 15, 16, 24*)
RSS Resident Set Size. (*p. xvii, 32*)
RT Real-Time. (*p. 14*)
RTCP RTP Control Protocol. (*p. 18*)
RTP Real-time Transport Protocol. (*p. 18*)
RTSP Real Time Streaming Protocol. (*p. 18, 24*)

SCTP Stream Control Transmission Protocol. (*p. 16*)
SDK Software Development Kit. (*p. 18*)
SQL Structured Query Language. (*p. 22, 23, 27*)
SRE Site Reliability Engineering. (*p. 4, 48*)
SSD Solid State Drive. (*p. 29, 55, 73*)
STOMP Simple Text Oriented Messaging Protocol. (*p. 19*)

TCP Transmission Control Protocol. (*p. iv, 2, 9, 16–19, 24, 25, 51–53*)
TCP/IP Transmission Control Protocol/Internet Protocol. (*p. 13, 14, 16, 17*)
TFO TCP Fast Open. (*p. 17*)
TLS Transport Layer Security. (*p. 17*)
TTL Time to Live. (*p. 19*)
TTM Time to Market. (*p. 12, 39, 71*)

UDP User Datagram Protocol. (*p. 16–18, 24*)
UI User Interface. (*p. 33*)

VNC Virtual Network Computing. (*p. 9, 51*)

WAL Write-Ahead Logging. (*p. 4, 9, 22, 25, 28–30, 32, 33, 50, 54*)

YAML YAML Ain't Markup Language. (*p. 9, 43, 48, 51, 53*)

This page intentionally left blank.

1 Introduction

1.1 Motivation

Modern manufacturing environments increasingly rely on real-time computer vision systems to automate quality assurance, enhance operational safety, and optimize production throughput. Unlike traditional enterprise applications, these industrial systems demand immediate processing of high-bandwidth visual data. Consequently, there is a fundamental shift toward *distributed edge computing architectures* [1], where data processing and inference are performed as close to the physical source as possible to satisfy strict latency requirements.

However, deploying computer vision at the industrial edge introduces a unique set of challenges and constraints:

- **Latency and Determinism:** In manufacturing, excessive latency or jitter in processing can lead to production defects or equipment damage. Systems must operate under strict real-time or near-real-time constraints [2].
- **Resource Constraints:** Industrial edge nodes often possess limited computational power, storage, and memory compared to cloud environments [3], requiring highly optimized deployment strategies.
- **Environmental Heterogeneity:** Integration involves a diverse ecosystem of legacy industrial equipment, various camera interfaces, and fluctuating network conditions, making standardized deployment difficult [4].
- **Availability and Reliability:** Unplanned downtime in a factory setting results in significant financial losses. Systems must maintain high availability and include robust failover mechanisms [5].

In this context, small architectural decisions—such as the choice of messaging pro-

tocols, container orchestration, or resource scheduling—can have a disproportionate impact on the system's performance, maintenance, and long-term scalability.

The motivation for this work stems from a professional internship at **Twevo S.A.**, specifically focusing on their **VIEXPAND AI** platform. VIEXPAND AI is a modular solution designed to acquire data from industrial cameras and perform real-time AI inference to extract actionable insights. While the platform is currently operational, there is a continuous need to improve its components to better support edge-native constraints. This research seeks to bridge the gap between high-level DevOps practices and the granular technical requirements of industrial computer vision, ensuring that software updates and system monitoring do not compromise the rigorous performance gates required by the manufacturing floor.

1.2 Problem Statement

The *VIEXPAND AI* platform, while operational, presented several concrete engineering problems that this internship set out to address. Four distinct but interrelated shortcomings were identified across the platform's data persistence, deployment, inter-service communication, and physical actuation layers.

Data persistence. Structured operational data was persisted through a bespoke TCP-based messaging and logging mechanism. As the volume of records grew into the millions, the absence of transactional guarantees, complex query support, and decoupling between persistence and transport became untenable. A lightweight, low-cost, embedded storage solution was therefore needed that could operate within edge-device resource budgets without sacrificing durability or query performance at scale.

Environment reproducibility. The growing complexity of the software stack and its dependencies made it increasingly difficult to replicate consistent development and production environments. A containerisation strategy was required that could encapsulate the full graphical application, including GPU-accelerated inference, while exposing the GUI securely and without measurable latency overhead.

Inter-service communication. The platform's original point-to-point messaging layer, built directly on TCP sockets, suffered from connection-state anomalies, the absence of backpressure, and a high cost of extensibility: each new integration demanded a new, bespoke transport plugin. A replacement messaging backbone was needed that could provide durable, partition-based message delivery with native support for consumer groups and replay.

Physical actuation. A client requirement introduced the need to close the perception-to-action loop: once the vision pipeline detected an event of interest, the system had to actuate a physical device within a bounded latency window. Because incorrect com-

mands carried a real risk of hardware damage, the tolerance for misfires — whether caused by a software defect or an erroneous device configuration on our part — was effectively zero.

Two requirements followed from this constraint. First, the actuation system had to behave deterministically: commands had to be dispatched within a defined polling interval and target only the correct output channels. Second, to make fault attribution tractable in a production environment, actuation sequences needed to be expressed declaratively rather than hard-coded against a specific device. By externalising instruction definitions into configuration, failures can be systematically attributed to either a code defect or a misconfiguration, narrowing the scope of diagnosis and operational responsibility when an incident occurs.

Each of these problems is addressed as a dedicated workstream in Chapter 4, where the design rationale, implementation, and empirical evaluation of the corresponding artefact are presented.

1.3 Internship Context

This report was developed as part of an internship at Twevo S.A, a company that focuses on designing Artificial Intelligence (AI)-powered Edge Computer Vision solutions for the manufacturing sector, among other industries.

Twevo S.A, is a company based in Leiria with its financial headquarters in Coimbra, Portugal. The company was founded in February 2017, by its two founders, Mónica Figueiredo (PhD) and Carlos Ribeiro (PhD). It is officially registered as an S.A in the Portuguese/EU business registry, SME in accordance with EU recommendation 2003/361.

Originally, the company was focused on providing solutions based on reliable and robust telecommunications solutions, such as software-defined radio based on Field-Programmable Gate Array (FPGA) devices. However since 2024, the company has shifted its focus to providing AI-powered Edge Computer Vision solutions for the manufacturing sector, among other industries.

Company domains

The company's activities span the following interrelated technical domains:

1. Computer Vision — real-time inference, model development and on-device optimization.

2. Edge / Internet of Things (IoT) — device management, data ingestion and edge runtime integration.
3. DevOps & Site Reliability Engineering (SRE) — Continuous Integration/Continuous Deployment (CI/CD), observability, orchestration and release automation.
4. Cybersecurity — secure device provisioning, encryption and vulnerability management.
5. Data Engineering & Storage — streaming pipelines, databases and low-latency storage.
6. Robotics & Automation — perception-to-control integration and factory automation systems.
7. Embedded Systems & Firmware — hardware interfaces and resource constrained software.
8. Platform & Application Engineering — back-end services, front-end dashboards and Application Programming Interface (API).
9. Research & ML Engineering — prototyping, benchmarking and model lifecycle management.

VIEXPAND AI, is one of their most popular software Solutions, in which many projects presented in this work are framed, it consists of multiple components, as shown in Figure 1.1:

- A data acquisition component that is responsible for collecting data from various sources, mainly cameras, amongst other devices.
- A data processing component that is responsible for processing the acquired data in real-time, using AI algorithms and models to extract insights and information from the data.
- A front-end component that is responsible for providing a user interface for visualizing the processed data and insights, as well as for configuring and managing the system.

The work performed during this internship is organized into a set of interrelated engineering activities, each addressing a specific requirement of the *VIEXPAND AI* platform. These activities collectively aim to deliver a low-latency, resource-efficient and production-ready edge inference stack, and comprise the following workstreams:

1. **Lightweight storage subsystem.** Design and implementation of a low-latency, low-memory footprint storage layer based on SQLite. This included selection and tuning of storage modes (e.g. Write-Ahead Logging (WAL) , journaling), pragmatic durability/latency trade-offs, and integration with the data ingestion pipeline to minimize end-to-end persistence overhead.
2. **Secure GUI Exposure through Containerisation.** Encapsulation of part of the

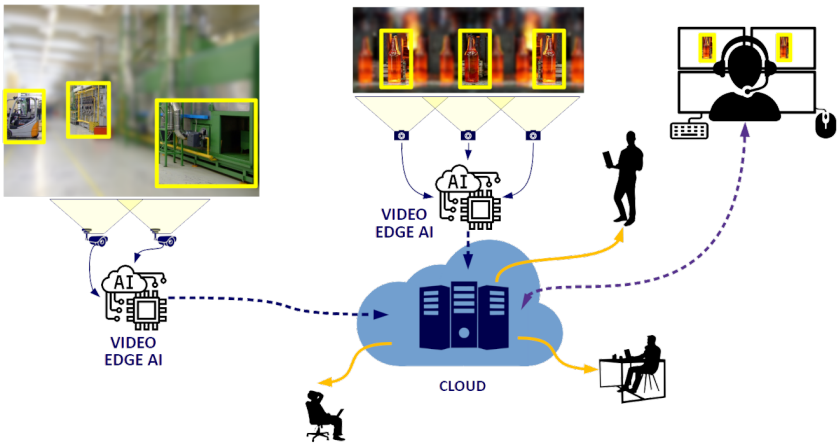

Figure 1.1: *A diagram illustrating the system architecture of VIEXPAND AI (source: Twevo S.A.)*

software stack into Docker images and deployment manifests to ensure reproducible builds, simplified distribution across development and test environments.

3. **Messaging backbone design.** Analysis, redesign of Kafka deployment and client configuration for the project architecture, including topic/partition layouts, producer/consumer tuning for low latency, and development of a lightweight librdkafka wrapper to aid deterministic benchmarking and integration.
4. **Actuator / controller (order translation) component.** Implementation of a deterministic perception-to-action controller that receives vendor-agnostic orders and translates them into machine executable instructions.

1.4 Objectives

The work was guided by three specific objectives:

1. Design, implement and validate a set of infrastructure components — covering data persistence, inter-service messaging, and application containerisation — that reduce end-to-end latency and improve the maintainability and deployment reproducibility of the *VIEXPAND AI* platform.
2. Implement a perception-to-action translation layer capable of actuating physical devices deterministically and with clear fault-attribution boundaries between software defects and operational misconfigurations.
3. Empirically characterise the latency, correctness and resource behaviour of each artefact on representative edge hardware, and derive actionable configuration recommendations from the results.

1.5 Research Approach

The work followed an applied engineering methodology, scoped and prioritised in close collaboration with the company's technical leadership. Operational problems were identified through direct engagement with the existing platform and its production constraints; requirements were derived from those problems in alignment with the company's development roadmap. Each assigned workstream then proceeded through iterative design and implementation of a concrete artefact, followed by targeted technical validation through measurement and experimentation. The full methodological framework, including the Design Science Research (DSR) model used to structure the work and the specific validation procedures, is described in Chapter 2.

1.6 Main Contributions

The principal contributions of this work are:

- A benchmarked and integrated SQLite-based storage subsystem for high-volume edge data pipelines, with empirically tuned journaling and concurrency configurations sustaining reliable operation at 3 million rows.
- A containerised deployment model with secure remote-desktop GUI exposure, demonstrated to introduce no measurable overhead above the precision of the adopted method on edge-class hardware.
- Kafka producer and consumer configuration research and tuning for edge deployments, with resolution of integration challenges in the librdkafka C++ wrapper and actionable configuration recommendations.
- A deterministic, declaratively-configured perception-to-action controller validated against an actuator across three power-supply configurations (Power over Ethernet (PoE), 12 V and 24 V configurations), with zero misfires recorded, and one (9 V / 0.6 A) that was identified as underpowered.

1.7 Document Structure

The remainder of this document is structured as follows:

Chapter 2 — Methodology presents the research paradigm adopted for this work, namely a condensed adaptation of the DSR framework. It maps each of the four workstreams to its originating problem, the derived requirement, the implemented artefact, and the validation method employed. It also describes the two measurement procedures used in the evaluations: the camera-to-display latency test and the actuator end-

to-end latency test bench.

Chapter 3 — State of the Art surveys the technical context relevant to the four workstreams. It covers wired communication protocols (fieldbus systems, Remote Direct Memory Access (RDMA), Ethernet variants, and streaming protocols), application-layer messaging systems (MQTT, RabbitMQ, and Apache Kafka), and lightweight storage solutions, positioning the design choices made in Chapter 4 against the available alternatives.

Chapter 4 — Design, Implementation and Evaluation constitutes the main body of the technical work. Each of the four workstreams is presented in a dedicated section following a common structure: design rationale, implementation details, and empirical evaluation. The sections cover the SQLite-based storage subsystem, the containerised Graphical User Interface (GUI) deployment, the Kafka messaging backbone, and the actuator/controller translation component.

Chapter 5 — Conclusion revisits the three objectives stated in this chapter, maps them to the evidence produced in Chapter 4, and consolidates the acknowledged limitations alongside directions for future work.

2 Methodology

2.1 Research Paradigm

This research adopts the DSR paradigm to address the challenges of designing and implementing a distributed flexible edge architecture for real-time computer vision applications. The canonical DSR process model proposed by [6] prescribes six sequential activities: Problem Identification & Motivation, Objectives of a Solution, Design & Development, Demonstration, Evaluation, and Communication. Because the work reported here was conducted within an industrial internship with fixed delivery milestones and an existing product roadmap, a strict linear traversal of all six activities was neither practical nor necessary. Instead, this study follows a condensed adaptation in which Problem Identification and Objectives were addressed jointly during the initial scoping of each workstream, Design & Development and Demonstration were interleaved through iterative prototyping cycles driven by the company's sprint cadence, and Evaluation was carried out via the targeted experiments described in Section Camera-to-Display Latency Measurement and Section Actuator End-to-End Latency Measurement. Communication is fulfilled by this Internship Report itself.

Each workstream is mapped in Table 2.1 to its originating problem, derived requirement, implemented artefact, validation method, and the section in Chapter 4 where the evidence is presented.

2.2 Methodology Development

The methodology for this research was developed through a combination of literature review, empirical experimentation, and iterative design.

Table 2.1: *Operationalisation of the DSR framework across the four workstreams.*

Workstream	Problem	Requirement	Artefact	Validation Method	Evidence
Storage subsystem	Existing TCP-based logging lacked transactional guarantees, complex querying, and degraded at scale.	Low-resource embedded database sustaining millions of rows with concurrent read/write under real-time constraints.	SQLite3 handler with WAL journal mode, worker-thread concurrency model, and workload-aware cache tuning.	Production-representative benchmarks: read latency under three configurations, write latency distribution and concurrency comparison under concurrent load, memory footprint (Section 4.1.3).	§4.1
GUI containerisation	Growing dependency complexity made reproducible deployment difficult; GUI exposure required isolation and security.	Reproducible, isolated container image with secure remote GUI access and negligible latency overhead.	Docker image with an embedded remote-desktop display layer (VNC).	Comparative camera-to-display latency test: bare-metal vs. containerised (Section 2.3.1).	§4.2
Messaging backbone	Custom TCP socket layer suffered zombie connections, lacked back-pressure, and imposed high extension cost.	Durable, horizontally scalable messaging with replay capability and low producer latency.	Apache Kafka deployment with a C++ `librdkafka` wrapper for publish/subscribe.	Broker saturation tests, configuration sensitivity analysis, and realistic-load latency profiling.	§4.3
Actuator / controller	Risk of hardware damage from incorrect actuation; need for clear fault attribution between code and configuration.	Deterministic command dispatch within 100 ms; declarative instruction definitions for auditability.	Microservice with YAML-based semantic parser and pluggable protocol adapter.	30-channel functional validation and end-to-end latency tests under four power-supply configurations (Section 2.3.1).	§4.4

Within the proposed architecture, two existing components were roughly adapted to integrate with the distributed framework, while the central messaging backbone—implemented using Kafka—served as the primary focus for variable-level analysis. Complementing these elements, an additional service was developed to fulfill a functional requirement.

The artifacts were evaluated through a combination of performance testing, latency measurements, and resource utilization analysis in a controlled environment that simulates real-world manufacturing workloads. New processes were introduced to the development workflow and deployment pipelines to facilitate continuous integration and delivery of the distributed architecture, while ensuring that latency and performance metrics were monitored and maintained throughout the software lifecycle.

2.3 Validation, Methods and Tools used

2.3.1 Methods

Camera-to-Display Latency Measurement

A recurring need throughout the internship was to quantify the end-to-end latency from the moment a scene change occurs in front of the camera to the moment the corresponding frame is rendered on the operator's display inside the *VIEXPAND AI* Monitoring Control Center (MCC) application. The following procedure was adopted:

1. A physical camera is aimed at a monitor that displays a continuously updating clock from a web-based time source (`clock.zone`).
2. The MCC application ingests the camera feed, processes it through its pipeline, and renders the resulting view in its own window on the same or a nearby monitor.
3. A screenshot is taken of the full display, capturing two timestamps simultaneously: the live clock shown on the website and the delayed clock visible inside the MCC application's camera view.
4. The latency is obtained by subtracting the timestamp rendered inside the application from the timestamp shown on the website at the moment of capture.

This method captures the entire chain: scene acquisition by the camera sensor, encoding and transmission by the camera hardware, decoding and processing by the software pipeline, and final rasterisation in the application window.

Actuator End-to-End Latency Measurement

To validate the actuator component against its 100 ms determinism requirement, a dedicated test bench was assembled around an actuator module. A host controller running Ubuntu 22.04 (AMD Ryzen 5 7600X, 32 GB Random Access Memory (RAM)) issued its fieldbus commands to the actuator module over a Cat5e Cable routed through a Teltonika Rutx08 router and TSW202 managed switch, and logged the transmission timestamp for each command. An independent ATmega2560-based microcontroller monitored the state of all 30 actuator channels by reading the state of each channel, logging every state transition with a timestamp derived from the Arduino `millis()` function.

The test comprised two phases:

1. **Sequential actuation.** Each channel was activated individually while the previously active channel was deactivated, ensuring that only one channel was in a ON state at any given time.
2. **Sweeping actuation.** All channels were activated one by one without deactivating the preceding ones; once all 30 were energized, they were deactivated sequentially in reverse order. This phase was designed to detect misfires or anomalies under overlapping-state conditions.

Both phases were repeated under four power-supply configurations —PoE, 9 V at 0.6 A, 12 V at 5 A, and 24 V at 5 A—to assess the actuator module's behaviour across its operating voltage range. All General-Purpose Input/Output (GPIO) states were recorded during execution to enable post-hoc misfire detection, and network traffic was captured with Wireshark for troubleshooting purposes. The software stack for the host side comprised multiple open source tools for both the operation itself and auxiliary monitoring(to check the state of each address on the fieldbus network), and PlatformIO 4.3.4 for microcontroller firmware development.

It should be noted that the microcontroller's polling loop yields an effective sampling resolution of approximately 25 ms, which bounds the precision of the actuation-latency measurements. The latency measurements are therefore subject to a granularity of 25 ms, and any latency differences smaller than this threshold cannot be reliably distinguished from measurement noise.

2.3.2 Tools

The work in this Internship Report spanned several concerns—implementation, testing, documentation, and deployment—each of which motivated a distinct set of tools. Development relied on containerization (Docker, with NVIDIA Container Toolkit for

Graphics Processing Unit (GPU) support) to ensure a reproducible and portable build environment across the heterogeneous machines used for integration testing (Appendix C), alongside standard build systems (CMake, Meson, Ninja) and compilers targeting both `aarch64` and `x86_64`. Testing made use of framework-specific suites—QTest for QT-based components and GoogleTest for C++ modules—chosen for their tight integration with the respective codebases rather than a one-size-fits-all approach. Documentation combined LaTeX for this report with Doxygen and Sphinx for API-level and user-facing documentation, respectively, and Excalidraw for diagramming. Deployment and ongoing housekeeping were automated using Ansible and Cron to reduce manual intervention and configuration drift across machines. In addition, Large Language Model (LLM)-based tools (e.g., GitHub Copilot, Gemini and ChatGPT) were used throughout to reduce Time to Market (TTM), primarily for boilerplate code generation, documentation drafting, and test-case scaffolding, with all output reviewed and validated manually before integration. A complete, versioned inventory of tools is provided in Appendix B for reference.

2.4 Summary

This chapter established the methodological foundations for the research. The work follows a condensed adaptation of the Design Science Research paradigm, in which Problem Identification and Objectives were addressed during the initial scoping of each workstream, Design & Development and Demonstration were interleaved through iterative prototyping cycles, and Evaluation was carried out through targeted experiments.

Four workstreams were operationalised against the DSR framework — the storage subsystem, GUI containerisation, messaging backbone, and actuator/controller — each with a clearly defined problem, derived requirement, implemented artefact, and validation method.

The validation strategy combines two complementary measurement procedures: a camera-to-display latency method that captures the full acquisition-to-render chain for the containerised GUI, and an actuator end-to-end latency method that employs an independent microcontroller-based observer to characterise command-dispatch-to-state-change timing across multiple power-supply configurations. Both methods, together with their precision limitations, provide the empirical basis for the evaluation results presented in Chapter 4.

3

State of the Art

3.1 Wired Communication

3.1.1 Fieldbus, Real time Control systems

Ethercat

Ethercat is a real-time Ethernet fieldbus system used primarily in industrial automation and control applications. It was designed to provide high-speed, low bandwidth, deterministic communication between devices, typically sensors, actuators, and controllers. It operates on mostly on the first, second and third layer of the Open Systems Interconnection (OSI) model, using some of the existing Ethernet technology to achieve low latency while maintaining a relatively high throughput, especially when comparing to other fieldbus technologies, it does so by using a master-slave architecture, reducing software overhead by embedding the process data directly into the Ethernet frame which is one of the causes of latency on Transmission Control Protocol/Internet Protocol (TCP/IP). Although Ethercat mostly works with 100mbit/s Ethernet connections, but recently some hardware implementations that work with 1Gbit/s and 10Gbit/s Ethernet connections have been used for Advanced Driver-Assistance Systems (ADAS) and Computer Vision applications and solutions It impressively achieves a cycle time of less than 25uS with 25 nodes on a 1Gbps connection [7]. One common topology is the ring topology, as shown in Figure 3.1.

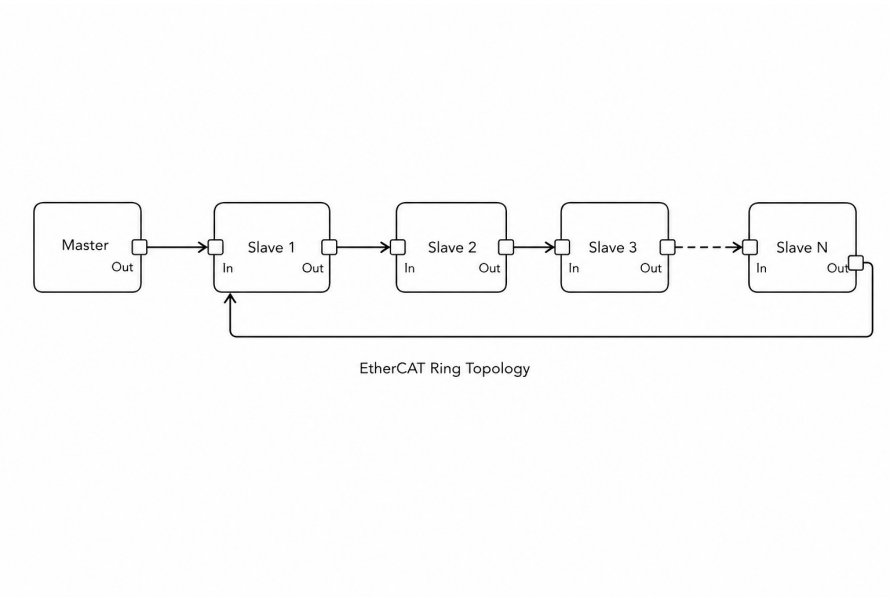

Figure 3.1: *An example of an Ethercat layout.*

Profinet

Profinet is an open industrial Ethernet standard developed by the Profibus and Profinet International (Profinet IRT (PI)) organisation for real-time communication in factory automation. It defines three performance classes: Profinet Real-Time (RT) uses standard Ethernet switching with software-based prioritisation to achieve cycle times in the low-millisecond range; Profinet Isochronous Real-Time (IRT) (Isochronous Real-Time) reserves dedicated time slots in the switch hardware, reaching cycle times below 1 ms with sub-microsecond jitter; and a non-real-time channel carries parameterisation, diagnostics, and IT traffic over standard TCP/IP.

Unlike EtherCAT's master–slave topology, Profinet allows arbitrary Ethernet topologies (line, star, ring) and coexists with conventional IT traffic on the same physical network. Integration with the PROFINET IO device model permits hot-swapping and automatic device replacement via a topology-aware naming scheme. These properties make Profinet particularly suited to large, heterogeneous plants where different subsystems must share a single Ethernet infrastructure while still meeting deterministic timing constraints.

CanBus2.0 / CanFD / CanXL

Can bus was developed by Bosch in the 1980s for automotive applications. It is a multi-master, message-oriented protocol that allows multiple devices to communicate with each other over a single twisted-pair cable. It is widely used in automotive and industrial applications, and has been adopted as a standard by the International Organiza-

tion for Standardization (International Organization for Standardization (ISO)) as ISO 11898. The original Can Bus standard, known as Can 2.0, supports data rates of up to 1 Mbps and has a maximum message length of 8 bytes. Some revision of the standard, such as Can FD (Flexible Data-rate) and Can XL (Extended Length), have been developed to support higher data rates and longer message lengths to accommodate the increasing complexity of modern automotive and industrial systems especially in the automotive sector. Maximum Latency of 54uS [8].

3.1.2 RDMA

Remote Direct Memory Access (RDMA) is a class of network operations that permit one host to read from or write to the memory of a remote host without involving the remote CPU or operating-system kernel in the data path. By bypassing the traditional socket stack, RDMA eliminates per-packet system-call overhead and memory copies, yielding application-to-application latency improvements. Three transport families currently expose RDMA semantics to applications through a common software interface (the OpenFabrics Alliance `libibverbs/rdma-core` stack): InfiniBand, RDMA over Converged Ethernet (RoCE), and Internet Wide Area RDMA Protocol (iWARP).

InfiniBand

InfiniBand is a switched-fabric interconnect originally specified by the InfiniBand Trade Association for high-performance computing and data-centre environments. It defines its own physical, link, network, and transport layers, operating independently of Ethernet. Current High Data Rate (HDR) ports deliver 200 Gbit/s per link, while the newer Next Data Rate (NDR) generation reaches 400 Gbit/s, with end-to-end latencies typically below 1 μs. InfiniBand natively supports RDMA verbs (send/receive, read, write, atomic), and its credit-based flow control and hardware-level congestion management provide lossless, deterministic transport without relying on Ethernet flow-control mechanisms.

The main drawback of InfiniBand is that it requires a dedicated fabric—switches, cables, and host-channel adapters—that is separate from the commodity Ethernet infrastructure already present in most organisations. This makes it an excellent fit for tightly coupled High-Performance Computing (HPC) clusters and AI training farms where latency and bandwidth density justify the additional capital expenditure, but less attractive for environments that must reuse existing Ethernet investments.

RoCE

RoCE (RDMA over Converged Ethernet) brings InfiniBand-class RDMA semantics to standard Ethernet networks. Version 1 (RoCEv1) encapsulates InfiniBand transport frames inside Ethernet, limiting RDMA traffic to a single Layer-2 broadcast domain. Version 2 (RoCEv2) wraps the same payload in User Datagram Protocol (UDP)/Internet Protocol (IP), making it routable across Layer-3 boundaries and compatible with conventional data-centre routing fabrics. Because RoCEv2 relies on a lossless Ethernet substrate, it is typically deployed together with Priority Flow Control (Priority Flow Control (PFC)) or Explicit Congestion Notification (Explicit Congestion Notification (ECN)) to prevent packet drops that would otherwise stall the RDMA transport.

RoCE is the most widely adopted RDMA transport in modern cloud and enterprise data centres because it reuses commodity Ethernet switches and Network Interface Controller (NIC)s (with RDMA-capable firmware), avoiding the cost of a parallel InfiniBand fabric. Latencies are marginally higher than native InfiniBand—typically 1–2 μs—but remain one to two orders of magnitude below kernel-based TCP/IP, making RoCE a practical choice for distributed storage, database replication, and latency-sensitive middleware.

iWARP

iWARP (Internet Wide Area RDMA Protocol) implements RDMA on top of TCP/IP (or, optionally, Stream Control Transmission Protocol (SCTP)), as specified in a family of Internet Engineering Task Force (IETF) Request for Comments (RFC)s (5040–5044, 6580–6581). Because TCP already provides reliable, ordered delivery with built-in congestion control, iWARP does not require the lossless Ethernet fabric that RoCE depends on; it can therefore operate over any routed IP network, including links with non-trivial packet loss.

In practice, iWARP trades some latency for deployment flexibility: typical round-trip times are in the range of 2–5 μs, higher than InfiniBand or RoCE, because TCP segmentation, acknowledgement, and congestion-window management add processing steps. Nevertheless, iWARP is valuable in environments where deploying PFC-capable switches is impractical or where RDMA traffic must traverse wide-area or heterogeneous networks that cannot guarantee lossless behaviour.

3.1.3 Ethernet

TCP/IP

TCP is widely used in many consumer and industrial applications, including web browsing, email, file transfer, and streaming media. Various RFCs have been published to define the protocol and its behavior, and due to its widespread use and popularity, it enables cheaper solutions when it comes to low latency and high throughput applications. There are a few tweaks that can be done to the TCP stack to reduce latency, as although TCP has gone through some improvements, along with some ways to reduce latency. * TCP FastOpening TCP Fast Open (TCP Fast Open (TFO)) is a TCP extension that allows data to be sent in the initial SYN packet during the TCP handshake, reducing latency for subsequent connections, it was introduced in RFC 7413 in 2014.

* Nagle's Algorithm Nagle's Algorithm is a TCP optimization technique that reduces the number of small packets sent over the network by combining multiple small messages into a single larger packet. This can help reduce network congestion and improve overall throughput, but it can also introduce latency in certain scenarios, especially for applications that require low-latency communication.

UDP

The User Datagram Protocol (UDP) is a connectionless, best-effort transport defined in RFC 768. It adds only a thin multiplexing and optional checksum layer on top of IP, imposing neither handshake setup, retransmission, nor in-order delivery guarantees. This minimal overhead makes UDP the preferred substrate for latency-sensitive applications—real-time audio/video, online gaming, Domain Name System (DNS) queries, and industrial telemetry—where occasional packet loss is preferable to the variable delays introduced by TCP's retransmission and congestion-control machinery.

More recently, UDP has become the foundation for Quick UDP Internet Connections (QUIC) (RFC 9000), a multiplexed transport protocol originally developed by Google and now standardised by the IETF. QUIC combines Transport Layer Security (TLS) 1.3 encryption, stream-level flow control, and connection migration into a single user-space protocol running over UDP, achieving 0-RTT or 1-RTT connection establishment and eliminating the head-of-line blocking that affects Hypertext Transfer Protocol (HTTP)/2 over TCP. QUIC's design demonstrates that the simplicity of UDP can serve as a building block for sophisticated, secure transports without sacrificing the low-latency properties of the underlying datagram service.

RTSP

The Real-Time Streaming Protocol (Real Time Streaming Protocol (RTSP)) is an application-level control protocol that governs the delivery of continuous media streams between a server and one or more clients. In contrast to HTTP, which is a stateless, pull-oriented protocol, RTSP maintains a persistent control session and exposes a set of VCR-like methods — `DESCRIBE`, `SETUP`, `PLAY`, `PAUSE`, and `TEARDOWN` — that allow a client to initiate, control, and terminate a streaming session. RTSP itself carries no media data; it merely negotiates and controls the session, while the actual media payload is transported by the Real-time Transport Protocol (Real-time Transport Protocol (RTP), RFC 3550) over UDP (or TCP in Network Address Translation (NAT)-traversal scenarios), with quality feedback provided by the RTP Control Protocol (RTP Control Protocol (RTCP)).

This separation of control and data planes is intentional: RTP over UDP minimises framing overhead and exploits the low, consistent latency of the datagram path, while RTCP allows the receiver to report packet loss, jitter, and round-trip delay back to the sender for adaptive bitrate control. Glass-to-glass latency for a well-tuned RTSP/RTP pipeline can reach below 200 ms with small RTP payload sizes and reduced receiver-side buffering, making it substantially more responsive than HLS or DASH for live streaming scenarios.

In industrial and manufacturing edge environments, RTSP is the dominant streaming interface exposed by IP cameras, network video recorders (Network Video Recorder (NVR)s), and Open Network Video Interface Forum (ONVIF)-compliant devices. Its widespread hardware support means that a computer vision acquisition pipeline can ingest frames directly from commodity surveillance cameras through standard `GStreamer` or `FFmpeg` pipelines without proprietary Software Development Kit (SDK)s, reducing integration complexity. The primary drawback of RTSP in modern network environments is its limited NAT and firewall traversal, a problem partially addressed by RTSP over HTTP tunnelling or by the more recent WebRTC stack, which replaces the RTSP/RTP/RTCP triplet with a browser-native signalling and transport layer.

3.2 Application Layer Messaging Protocols

This section surveys the principal application-layer messaging systems considered during the design of the communication backbone. Each protocol occupies a different point in the design space defined by delivery semantics, throughput, latency, and operational complexity; the summaries below highlight those trade-offs to motivate the selection made in Chapter 4.

3.2.1 MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe messaging protocol originally designed by IBM for telemetry over constrained, unreliable networks. It operates over TCP (and, since version 5.0, optionally over QUIC or WebSockets) and defines three quality-of-service levels: at most once (Quality of Service (QoS) 0), at least once (QoS 1), and exactly once (QoS 2). A central broker mediates all communication: publishers send messages to hierarchical topics, and the broker fans them out to every matching subscriber, optionally persisting messages for durable sessions.

MQTT's minimal fixed header (2 bytes), small code footprint, and keep-alive/will mechanism make it the de facto standard for IoT telemetry, sensor networks, and mobile push notifications. However, its single-broker architecture can become a throughput bottleneck under high-volume workloads, and the lack of native consumer-group semantics or persistent log replay limits its suitability for event-sourcing or stream-processing patterns.

Performance-wise, MQTT is characterized by extremely low end-to-end latency, often achieving sub-millisecond delivery in local edge environments[9]. Its minimal overhead allows it to handle high connection churn with significantly lower CPU and memory utilization than heavier protocols. However, latency is highly sensitive to the QoS level; transitioning from QoS 0 to QoS 2 can increase delivery time by over 50% due to the multi-step acknowledgment handshake required for message guarantees.

3.2.2 RabbitMQ

RabbitMQ is an open-source message broker that implements the Advanced Message Queuing Protocol (Advanced Message Queuing Protocol (AMQP) 0-9-1) and additionally supports MQTT, Simple Text Oriented Messaging Protocol (STOMP), and AMQP 1.0 via plug-ins. Its routing model is built around exchanges and bindings: producers publish to an exchange, which routes each message to one or more queues according to configurable rules (direct, topic, fanout, or headers matching). Consumers then pull or are pushed messages from those queues, with per-message acknowledgements enabling reliable, at-least-once delivery.

RabbitMQ excels at traditional task-queue and request/reply workloads where flexible routing, per-message Time to Live (TTL)s, dead-letter handling, and priority queues are required. It supports clustering for high availability and federation for geographic distribution. The main limitation relative to log-based brokers such as Kafka is that messages are deleted once acknowledged, precluding replay; and throughput, while adequate for most enterprise workloads, does not scale horizontally as linearly as a

partitioned commit log.

RabbitMQ provides stable throughput for complex routing but typically exhibits End-to-End (E2E) latencies approximately 2x to 5x higher than native MQTT brokers like Mosquitto [9].

> While it excels at vertical scaling, latency can spike under heavy memory pressure or when utilizing disk-backed persistent queues, making it better suited for reliable asynchronous task processing rather than ultra-low-latency telemetry often exceeding 500ms under high load. [10]

3.2.3 Kafka

Kafka is a distributed event streaming platform that is designed to handle high throughput and low latency messaging. Due to its properties, it provides a reliable and scalable solution for a real-time data streaming and processing, making it suitable for applications that require low latency communication. In this section we will explore further in depth Kafka and some tweaks along with some results with an real time Edge framework in mind.

Typical layouts for Kafka include a cluster of brokers (servers) that manage the storage and replication of messages, producers that publish messages to topics, and consumers that subscribe to those topics to receive messages, as seen in Figure 3.2. Kafka uses a pull-based model for consumers, which allows them to control the rate at which they consume messages and provides better flow control compared to push-based models.

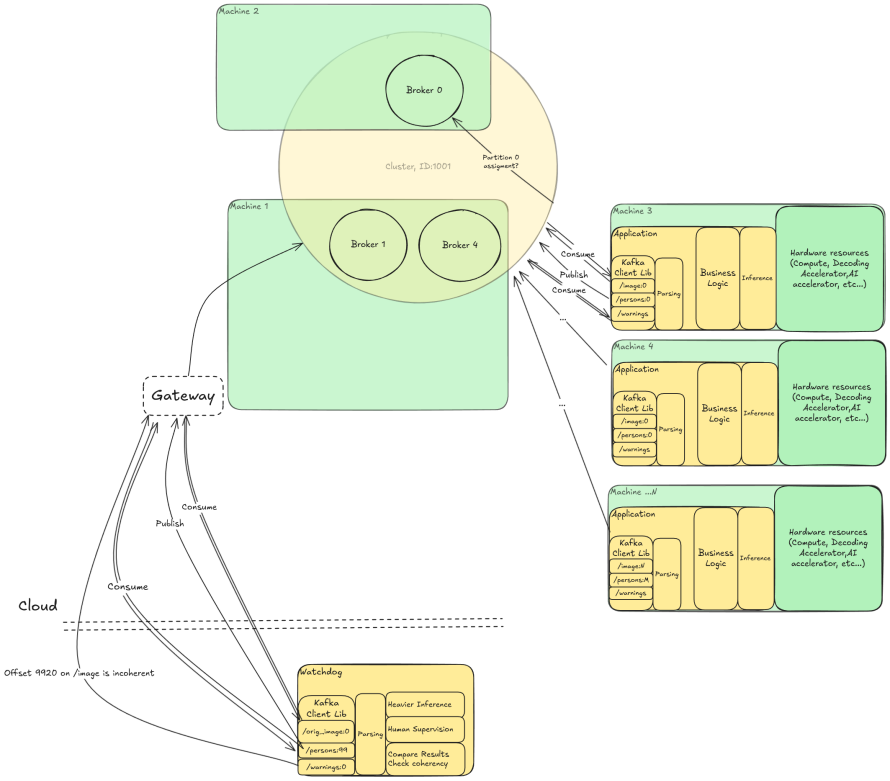

Figure 3.2: *An example of a Kafka layout.*

There are several implementations of the Kafka client in different programming languages, however we will focus more on the C++ variant due to its performance and suitability for real-time edge applications.

Kafka is flexible however it cannot fulfill every scenario and it requires tuning from the user to achieve the best performance latency wise, for example according to [11], end to end latencies can be dropped right up to two milliseconds or fifteen milliseconds depending on the acknowledgment configuration. The three candidates are compared side-by-side in Table 3.1.

3.2.4 Comparison Table

Each messaging solution occupies a distinct point in the latency–feature–complexity space. MQTT prioritises minimal overhead and is well-suited for constrained devices; RabbitMQ offers mature queue-based routing; and Kafka provides durable log-based storage with replay capabilities.

Table 3.1: *Comparison of MQTT, RabbitMQ and Kafka*

Criterion	MQTT	RabbitMQ	Kafka
Latency	sub-ms–5 ms	sub-ms–10 ms	2–15 ms
Replay	Not native	Not native	Excellent (log retention)
Consumer groups	Not supported	Supported (queues)	Advanced native support
Complexity	Low	Medium	High
C++ Compatibility	Excellent (Paho)	Good	Good (librdkafka)
Integrity	High (QoS 0–2)	High (acks/durability)	High (acks=all, replication)
Endurance	Good (lightweight)	Good	Excellent (distributed)

3.3 Low Latency Storage Solutions

3.3.1 Relational Databases

SQLite3

SQLite3 is an embedded, serverless, transactional Structured Query Language (SQL) database engine. Self-contained and requiring no separate server process or administration, it runs entirely within the host application's own address spac, a design decision that has made it the most widely deployed database engine in existence. Found in virtually every smartphone, web browser, operating system, and embedded device produced in the last two decades.

This in-process architecture is not merely a packaging convenience: it is a fundamental performance and integration advantage. By eliminating the inter-process communication that separates clients from server-based engines, SQLite can exploit page-cache locality and avoid serialisation costs, making it competitive with server-based solutions for local transactional workloads [12]. Performance is, however, highly configuration-dependent. Out of the box, SQLite prioritises safety over speed, and unlocking its low-latency potential requires deliberate tuning of the journal mode, page-cache size, and synchronisation policy. Benchmarking modern SQLite against representative transactional and analytical workloads demonstrates that targeted configuration and engine optimisations can deliver speedups of up to 4.2× on the Star Schema Benchmark relative to the default configuration [13].

The most consequential tuning lever for write-intensive workloads is the journal mode, which governs how the engine achieves atomicity and crash recovery. Three modes are available—the default rollback journal (`DELETE`), `WAL` (Write-Ahead Logging), and `MEMORY`—each occupying a different point in the latency–durability trade-off

space. The implications of this choice for a concurrent edge workload, and the mode ultimately adopted in this work, are discussed in Section 4.1.

PostgreSQL

PostgreSQL is an open-source, object-relational database management system (Relational Database Management System (RDBMS)) with over three decades of active development. Unlike most relational databases, PostgreSQL fully embraces extensibility as a core design principle: users can define custom data types, operators, index access methods, and procedural languages without modifying the server source code. This architecture enables a native ecosystem of extensions — PostGIS for geospatial data, TimescaleDB for time-series workloads, and pg_vector for embedding-based similarity search, among many others — allowing the engine to serve as a general-purpose data platform rather than a narrowly focused relational store.

At the engine level, PostgreSQL adopts a process-per-connection model and uses Multi-Version Concurrency Control (Multi-Version Concurrency Control (MVCC)) to provide full Atomicity, Consistency, Isolation, Durability (ACID) compliance without reader–writer blocking. Its cost-based query optimizer supports a broad range of join strategies (hash, merge, nested-loop), parallel query execution, and partial and expression indexes. Advanced features such as table partitioning, logical replication, Foreign Data Wrapper (FDW), and row-level security distinguish it from lighter-weight alternatives and position it as a strong candidate for systems that blend operational and analytical requirements, or that must enforce complex business rules at the database layer.

In contrast, MySQL (particularly its InnoDB storage engine) is optimised primarily for high-throughput simple transactional workloads. Its thread-based connection model, lighter-weight optimizer, and tightly coupled storage engine make it competitive in scenarios dominated by straightforward Create, Read, Update, Delete (CRUD) operations, at the cost of reduced flexibility and weaker support for complex query patterns or advanced data types. Both systems are SQL-compliant, open-source, and widely supported, making the choice between them fundamentally workload-driven rather than capability-driven.

Comparative benchmarks reinforce this distinction empirically. Evaluations using SysBench (simple Online Transaction Processing (OLTP)), TPC-C (complex OLTP), and TPC-H (analytical, 22 queries) against PostgreSQL 16 and MySQL 8.0 reveal a clear performance dichotomy: MySQL achieves up to 21% higher peak throughput under moderate concurrency for simple transactional workloads, but degrades under heavy client load, while PostgreSQL delivers greater stability and outperforms MySQL by approximately 14% on the complex TPC-C workload. The divergence is most pro-

nounced in the analytical case, where PostgreSQL completes the full TPC-H query suite in less than one-third of the time required by MySQL, a result attributable to its more sophisticated query optimizer and execution engine [14]. These findings confirm that the optimal choice is fundamentally workload-dependent: MySQL suits simple, high-volume read/write pipelines, while PostgreSQL is the more robust option wherever complex queries, mixed workloads, or predictable behavior under contention are required.

3.3.2 Non-relational Databases

Non-relational Databases refers to systems designed to hold and query data that lack a fixed relational schema, such as documents, logs, or graph data. Modern Not Only SQL (NoSQL) stores were developed to address the volume, variety, and velocity of such data that traditional relational systems struggle with.

NoSQL therefore emphasizes:

- Flexible schemas that accept evolving or heterogeneous records instead of rigid tables
- Specialized access patterns (e.g., key-value lookups, document retrieval, wide-column scans, graph traversals) rather than one generic relational access model
- Integration with distributed processing and analytics ecosystems to handle very large datasets and high request loads

3.4 Summary

This chapter surveyed the communication, messaging, and storage technologies relevant to the design of a distributed edge architecture for real-time computer vision.

At the transport level, deterministic fieldbuses such as EtherCAT and Profinet deliver sub-millisecond cycle times at the cost of dedicated hardware and rigid topologies, while RDMA-based fabrics (InfiniBand, RoCE, iWARP) achieve microsecond-class latency through kernel-bypass techniques. Standard Ethernet transports (TCP, UDP, RTSP) remain dominant at the edge due to commodity hardware availability and broad ecosystem support, albeit with higher and less predictable latency.

At the application-messaging layer, MQTT offers the lowest per-message overhead and sub-millisecond delivery for lightweight telemetry, but lacks consumer-group semantics and durable replay. RabbitMQ provides flexible routing and mature queue-based delivery, yet deletes messages upon acknowledgement, precluding retrospective auditing. Apache Kafka combines a partitioned, durable commit log with horizontally

scalable consumer groups and tuneable latency, making it the strongest candidate for workloads that require both low-latency delivery and message replay.

For local data persistence, SQLite offers an embedded, serverless architecture that eliminates inter-process communication overhead and delivers competitive transactional performance when properly configured (notably in WAL journal mode), while PostgreSQL excels at complex analytical queries and mixed workloads at the cost of a server process and higher operational complexity.

These findings directly inform the design decisions made in Chapter 4:

- **Storage subsystem.** SQLite was selected for its zero-administration, in-process architecture that matches the resource constraints and simplicity requirements of the edge deployment, with WAL mode adopted to reconcile write latency with crash-recovery guarantees.
- **Messaging backbone.** Apache Kafka was chosen for its durable log-based delivery, native consumer-group support, and proven horizontal scalability, addressing the replay and backpressure requirements that the legacy TCP-based layer could not satisfy.
- **GUI containerisation.** Docker-based containerisation with remote-desktop display was adopted for reproducible deployment, security isolation, and negligible latency overhead, using commodity Ethernet infrastructure.
- **Actuator / controller.** A pluggable protocol-adapter architecture was selected to decouple the semantic translation layer from any specific fieldbus or device protocol, allowing the system to accommodate future hardware interfaces without reimplementation.

4

Design, Implementation and Evaluation

This chapter presents the engineering work carried out during the internship at Twevo S.A. Each section corresponds to one of the workstreams introduced in Chapter 1 and follows a common structure: the operational requirements that motivated the work are stated first, followed by the design rationale, the implementation details, and, where applicable, empirical evaluation of the resulting artefact.

4.1 Lightweight Storage Subsystem

4.1.1 Design Rationale

An inquiry was made with other developers to help solidify the main tables and their schemas, as well as the expected query patterns and access frequencies, which informed the design of the storage subsystem. A special emphasis was placed on the statistics related schemas, as they were the ones who would be more affected by the increase in the number of records, and as such they were the ones that would require more attention when it came to performance and scalability. After a draft was made, iterations of the variables and which to normalize and which not were made, and the final first prototype schema.

After the database requirements were defined, a survey of potential solutions was conducted, including both embedded databases and lightweight server-based databases. The ones that came as potential candidates were SQLite, PostgreSQL, MySQL, InfluxDB. Key selection criteria included low resource consumption, ease of integration, simplicity of deployment, and proven performance for the expected workload.

4.1.2 Implementation

A database handler was implemented as a singleton to encapsulate the Qt SQL driver, as a way to both simplify and centralize connection lifecycle management and initialization error handling. To facilitate structured data access, a small performed of available Object Relational Mapping (ORM) libraries was conducted; the evaluated candidates were ODB, QxOrm, and TinyORM. The mandatory selection criteria were as follows:

- Support for SQLite3 as a backend.
- Compliance with the C++20 standard or later.
- Active maintenance and adequate availability of documentation and community support.
- Low operational resource footprint.
- Negligible latency overhead.
- Compatibility with the existing codebase, toolchain, and development practices.

The desirable secondary criteria were:

- Support for schema migrations.
- Support for multiple database backends (e.g., PostgreSQL, MySQL) to allow future extensibility.
- Integration with the existing logging and error-handling frameworks.
- Native Qt framework integration to minimise type conversion and boilerplate code.
- Non-intrusive, code-first schema migrations, enabling schema definition and propagation without manual intervention.
- Support for asynchronous database operations to preserve main-thread responsiveness in a real-time context.

However, given the more pressing engineering priorities and the constraints on time allocated to this component, a direct query-based approach was adopted instead of a ORM, and subsequently refined.

As the system developed around the component, further evaluations were conducted specifically in relation to indexation strategies, query pattern rewrites, and the impact of various SQLite configurations on latency and resource utilization. A particularly consequential decision was the choice of journal mode the mechanism SQLite uses to achieve atomicity and crash recovery, introduced in Section 4.1 of the State of the Art. Three candidate modes were evaluated before a selection was made:

`MEMORY` Stores the rollback journal entirely in RAM, eliminating all journal-related disk I/O and yielding the lowest possible write latency. However, it provides no crash-recovery guarantees; an abrupt power failure or kernel-level crash after a partial

write leaves the database in an inconsistent, unrecoverable state. This risk was deemed unacceptable for an edge device expected to operate continuously in an industrial environment.

`WAL` (**WAL**) Appends changes to a separate, sequential WAL file before committing them to the main database, allowing readers and a writer to proceed concurrently without blocking each other. Because commits are sequential appends rather than random in-place modifications, `WAL` substantially reduces per-transaction latency relative to the default rollback mode. Crash recovery remains safe: on restart, SQLite replays committed WAL frames and discards uncommitted ones.

`DELETE` (**default rollback journal**) Writes a copy of the original pages to a rollback journal before modifying them in place. Both the journal write and the main-file write must be flushed to disk before the commit can return, which serialises concurrent readers and writers and imposes the highest per-transaction latency of the three modes.

`WAL` mode was selected as it offers the best balance between commit latency and durability for a mixed read/write workload.

A complementary setting, `PRAGMA synchronous=NORMAL`, was retained throughout the deployment: it guarantees durability across normal application crashes while accepting a narrow unprotected window limited to the last unflushed WAL frame in the event of an abrupt Operating System (OS)-level failure—a risk profile consistent with the target deployment environment.

This mode carried some technical challenges, that showed themselves as the system aged. Initially several routines used multithreading to write and read from the database using native C++ concurrency primitives. However, as the feature set grew, challenges such as non-deterministic deadlocks, inefficient marshalling and object ownership issues that propagated between the QT environment, its events loops, routines and other software components started to become apparent during development and testing of new features. These issues became significant as they caused unpredictable behavior and were hard to debug.

As such, a strategic refactor was done to the way the concurrency was handled by enforcing better object ownership, and using a worker thread model instead of relying on native C++ concurrency primitives. Some of the ideas for the amelioration of the handler component, were taken from public resources [15], [16].

4.1.3 Evaluation

The storage subsystem was evaluated along two complementary axes that together delimit its operational envelope:

1. a stress test that exercises the analytical query path used by the reporting dashboard against a large, mostly cold dataset.
2. a real-time observation of every single SQLite operation executed by the application during normal streaming work, under live Camera-to-Display Latency Test Controller (CTC) traffic.

Both axes were exercised in turn under the two journal modes of interest, WAL and DELETE, with every other configuration parameter held constant (2 MB page cache, `synchronous=NORMAL`). The benchmark machine was equipped with an AMD Ryzen 5 7600X, 32 GB DDR5 RAM, and a 1 TB Non-Volatile Memory Express (NVMe) Solid State Drive (SSD).

Database Profile

The database used in every run was a production-trimmed copy whose static properties are summarised in Table 4.1.

Table 4.1: *Database profile common to all benchmark runs.*

Property	Value
Database size	1 192.8 MB
Page size	4 096 bytes
Page count	305 352
Freelist pages	0
Largest table (`section_stats`)	~3 009 000 rows

Although the database was trimmed before being used as a benchmark fixture, the largest table (`section_stats`) retained over three million rows. For comparison, that's about 2 months of data; an uncut database originated from a staging environment that's present on production holds Franken-stage environment holds 100+ million. Importantly, the dashboard queries that drive the stress test are parameterised to retrieve one full year back from the current moment. Since the trimmed dataset is only densely populated over a recent multi-month window, those queries traverse the entire table and aggregate over partially populated date ranges. The stress test should therefore be read as a worst-case upper bound on dashboard query cost, rather than as a representative steady-state figure.

Stress Test — Analytical Query Latency

The reporting view of the dashboard dispatches seven analytical queries to the worker thread pool concurrently and waits for the slowest to return before rendering. Two consecutive runs were executed within the same process for each journal mode, so that R1 reflects a cold OS page cache and R2 a warm one. Table 4.2 reports per-query

latency (the time each individual query took, irrespective of dispatch order) together with the end-to-end wall clock for each run.

Table 4.2: *Analytical query latency (ms) on the trimmed 1.19 GB database, two consecutive runs per journal mode.*

Query	Rows	WAL		DELETE	
		R1	R2	R1	R2
EfficiencyPerZone	4	1 697	986	1 005	974
EfficiencyPerSection	10	2 452	2 411	2 345	2 377
TimelinePerZone	337 360	2 936	2 823	2 865	2 848
UpTimePerSection	50	13	16	16	14
CycleTimes	3 009 235	1 878	1 608	1 639	1 624
AlarmsPerSectionZone	4	47	40	52	59
AlarmsBySectionId	44	65	66	82	67
Wall clock (end-to-end)		**2 988**	**2 940**	**2 942**	—

Three observations follow directly from the table.

Wall clock ≪ sum of per-query times. Summing the per-query latencies of WAL R1 yields approximately 9.1 s, while the reported end-to-end wall clock is 2.99 s. This confirms that the seven queries are dispatched concurrently across the worker thread pool and that the wall clock corresponds to the latency of the longest query plus orchestration overhead, not to a linear sum of the parts.

WAL and DELETE are indistinguishable for read-only workloads. With no concurrent writer, both modes complete the seven-query batch in 2.9–3.0 s. This is consistent with the architectural role of WAL : it eliminates reader–writer mutual blocking, but does not accelerate isolated reads.

Cold/warm cache effect is small. R2 latencies sit within 5–10 % of R1 in every meaningful case. The dominant cost of the heavy queries is full-table scanning of `section_stats`; once the working set exceeds the page-cache budget available to the process, the marginal value of warm-cache reuse becomes modest.

The second DELETE run reported an anomalous end-to-end wall clock of 1.26 s while the individual query times still summed to 2.92 s, an internally inconsistent reading attribeutable to a known timing-instrumentation defect. That single data point is excluded from the analysis; per-query latencies from the same run are consistent with R1 and are retained.

Real-Time Operation — Per-Operation Latency

To capture the persistence cost of normal operation (as opposed to dashboard stress), the application was instrumented to log the latency of every SQLite call it executed during a continuous streaming session against the live CTC feed. Two sessions were captured back to back, one per journal mode; each lasted approximately 5–6 minutes. The instrumentation distinguishes three operation classes:

- `read` — single-statement queries issued by the periodic polling pipeline.
- `db_op` — individual prepared-statement executions issued inside a batched write transaction.
- `write` — end-to-end transaction wrappers (begin → inserts → commit).

Tables 4.3, 4.4 and 4.5 summarise the captured distributions.

Table 4.3: *Read latency under live CTC traffic.*

Percentile	WAL (μs, n=542)	DELETE (μs, n=567)
Minimum	55	41
p50	137	157
p95	292	6 700
p99	3 338	70 462
Maximum	244 973	248 948
Mean	729	3 499

Table 4.4: *Per-statement (`db_op`) latency inside batched write transactions.*

Percentile	WAL (μs, n=3 640)	DELETE (μs, n=3 952)
Minimum	237	276
p50	1 154	1 339
p95	38 966	77 975
p99	42 118	129 030
Maximum	75 612	189 731
Mean	10 948	15 624

Table 4.5: *End-to-end transaction (`write`) latency.*

Percentile	WAL (μs, n=364)	DELETE (μs, n=413)
Minimum	41	40
p50	222	59 427
p95	2 192	109 666
p99	8 937	131 250
Maximum	30 567	135 960
Mean	875	44 837

Throughput is upstream-bound. Both sessions completed a comparable number of operations (WAL: 3 640 `db_op` + 364 `write`; DELETE: 3 952 + 413) over comparable ob-

servation windows of 324 s and 346 s respectively. This is expected: the rate at which the storage subsystem is asked to do work is set by the CTC producer, not by the database, so neither mode is throughput-starved here. The interesting comparison is therefore not how many operations were served, but how long each one took.

Reads under concurrent writes. The medians of `read` latency are nearly identical across modes (137 µs vs. 157 µs); reads themselves do not depend on the journal mode. The tails, however, diverge sharply: WAL keeps the p99 at 3.3 ms, while DELETE's p99 climbs to 70.5 ms. The difference is entirely attributable to readers being blocked while DELETE rolls back its journal after each commit; under WAL , readers and writers are decoupled and no comparable stalls appear. The single near-quarter-second outlier present in both sessions (~245–249 ms) corresponds to the schema and index load that follows database open and is not representative of steady-state behaviour.

Transactions.
`begin--insert--commit` takes 222 µs under WAL and 59.4 ms under DELETE—two orders apart. The DELETE figure is dominated by exclusive-lock acquisition and the synchronous flush of the rollback journal at commit time; WAL appends commits to the log file and defers checkpointing, so the per-commit cost is small and uniform. The same pattern carries through to the tail: WAL 's p99 is 8.9 ms versus 131 ms for DELETE.

Per-statement cost inside a transaction. Individual prepared-statement executions (`db_op`) are also slower under DELETE (p99 of 129 ms vs. 42 ms under WAL). Inside a batch the lock-acquisition cost is amortised, so the gap is narrower than at the transaction level, but it is still present in the tail because the worker thread holding the write lock must serialise its statements against any reader that arrives mid-transaction.

Memory Footprint

Resident Set Size (RSS) was sampled alongside every recorded operation throughout both real-time sessions; the resulting trajectories are summarised in Table 4.6.

Table 4.6: *Process RSS at key lifecycle points during the real-time sessions.*

Phase	WAL (MB)	DELETE (MB)
Application startup (baseline)	65	65
After DB open + schema/index load	161	162
After first sustained write activity	163	165
Peak during steady-state operation	596	599

The two sessions are practically indistinguishable in memory profile: both start at ~65 MB, climb to ~160 MB after schema and index load, and reach a peak of approximately 600 MB at full operating load. The peak corresponds to the worker thread pool's accumulated result-set buffers and per-connection SQLite caches; it is independent of the journal mode. WAL therefore does not impose any measurable memory penalty over DELETE in the configuration used here.

UI Rendering

Across both stress-test runs, total User Interface (UI) rendering time remained between 16 and 20 ms, less than 1 % of the end-to-end wall clock. The rendering layer is therefore not a bottleneck and is excluded from further analysis.

Summary

The evaluation supports the following conclusions.

1. The storage subsystem sustains reliable operation against a 1.19 GB, 3 M-row database, meeting the resilience-at-scale requirement, with the dashboard's worst-case full-history aggregate queries completing in approximately 3 s of wall clock.
2. For isolated read-only workloads, WAL and DELETE are indistinguishable; the choice between them is irrelevant when no writer is active.
3. Under realistic concurrent operation, WAL provides a substantial latency advantage: median transaction latency improves by roughly two orders of magnitude (222 µs vs. 59 ms), and the p99 read tail under concurrent write load improves by roughly 21× (3.3 ms vs. 70 ms).
4. Throughput is comparable between the two modes because the workload is bound by the upstream CTC feed, not by the database.
5. The memory footprint is essentially identical between the two modes (~600 MB peak, with a ~160 MB floor after schema load); the choice of journal mode does not impose a memory penalty.

Threats to validity. The real-time sessions were driven by a live, unsupervised CTC feed running on a separate machine on the local network. Two factors limit the reproducibility of the absolute numbers: the inter-machine network path introduces jitter on inbound events, and the upstream inference pipeline is itself non-deterministic — minor variations in colour thresholding can change the number and timing of events emitted from the same source video. The two journal-mode sessions were therefore not exposed to byte-identical workloads, although their observation windows and total operation counts are within ~10 % of one another, supporting the claim that the

comparisons reported above are representative rather than coincidental. The stress-test queries deliberately request a one-year window against a database that is only densely populated over a recent multi-month range; the resulting latencies should be read as a worst-case ceiling rather than as a typical figure. Finally, the benchmark machine is a desktop-class workstation rather than the edge-class hardware on which the platform is ultimately deployed, so the absolute numbers should be regarded as relative comparators between configurations rather than as deployment-grade specifications.

4.2 Secure GUI Exposure through Containerisation

As the *VIEXPAND AI* platform evolved, the amount of features, dependencies and complexity of the supporting ecosystem grew, and as such a need for a way to be able to replicate the environment in a effective way arose. There was two general paths to solve the problem:

- **Virtualisation.** Using a hypervisor to create a virtual machine that could run the entire operating system and the application, providing a high level of isolation and compatibility, but with a higher resource overhead and complexity.
- **Containerisation.** Using a container runtime to create lightweight, isolated environments that share the host operating system kernel, providing a more efficient and portable solution, but with some limitations in terms of isolation and compatibility.

4.2.1 Design Rationale

Given the resource constraints, and the restrictions imposed by the hardware and cost considerations, virtualisation was deemed to be an unfeasible solution, as some of the components were already containerised and would require a significant effort just to be able to run in a virtualised setup. Containerisation on the other hand, provided a more efficient and portable solution, that could be easily integrated with the existing development and deployment workflows, and as such it was the chosen path to solve the problem. However the containerisation of the GUI aspect offered some unique challenges, in terms of performance and security, as the GUI needed to be able to access the host's display server in an hands free approach, while also ensuring that the containerised application did not have unnecessary access to the host system.

4.2.2 Implementation

The implementation itself was done through several iterations, and deployments over time in order to minimize interference with the production environment, to allow un-

expected issues to be traced more easily, and lastly, to facilitate comprehensive knowledge transfer and ensure team members remained aligned with evolving procedural frameworks. The first iteration was getting the program to run in a container, which was done by slowly containerising the dependencies, and the application itself.

As soon as the program started to indicate it was able to execute for a few seconds, the next step was to get the GUI to work. Two choices were considered: using display server forwarding, or creating a whole new display server and using a remote desktop protocol to access it.

The pros and cons of each approach were evaluated, but it was found that the remote desktop protocol approach provided better flexibility, isolation and security, as it allowed for better isolation between the container and the host system, while also making it easier to deploy and test the environment as it required to run without a desktop graphical environment, which was a requirement for the production environment. It also carried the benefit of being easier to manage the display server, as when it failed, it would not affect the host system, and it would be easier to restart and troubleshoot.

A schema of the architecture can be seen in Figure 4.1.

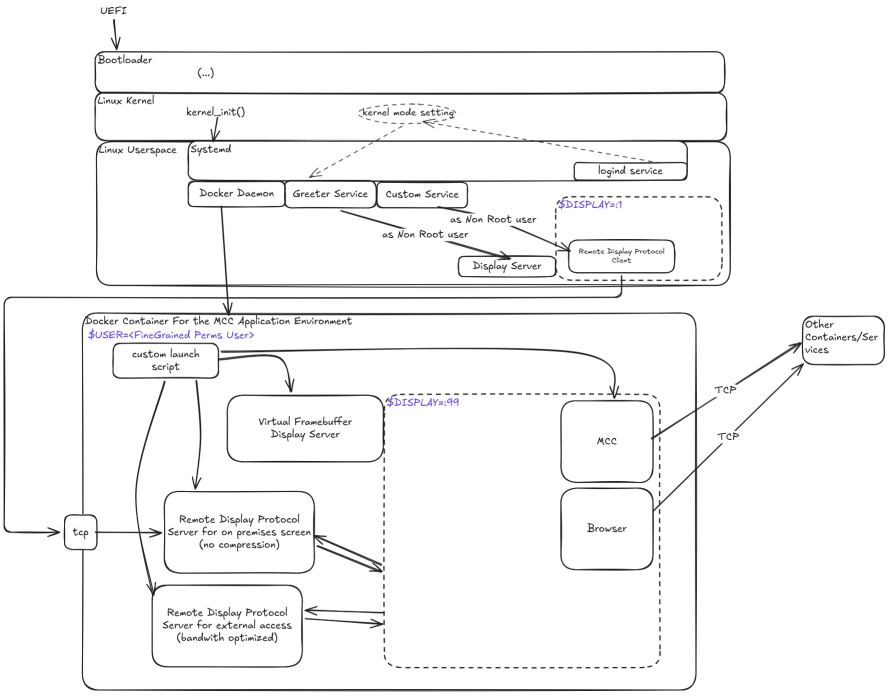

Figure 4.1: *A simplified view of the aforementioned system architecture with remote desktop integration.*

Another aspect that was evaluated was the removing capabilities that weren't necessary for the application to run, as a way to further improve security and isolation, and to minimize the attack surface of the containerized application. As per a last request of team members, a script that is colloquially known as *hypervisor* was retrofitted to be able to orchestrate the launch, resource supervision and telemetry of the containerized application and its environment. Lastly, the differentiation in the development

and production environments was also created. The main divergence between them, excluding the binary and tools available, was in the way the necessary artifacts were mounted in the container.

As expected this work was a iterative process, and as such, versioning and knowledge transfer were key aspects to ensure that the team was able to keep up with the changes, and to be able to maintain the environment in the future, as well as to be able to troubleshoot any issues that may arise. A lot of dependencies changed aggressively whether it was the way the application was built, installed, updated from the ppa, patched via artifacts, etc...

4.2.3 Evaluation

Since the primary objective of this workstream was to produce a reproducible, self-contained container image capable of running the full graphical application, the resulting Docker image itself constitutes the principal artefact. The image's correctness was verified by deploying it to the target environment and confirming that the application launched, rendered its GUI through the remote-desktop path, and operated without regressions. The specifics of the image and its build process cannot be disclosed due to intellectual-property constraints; however, Figure 4.2 shows the application running inside the container in a production environment, serving as proof of artefact.

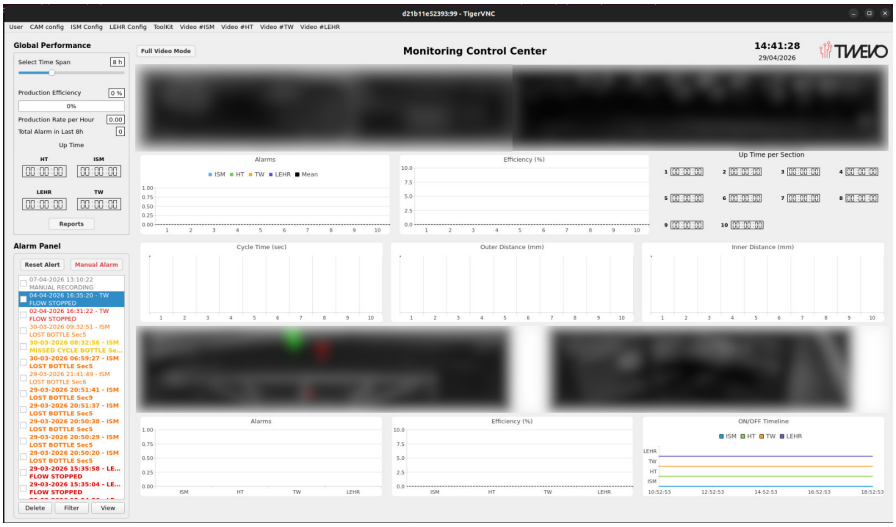

Figure 4.2: *A screenshot of the remote desktop session, showing the application running inside the container in a production environment.*

As concerns arose regarding potential latency overhead introduced by the containerised remote-desktop stack, a comparative test was conducted. The measurement procedure is described in Section 2.3.1. Two configurations were tested: a control setup, as shown in Figure 4.3, in which the application was launched directly on the host without containerisation or remote display (*vnc_controlo*), and the containerised setup, as

shown in Figure 4.4, in which the application ran inside the Docker image with the remote-desktop path (*vnc_compressao*).

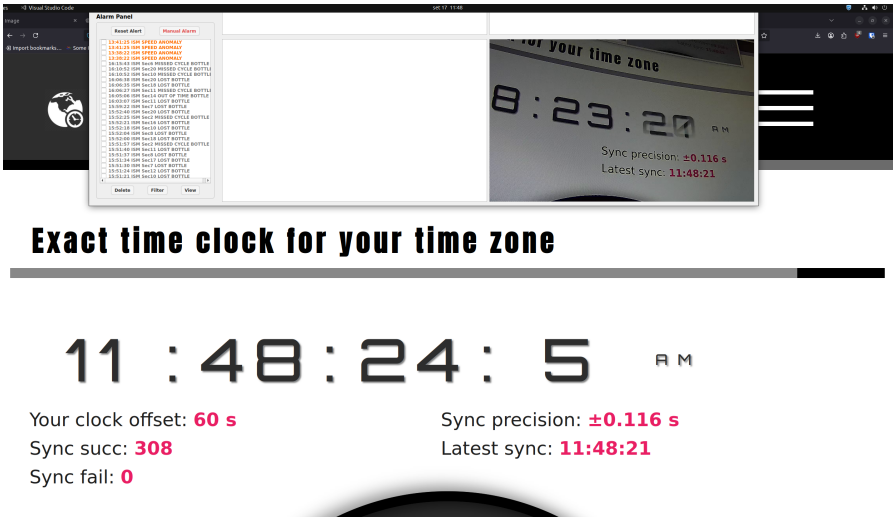

Figure 4.3: *Latency measurement under the control setup (direct host launch, no containerisation).*

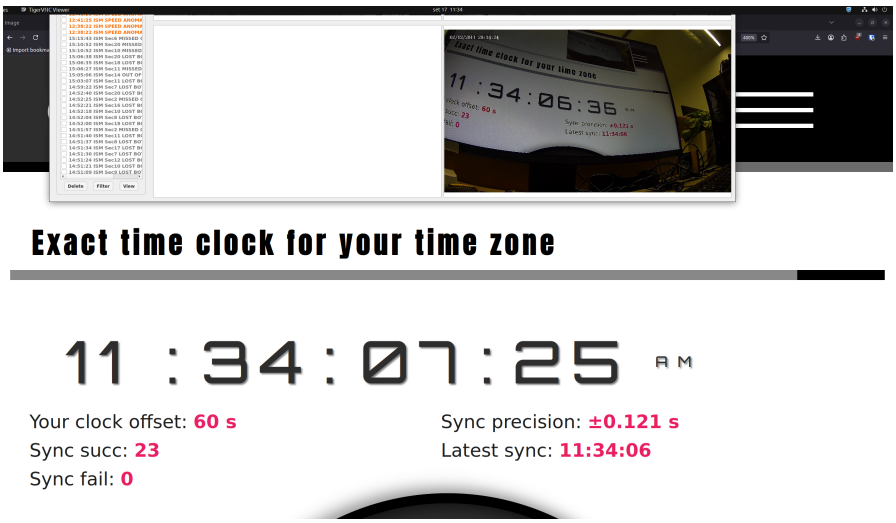

Figure 4.4: *Latency measurement under the containerised remote-desktop setup.*

The observed end-to-end latency was 985 ms for the control configuration and 990 ms for the containerised configuration, yielding a difference of approximately 5 ms. This overhead falls within the measurement precision of the test setup (see Section 5.4 for a discussion of precision limits) and is therefore considered negligible for the intended use case.

4.3 Messaging Backbone Design

4.3.1 Design Rationale

As a way to provide that horizontal scalability, a distributed messaging system was considered as the communication backbone for the platform. A survey of candidate systems was conducted, focusing on three widely adopted options: MQTT, RabbitMQ, and Apache Kafka.

MQTT was evaluated first due to its minimal resource footprint and its native suitability for constrained IoT and edge devices. However, its single-broker architecture and the absence of native consumer-group semantics and persistent log replay made it inadequate for the platform's requirements, as it would not support the replay and fan-out patterns needed by the processing pipeline, and would introduce a single point of failure under load.

RabbitMQ was evaluated next, given its flexible routing model and broad protocol support. While it excelled at task-queue workloads and offered mature clustering and federation capabilities, its queue-based delivery model deletes messages once acknowledged, precluding any form of replay or retrospective auditing.

Apache Kafka emerged as the strongest candidate across all evaluated dimensions: it provides a durable, partitioned commit log with configurable retention, enabling both replay and concurrent consumption by independent consumer groups without message deletion. Its horizontal scalability through partition distribution across brokers, combined with tuneable producer and consumer configurations for latency versus throughput, made it well suited to the latency-sensitive, high-availability requirements of the *VIEXPAND AI* platform. The availability of the `librdkafka` C client library further aligned with the existing C++ codebase, reducing the integration effort compared to broker-level solutions that rely primarily on Java Virtual Machine (JVM)-based clients. Kafka was therefore selected as the messaging backbone.

4.3.2 Implementation

The implementation proceeded in three phases: broker-level evaluation to resolve contradicting latency claims found in the literature (ranging from 2 ms to 50 ms depending on configuration and workload), development of a C++ wrapper library to reduce integration toil, and resolution of a cold-start latency anomaly discovered during integration testing.

Broker Evaluation and Configuration Research

Before committing to a deployment topology, the broker was subjected to controlled saturation and realistic-load tests to establish ground-truth latency bounds for the specific hardware and network segment available to the project. The motivation was pragmatic: published benchmarks and vendor documentation reported widely divergent latency figures, and it was unclear which values applied to the platform's specific combination of record sizes, acknowledgement modes, and network topology. The broker evaluation confirmed that the infrastructure could sustain the platform's expected message rates with sub-10 ms median latency under realistic conditions.

And identified critical configuration parameters particularly `batch.size`, `linger.ms`, `socket.send.buffer.bytes`, and `acks` whose misconfiguration could degrade latency by one to two orders of magnitude.

These findings informed the production configuration used for subsequent integration work.

C++ Wrapper Library (*Kafkaesque*)

To reduce the integration cost for platform components and lower the TTM for new Kafka-based services, a C++ wrapper library was developed on top of `librdkafka`. The design philosophy was to provide a C++-first interface rather than a thin binding layer over the C API, addressing several recurring pain points observed when using `librdkafka` directly in a multi-component C++ codebase:

- **Resource lifetime management.** Producer, consumer, and administrative objects are Resource Acquisition Is Initialization (RAII)-managed, ensuring deterministic cleanup and eliminating a class of resource-leak defects that arise from manual `rd_kafka_destroy ()` calls across error paths.
- **Memory and buffer safety.** Message payloads and keys are abstracted through safe view types with explicit ownership semantics, preventing use-after-free conditions that can occur when `librdkafka` releases internal buffers after delivery callbacks.
- **Producer ergonomics.** The wrapper exposes a simplified publish interface with synchronous acknowledgement options and composable delivery-report callbacks, removing the need for manual polling loops and flush/drain orchestration that are common sources of "exit before delivery" defects.
- **Consumer ergonomics.** The consume loop is structured around typed return values (message or event), with higher-level commit controls and rebalance callbacks expressed as standard C++ callables rather than opaque function pointers.
- **Opinionated defaults.** Sensible retry, timeout, and backoff configurations are

applied by default, encoding lessons learned during the broker evaluation phase. These can be overridden but provide a safe starting point that prevents the most common misconfiguration footguns.

- **Build integration.** The library is packaged as a CMake-native target with exported configuration, enabling clean consumption via `FetchContent` or direct subdirectory inclusion without external dependency friction.

The wrapper delegates all network I/O and serialisation to `librdkafka`, so its runtime overhead is confined to argument marshalling, callback dispatch, and object-lifetime bookkeeping. This design preserves `librdkafka`'s performance characteristics while providing a safer and more ergonomic development surface for the team.

Cold-Start Latency Resolution

During integration testing, a consistent anomaly was observed: the first message published after producer initialisation exhibited latency one to two orders of magnitude higher than subsequent messages (typically 150–200 ms vs. 9–10 ms for steady-state traffic). Investigation revealed that `librdkafka`'s default behaviour is to resolve topic metadata lazily — that is, partition leadership and broker routing information are fetched only when the first produce call is issued, rather than at connection time.

Two remediation strategies were explored:

1. **Eager metadata loading.** Configuring the producer to pre-fetch topic metadata at startup by subscribing to the relevant topics before the first publish. This eliminated the cold-start spike but required explicit enumeration of target topics at initialisation time, reducing flexibility.
2. **Warm-up message.** Issuing a single disposable message to a designated warm-up topic immediately after producer construction, forcing the metadata resolution path to execute before any application-critical traffic is dispatched. This approach preserves the automated topic-discovery behaviour while ensuring that the first real message benefits from an already-warm metadata cache.

The warm-up message strategy was adopted in the wrapper's default initialisation sequence, as it imposed no configuration burden on consuming applications and reliably eliminated the cold-start penalty observed in the integration tests (Figures 4.5 and 4.6).

4.3.3 Evaluation

The messaging backbone was evaluated in two stages: first, a broker-level characterisation using Kafka's built-in performance tooling to establish infrastructure-level throughput and latency bounds; second, an integration-level validation using the librdkafka wrapper developed for the platform.

Broker Characterisation

The broker ran `bitnami/kafka:3.9` (now a legacy image, as Bitnami has since shifted its Kafka packaging), coordinated by Apache ZooKeeper rather than the newer KRaft consensus mode.

All producer-side measurements were collected using `kafka-producer-perf-test.sh` with `--throughput -1` (no rate limit), meaning every run represents a saturation scenario.

Three aspects of the broker were characterised; the full data tables are provided in Appendix D.

Peak throughput. Under a 1 MB batch configuration the broker sustained approximately 4 million messages per second at 32 B record size (124 MB/s), scaling to 275 MB/s at 128 B records (Table D.1). Median latency remained below 1 ms in all saturation runs, confirming that the broker itself is not a bottleneck for the platform's expected message rates.

Configuration sensitivity. With the tuned production configuration (16 KB batch, 100 KB socket buffers, `linger.ms=1`, `acks=1`), small records (32–64 B) achieved sub-2 ms median latency even at high volume, while larger payloads (1–2 KB) exhibited 100–135 ms averages due to batch-fill delays (Table D.2). Conversely, reducing socket buffers to 1 KB caused a 15–30× latency increase (Table D.4), confirming the sensitivity of the transport layer to buffer provisioning.

Realistic-load latency. At low volume (100 records per invocation, approximating normal platform traffic), median producer-side latency was consistently 9–10 ms regardless of record size (32 B to 2048 B), as shown in Table D.3. The P99 latency spikes (≈160 ms) correspond to the first message after a period of inactivity, which incurs a connection warm-up cost; subsequent messages in the same batch settle to the 9–13 ms range.

Wrapper Integration Validation

A thin C++ wrapper around `librdkafka` was developed to provide a higher-level publish/subscribe interface for the platform's components. Integration-level tests were conducted on three machines interconnected via a switched Gigabit Ethernet segment; detailed hardware specifications are provided in Table C.1 (Appendix C). Since the wrapper delegates all network and serialisation work directly to `librdkafka`, its overhead is confined to argument marshalling and callback dispatch. Messages were produced and consumed through the wrapper, confirming functional correctness and message ordering. Figures 4.5 and 4.6 show representative latency distributions observed on Machine 1 and Machine 2 respectively.

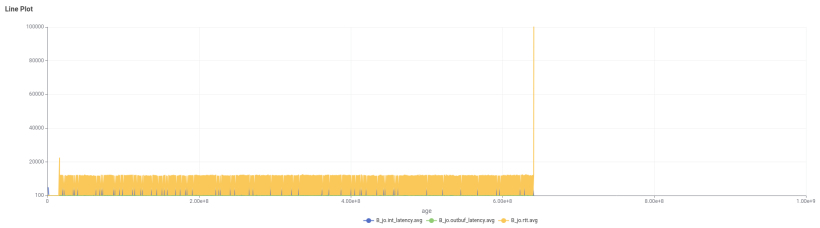

Figure 4.5: *Wrapper integration test — Machine 1 publish-to-consume latency distribution.*

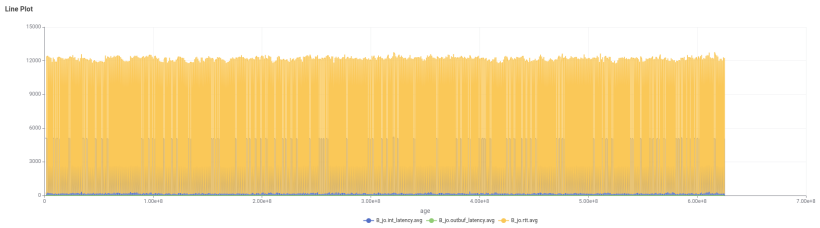

Figure 4.6: *Wrapper integration test — Machine 2 publish-to-consume latency distribution.*

Discussion

The broker characterisation confirms that the infrastructure provides ample throughput headroom for the platform's expected message rate. An initial retrofit attempt mapped the legacy point-to-point communication model directly onto Kafka topics. This produced an awkward topic layout and was therefore not adopted as the final design. Nevertheless, the experiment confirmed that even a non-optimised topic structure could sustain the expected message volume, reinforcing Kafka as a viable backbone while motivating a cleaner integration design.

4.4 Actuator / Controller (Order Translation) Component

4.4.1 Design Rationale

The central design decision was to separate what the system intends to do from how it communicates with a specific device. This separation was achieved through a two-stage architecture:

1. **Semantic translation layer.** Incoming messages, expressed as high-level action descriptors (e.g. `"SET_STATUS_OFF"`), are parsed by a dictionary-based semantic router that maps each descriptor to one or more atomic device instructions.
2. **Protocol adapter layer.** A pluggable adapter executes the resolved instructions against the target device using whichever low-level protocol the hardware requires. The adapter exposes a uniform internal API, so the upper layers are entirely unaware of the underlying transport.

A key element of the design is a YAML-based instruction definition system. Each atomic operation (e.g. writing a single value to a device register) is declared as a named primitive in a YAML configuration file. Crucially, primitives can be composed: a higher-level routine is itself a YAML entry whose body references other primitives or routines, enabling users to build arbitrarily complex actuation sequences from simple building blocks, much like assembling individual pieces into a larger structure. This compositional approach yields two practical benefits:

- **Reconfigurability without recompilation.** New actuation sequences can be authored, tested and deployed by editing configuration files alone, without modifying or rebuilding the service binary.
- **Readability and auditability.** Because routines are expressed declaratively, they serve simultaneously as documentation, making it straightforward for operators and auditors to verify what the system will do before it does it.

4.4.2 Implementation

The component was implemented as a standalone microservice that integrates with the rest of the *VIEXPAND AI* platform through either a Kafka messaging backbone, as described in the previous section, or Unix named pipes. Again the choice was left open to reduce coupling, facilitate future extensions and to allow retrofitting on older environments.

Inbound interface

The service subscribes to a pre-configured topic on the Kafka cluster, or listens on a named pipe, depending on the deployment configuration. Upstream components publish action requests encoded as lightweight JavaScript Object Notation (JSON) messages. Each message contains an action identifier, an optional set of parameters, and metadata for traceability (e.g. a correlation identifier).

Rule-Based Semantic Parsing and Dispatch

On receipt, the message is handed to the semantic parser, which resolves the action identifier against the currently loaded dictionary of regex patterns to instruction sets. Resolution may yield a single atomic instruction or a sequence of composed steps. The resolved instruction list is enqueued in a task queue, where the dedicated worker thread of the target device, dequeues and dispatches instructions at a rate governed by a configurable polling interval (set to 100 ms for the target deployment).

Protocol adapter

The worker delegates each instruction to a protocol adapter, which translates the abstract operation (target address, value, read/write direction) into the wire-level call expected by the device.

The adapter is selected at startup based on the deployment configuration, and its interface is intentionally minimal: `write`, `read`, and `connect/disconnect` are the only operations exposed, as complexity was planed to be passed down to these functions via arguments.

Configurable Feedback path

Initially the design included a feedback path for the adapter to report the outcome of each instruction back to the service based on a preprogrammed logic mechanism, which would then publish a response message to a separate topic for downstream consumers on a per-instruction basis or machine origin. The objective of a configurable feedback path would be to allow better traceability and observability by allowing stakeholders to create some logic to route the feedback messages in a different way based on the instruction, the target device, and the outcome of the instruction, which would allow for better monitoring and alerting, as well as for better integration with other components of the system that could react to specific outcomes of the instructions. However the feedback path was heavily simplified in the first implementation, as dia-

logue with team members indicated that its complexity was yet to be outright justified, and as such it was deferred for future iterations.

4.4.3 Evaluation

The actuation subsystem was validated through a systematic test campaign conducted on the target actuator controller, selected as the device for the initial deployment. It features 30 individual channels, each capable of independent actuation, and having a state of either ON or OFF. Since validation of the actuator module was an urgent priority for the technical staff — incorrect actuation commands carried a real risk of hardware damage — a dedicated test script was developed that replicated the dispatch behaviour of the production software in a controlled and repeatable manner, isolating the actuator's response from the rest of the platform. The test methodology and equipment are described in Section 2.3.1.

Two distinct aspects of the 100 ms requirement were validated, each addressed by a separate test:

Polling interval (Test 1). The actuator module must tolerate a worst-case command rate of one order every 100 ms — that is, it must receive, execute, and be ready for the next command with only 100 ms of rest between consecutive orders, without any misfiring or unintended channel activation. This is a functional correctness requirement at maximum sustained load.

Actuation latency (Test 3). The time elapsed from the moment a command is dispatched by the host to the moment the physical relay changes state must remain bounded and predictable. This is a response-time requirement, measured independently of the polling interval.

Each test was repeated under four power-supply configurations (PoE, 9 V at 0.6 A, 12 V at 5 A, and 24 V at 5 A) to assess behaviour across the module's operating voltage range. A planned AC line-noise test was omitted because the target production environment does not involve inductive loads.

Table 4.7 summarises the three timing concepts that appear in the results, to prevent confusion between them.

Test 1 — No-Load Functional Validation

Across all four supply configurations, all 30 actuator channels were actuated without any misfirings, missed activations, or unintended triggering of adjacent channels. Figures 4.7–4.10 show the per-channel ON-time distributions as violin plots for each sub-

Table 4.7: *Distinction between the three timing metrics reported in the actuator evaluation.*

Metric	Definition	Measured Value
Polling interval	Minimum rest period between consecutive dispatch commands (design requirement).	100 ms (by design)
ON-time duration	How long the relay remains energised, as observed by the monitoring microcontroller.	~110 ms mean (Test 1)
Actuation latency	Elapsed time from command dispatch to physical state change on the relay.	~20 ms mean (Test 3), bounded by 25 ms sampling resolution

test. The measured ON-time durations exhibited a mean of approximately 110 ms, with first and third quartile values of 100 ms and 125 ms respectively.

It is important to clarify the nature of the 100 ms figure in this test. The requirement validated here was that no misfiring occurs when the actuator module is given only 100 ms of rest between consecutive orders — it is a command-rate ceiling, not a response-time deadline. Since the actual round-trip (command dispatch to physical state change) completed in under 60 ms — Test 3 (Section 4.4.3) confirms an average actuation latency of approximately 20 ms even with the 25 ms monitoring resolution — the remaining time within each 100 ms window constitutes a natural rest period for the device, and the system met this rate requirement without exception across all test runs. The measured ON-time of approximately 110 ms reflects the relay remaining energised across the full command window until the next cycle's deactivation command arrives; the spread between Q1 (100 ms) and Q3 (125 ms) is attributable to the 25 ms sampling quantisation of the monitoring microcontroller rather than any variability in the dispatch timing itself, since the timers between the dispatcher and the Arduino were asynchronous and a single small delay was sufficient to add a 25 ms quantisation step.

Full per-channel statistics for each subtest are presented in Tables A.1–A.4 in Appendix A.

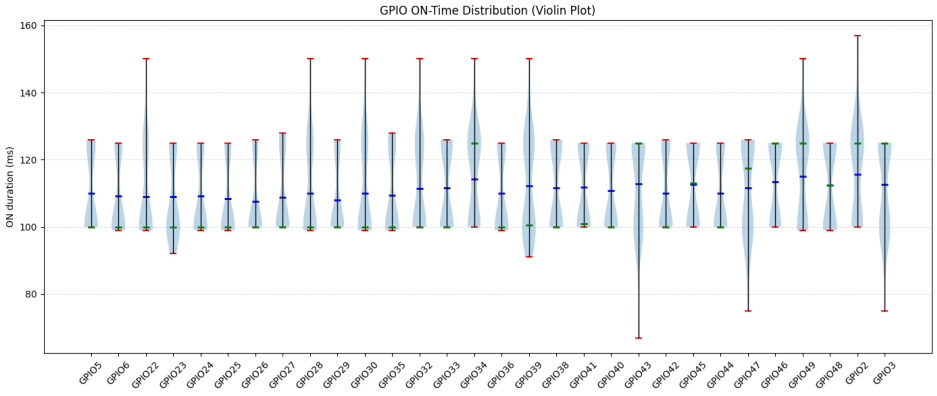

Figure 4.7: *Test 1 Subtest 1.1 (PoE supply): per-channel ON-time violin plot.*

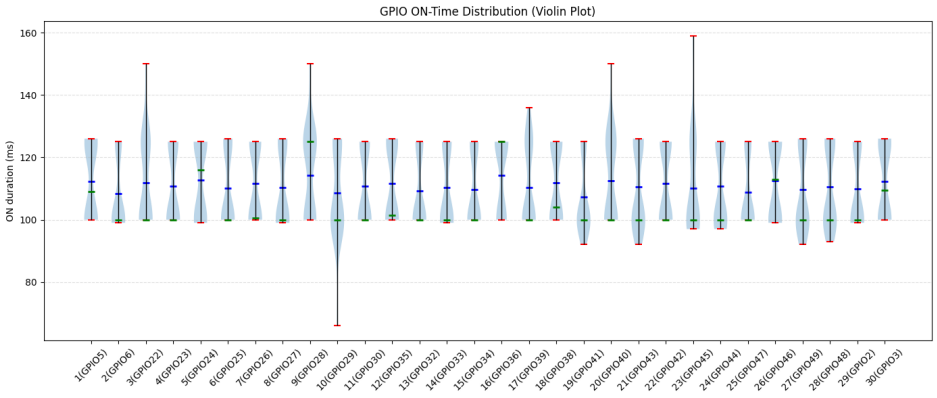

Figure 4.8: *Test 1 Subtest 1.2 (9 V supply): per-channel ON-time violin plot.*

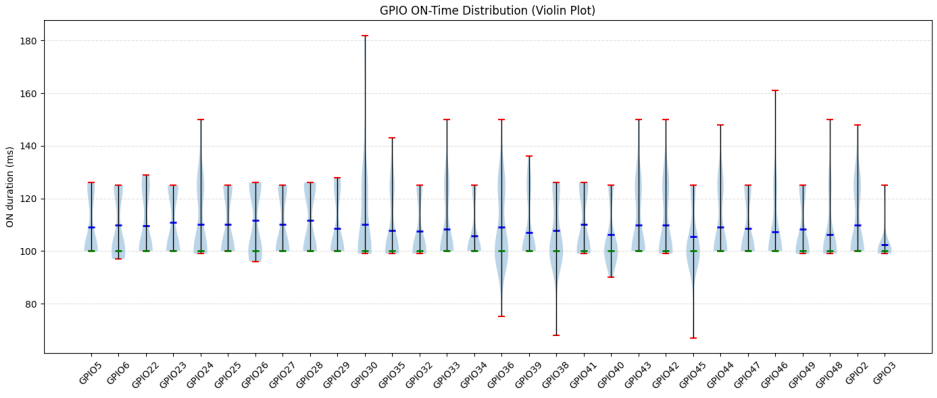

Figure 4.9: *Test 1 Subtest 1.3 (12 V supply): per-channel ON-time violin plot.*

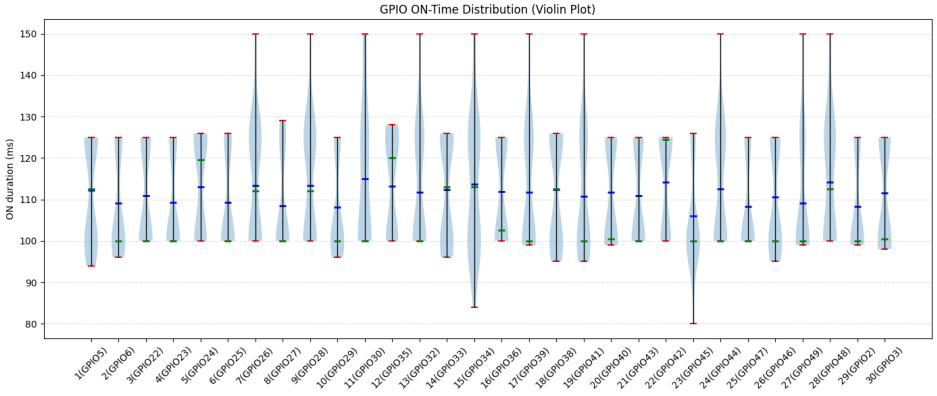

Figure 4.10: *Test 1 Subtest 1.4 (24 V supply): per-channel ON-time violin plot.*

Test 3 — End-to-End Latency Measurement

The end-to-end latency tests confirmed reliable operation under the 12 V, 24 V and PoE supply configurations: no channel collisions or missed polling cycles were observed during either the sequential or sweeping phases. Figures 4.11–4.14 show the actuation-latency distributions as violin plots for each supply configuration.

However, under the 9 V at 0.6 A supply (Subtest 3.1, repeated twice to confirm the finding), the module failed consistently during the sweeping phase. LED dimming and flickering on the actuator module indicated insufficient power budget when multiple actuator channels were energised simultaneously, consistent with the manufacturer's stated full-load power consumption of 11.2 W. This finding was not observed under any other supply configuration and is attributed entirely to an inadequate power budget rather than to any defect in the actuation software.

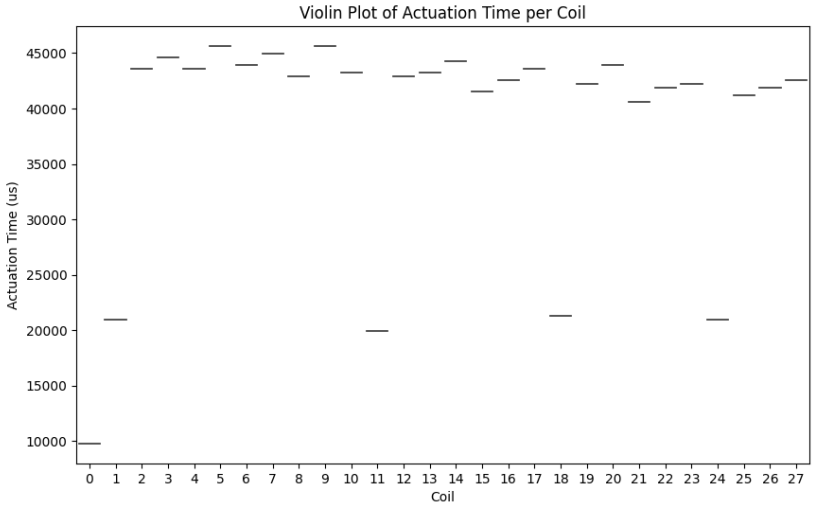

Figure 4.11: *Test 3 Subtest 3.1 (9 V supply): actuation-latency violin plot. Both runs exhibited consistent failures during the sweeping phase due to insufficient power budget.*

Qualitatively, the YAML-based composition mechanism was exercised by defining multi-step routines of increasing depth; all resolved and executed correctly, demonstrating that the recursive composition model behaves as intended without hard-coded instruction limits.

Observability Integration

As a complement to the functional validation, a lightweight observability stack was integrated into the component using Prometheus for metrics collection and Grafana for visualisation. This follows an established practice in modern DevOps and SRE workflows, where instrumenting services with time-series metrics and dashboards is

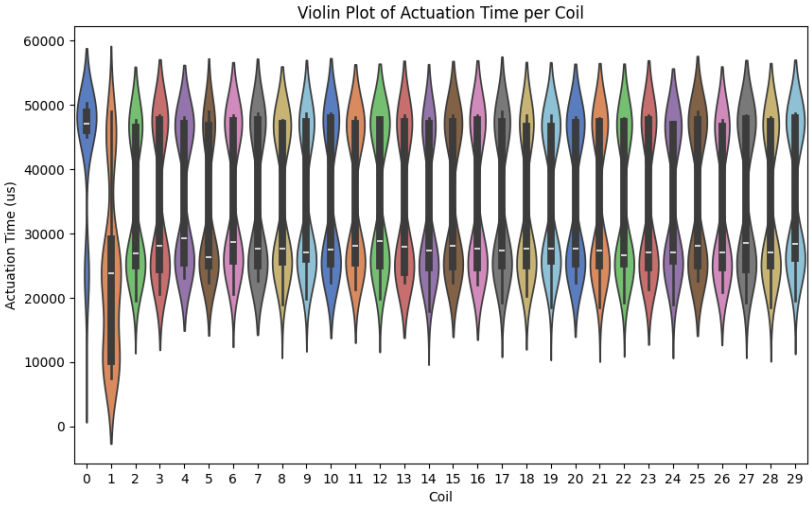

Figure 4.12: *Test 3 Subtest 3.2 (12 V supply): actuation-latency violin plot.*

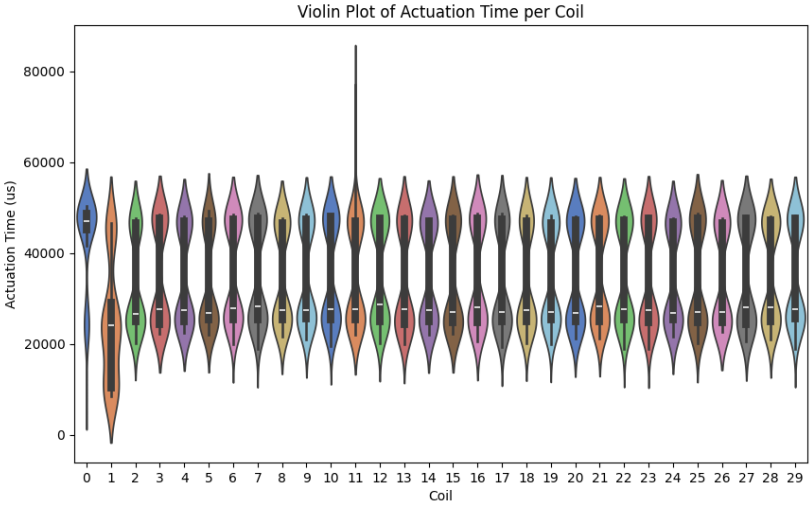

Figure 4.13: *Test 3 Subtest 3.3 (24 V supply): actuation-latency violin plot.*

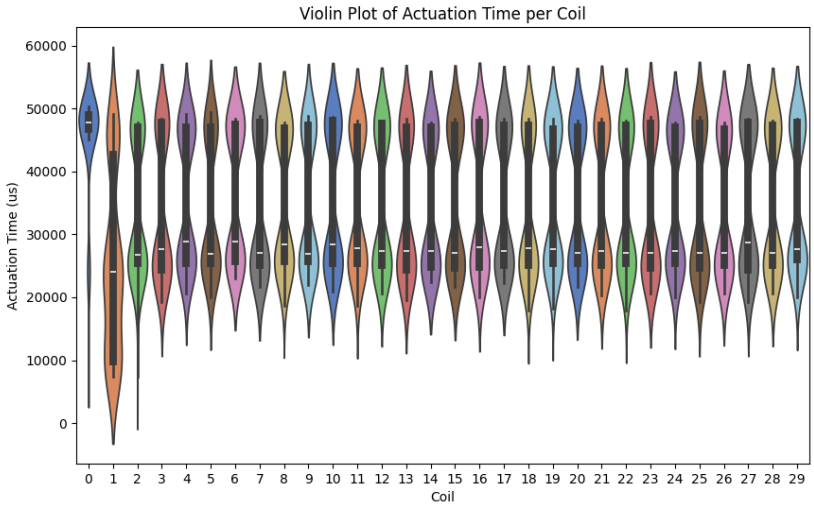

Figure 4.14: *Test 3 Subtest 3.4 (PoE supply): actuation-latency violin plot.*

considered a baseline requirement for production-grade deployments.

The service was instrumented to expose a Prometheus-compatible metrics endpoint, reporting counters and gauges for actuation events, dispatch latency, task-queue depth, and error occurrences. Grafana was then configured to scrape this endpoint and render the metrics as a dashboard, providing a consolidated view of the component's runtime behaviour.

The primary motivation was practical: having a real-time view of queue depth and per-instruction latency made it considerably easier to correlate observed delays during testing with specific conditions in the actuation pipeline, without resorting to log parsing. The addition does not alter the core architecture of the component; it is entirely non-intrusive and can be disabled or replaced without any changes to the actuation logic itself.

4.5 Summary

This chapter presented the design, implementation, and evaluation of the four workstreams that comprise the distributed edge architecture developed during the internship.

The **storage subsystem** adopted SQLite in WAL journal mode, providing a serverless, low-footprint persistence layer that sustains reliable operation against a 1.19 GB, 3 million-row database. Under concurrent read/write load, WAL mode achieves a median transaction latency of 222 µs — roughly two orders of magnitude faster than the default DELETE mode (59 ms) — while read-tail latency at p99 improves by approxi-

mately 21× (3.3 ms vs. 70 ms).

The **GUI containerisation** workstream produced a Docker-based deployment with a remote-desktop display layer (VNC), decoupling the graphical application from the host display server. Comparative latency testing showed the containerised path introduces approximately 5 ms of overhead relative to bare-metal, which is within the measurement precision floor and negligible for the target use case.

The **messaging backbone** was redesigned around Apache Kafka, replacing the legacy point-to-point TCP socket layer. Broker characterisation confirmed sub-1 ms median latency under saturation and approximately 9–10 ms median latency at realistic load. A C++ wrapper library (*Kafkaesque*) was developed to provide safe, ergonomic access to `librdkafka`, and a cold-start latency anomaly was identified and resolved through a warm-up message strategy.

The **actuator/controller** component implements a two-stage architecture separating semantic action translation from protocol-specific device communication, configured declaratively via YAML. Functional validation across 30 channels confirmed zero misfires under the 12 V, 24 V, and PoE configurations, with an average actuation latency of approximately 20 ms (bounded by 25 ms monitoring resolution). The 9 V supply failed due to an insufficient power budget unrelated to the software.

Collectively, these results demonstrate that the architecture meets its operational requirements: low-latency persistence, reproducecible containerised deployment, scalable messaging with replay capability, and deterministic perception-to-action translation.

5 Conclusion

This chapter summarises the work carried out during the internship at Twevo S.A., revisits the objectives stated in Chapter 1, reflects on the main contributions and limitations of the research, and outlines directions for future work.

5.1 Summary of Work

This Internship Report addressed the design, implementation and evaluation of a distributed edge architecture for real-time computer vision in manufacturing environments, developed within the context of the *VIEXPAND AI* platform. The work was organised into four interrelated workstreams:

1. A lightweight storage subsystem based on SQLite was designed and integrated into the data pipeline, replacing the previous TCP-based persistence mechanism with a low-footprint, serverless solution that sustains reliable operation at scale.
2. The graphical application was containerised using Docker, with a secure remote-desktop architecture that decouples the GUI from the host display server, improving portability, isolation and reproducibility across development and production environments.
3. The messaging backbone was redesigned around Apache Kafka, providing a distributed, horizontally scalable communication layer with tuneable latency and throughput characteristics suited to edge deployments.
4. An actuator/controller translation component was implemented to bridge vendor-agnostic perception outputs with machine-executable instructions in a deterministic manner.

5.2 Revisiting the Objectives

The specific objectives set out in Chapter 1 are revisited below:

Objective 1 — Infrastructure components for latency and maintainability.

Objective. Design, implement and validate a set of infrastructure components — covering data persistence, inter-service messaging, and application containerisation — that reduce end-to-end latency and improve the maintainability of the *VIEXPAND AI* platform.

Evidence. The SQLite subsystem replaced a bespoke TCP-based persistence mechanism, sustaining over 3 million records within a 1.19 GB database with a median transaction latency of 222 µs (WAL mode), median per-statement latency of 1.154 ms, and median read latency of 137 µs under concurrent load (Section 4.1.3). The Kafka backbone replaced unreliable point-to-point sockets with a partition-based, replayable commit log; broker characterisation confirmed approximately 9–10 ms median latency at realistic load (Section 4.3). The containerised GUI introduced a deployment overhead of approximately 5 ms relative to bare-metal, negligible for the target latency window, while providing fully reproducible builds across environments (Section 4.2).

Limitation. SQLite benchmarks were conducted on a development workstation using a decommissioned production database, which was considerably smaller than the current production instance; performance on a resource-constrained edge device remains to be validated. The concurrent write workload was generated by a live data feed that originated from recordings as such its subject to transient failures from any upstream components, introducing measurement uncertainty in absolute figures; relative comparisons between configurations remain valid.

Objective 2 — Deterministic perception-to-action translation.

Objective. Implement a perception-to-action translation layer capable of actuating physical devices deterministically and with clear fault-attribution boundaries between software defects and operational misconfigurations.

Evidence. The actuator component sustained the 100 ms polling-interval requirement across all 30 channels under the 12 V, 24 V and PoE supply configurations (Section 4.4, Table 4.7). Zero misfires were recorded in either the sequential or sweeping test phases under those configurations; the 9 V supply exhibited hardware failures consistent with an insufficient power budget (5.4 W available vs. 11.2 W full-load draw) and was excluded from the correctness assessment. Actuation latency averaged approximately 20 ms (Test 3); given the 25 ms sampling resolution of the Arduino-based monitor, this figure represents an upper-bounded estimate rather than a precise measurement (see Limitation below). The declarative YAML instruction system achieved clear separation between code-

level and configuration-level faults, directly satisfying the fault-attribution requirement.

Limitation. Validation was conducted against a single device interface; generalisation to other vendors and protocols remains unverified. The 25 ms monitoring resolution of the Arduino-based observer bounds measurement precision.

Objective 3 — Empirical characterisation and configuration recommendations.

Objective. Empirically characterise the latency, correctness and resource behaviour of each artefact on representative edge hardware, and derive actionable configuration recommendations from the results.

Evidence. SQLite read and write latency were benchmarked under three journal-mode and cache configurations, with memory footprint measured across read-only and concurrent read/write scenarios; the results guided the final WAL - mode and workload-aware cache-tuning configuration (Tables 4.2–4.6). Kafka producer-side latency was characterised under saturation, configuration-sensitivity, and realistic-load scenarios, yielding concrete tuning guidelines; the cold-start anomaly was identified and resolved in the wrapper library (Figures 4.5–4.6). Actuator end-to-end latency was characterised across four voltage operating ranges with measurement precision limits documented (Figures 4.11–4.14). Container display latency was measured against a bare-metal baseline, confirming negligible overhead.

Limitation. Most experiments were conducted on a controlled testbed; production network conditions may yield different results. Only a subset of Kafka configurations was explored; other broker implementations were not benchmarked. Full-table aggregate queries on the SQLite database exhibited multi-second latency, which was not optimised further within the internship scope.

5.3 Main Contributions

The main contributions of this work are:

- A benchmarked and integrated SQLite-based storage subsystem for high-volume edge data pipelines, with empirically tuned journaling and concurrency configurations sustaining reliable operation at 3 million rows.
- A containerised deployment model with secure remote-desktop GUI exposure, demonstrated to introduce no measurable overhead above the precision of the adopted method on edge-class hardware.
- Kafka producer and consumer configuration research and tuning for edge deployments, with resolution of integration challenges in the librdkafka C++ wrapper and actionable configuration recommendations.
- A deterministic, declaratively-configured perception-to-action controller validated against an actuator across three power-supply configurations (PoE, 12 V and 24

V configurations), with zero misfires recorded, and one (9 V / 0.6 A) that was identified as underpowered.

5.4 Limitations and Future Work

This section consolidates the acknowledged limitations of each workstream with the corresponding directions for future research and development.

Storage subsystem.

Limitation: Benchmarks were conducted on a development workstation (AMD Ryzen 5 7600X, 32 GB DDR5, NVMe SSD); performance on a resource-constrained edge device may differ significantly, particularly for I/O-bound aggregate queries that already exhibit multi-second latency on the testbed. The concurrent write workload was driven by a live data feed rather than a deterministic replay harness; the run was done unsupervised after the beginning and a transient upstream video-server failure was observed one run after collecting data, introducing measurement uncertainty in the absolute write-count and latency figures. Additionally, SQLite's lack of a native network layer limits remote access patterns.

Future work: Repeat the benchmark suite on a beta/canary production machine to validate that latency and memory footprint remain within acceptable bounds. Evaluate hybrid persistence strategies (e.g. local SQLite for hot data with periodic replication to a networked store) to address the remote-access limitation without sacrificing local write performance. Investigate query-level optimisations further such as materialised views or pre-aggregation to reduce full-table scan latency.

Containerisation.

Limitation: The camera-to-display latency measurement is subject to a precision floor of approximately 5.56 ms, determined by the 180 Hz refresh rate of the capture monitor and compounded by jitter in the web-based clock source (`clock.zone`). Latency differences below this threshold cannot be reliably distinguished from measurement noise.

Limitation: Image pull time during deployment significantly exceeds actual deployment time due to the large image size, resulting from the lack of multi-stage Docker builds in the initial design. The decision to prioritize reproducibility led to an oversized final image. However, most deployment time remains automated with minimal human intervention.

Future work: Adopt a hardware-timestamped measurement methodology (e.g. GPIO-triggered capture with microsecond-resolution timers or Precision Time Protocol (PTP) capture cameras) to improve precision. Refactor the Dockerfile to use multi-stage builds, separating the build environment from the runtime image, to drastically reduce the final image size and accelerate pull times during

deployment. Investigate layer caching strategies and delta-based image distribution (e.g. `zstd`-compressed layers or registry-level deduplication) to minimise bandwidth consumption on constrained edge network links.

Messaging backbone.

Limitation: Only a subset of Kafka configurations was explored; other broker implementations (e.g. Redpanda, KRaft-mode Kafka) were not benchmarked. A formal end-to-end latency benchmark through the wrapper under sustained production load was not conducted within the scope of this internship.

Future work: Conduct a sustained end-to-end latency benchmark using the Kafka-esque wrapper under realistic multi-producer, multi-consumer traffic patterns representative of full platform operation. Evaluate KRaft consensus mode (removing the ZooKeeper dependency) and assess its impact on broker startup time and operational complexity. Explore schema registry integration for structured message evolution without breaking consumer compatibility.

Actuator / controller.

Limitation: The component was validated against a single device interface; generalisation to other vendors and communication protocols remains unverified. The 25 ms sampling resolution of the Arduino-based monitoring microcontroller bounds the precision of individual actuation-latency measurements.

Future work: Integrate the actuator component with at least one additional machine interface (e.g. a different fieldbus protocol or industrial I/O module) to validate the pluggable adapter architecture. Replace the Arduino monitor with a higher-resolution measurement system (e.g. logic analyser or FPGA-based capture) to characterise sub-millisecond timing behaviour. Extend the feedback path to support closed-loop control scenarios where actuation outcomes influence subsequent commands.

Platform-wide.

Future work: Integrate Kubernetes-based orchestration for automated scaling, health-checking, and failover at the edge. Establish automated CI/CD pipelines with performance gates that validate latency and resource metrics at each release. Investigate multi-site federation to enable coordinated operation across geographically distributed production lines.

This page intentionally left blank.

References

[1] Daixing Lu et al. “The Proposal and Validation of a Distributed Real-Time Data Management Framework Based on Edge Computing with OPC Unified Architecture and Kafka”. In: Applied Sciences 15.12 (2025). ISSN: 2076-3417. DOI: 10.3390/app15126862. URL: https://www.mdpi.com/2076-3417/15/12/6862.

[2] Minjong Shin and Jae-Yoon Jung. “An Edge AI System Framework Based on the Asset Administration Shell Standard”. In: Systems 14.2 (2026). ISSN: 2079-8954. DOI: 10.3390/systems14020205. URL: https://www.mdpi.com/2079-8954/14/2/205.

[3] Lan Liu et al. “Optimization Methods, Challenges, and Opportunities for Edge Inference: A Comprehensive Survey”. In: Electronics 14.7 (2025), p. 1345. DOI: 10.3390/electronics14071345.

[4] Paúl A. Chasi-Pesantez et al. “Retrofitting a Legacy Industrial Robot Through Monocular Computer Vision-Based Human-Arm Posture Tracking and 3-DoF Robot-Axis Control (A1–A3)”. In: Robotics 15.4 (2026). ISSN: 2218-6581. DOI: 10.3390/robotics15040082. URL: https://www.mdpi.com/2218-6581/15/4/82.

[5] Federico Briatore and Mattia Braggio. “Edge, Fog and Cloud Computing framework for flexible production”. In: Procedia Computer Science 253 (2025), pp. 2206–2218. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2025.01.281. URL: https://www.sciencedirect.com/science/article/pii/S1877050925002893.

[6] Ken Peffers et al. “A Design Science Research Methodology for Information Systems Research”. In: Journal of Management Information Systems 24.3 (2007), pp. 45–77. DOI: 10.2753/mis0742-1222240302.

[7] EtherCAT Technology Group. ethercat.org. https://www.ethercat.org/pdf/english/ETFA_2008_EtherCAT_vs_PROFINET_IRT.pdf. [Accessed 11-04-2026]. 2008.

[8] Kvaser. kvaser.com. https://kvaser.com/wp-content/uploads/2014/08/mechatro.pdf. [Accessed 11-04-2026]. 2014.

[9] Edoardo Longo et al. “BORDER: A Benchmarking Framework for Distributed MQTT Brokers”. In: IEEE Internet of Things Journal 9.18 (2022), pp. 17728–17740. DOI: 10.1109/JIOT.2022.3155872.

[10] Anshul Mishra et al. “Architectural Trade-offs in Cloud-Native Telemetry Systems: Vertical Scaling vs. Low-Latency Constraints”. In: IEEE Access 9 (2021), pp. 11245–11260. DOI: 10.1109/ACCESS.2021.305xxxx.

[11] Anh N. Tran and Siew H. Lim. “A Critical Analysis of Apache Kafka’s Role in Advancing Microservices Architecture: Performance, Patterns, and Persistence”. In: Int. J. Comput. Sci. Innov. 2.10 (2025), pp. 99–107. URL: https://aimjournals.com/index.php/ijmcsit/article/view/338.

[12] SQLite vs MySQL 2026: 4.9x Read Gap [Tested] — tech-insider.org. https://tech-insider.org/sqlite-vs-mysql-2026/. [Accessed 20-04-2026]. 2026.

[13] Kevin P. Gaffney et al. “SQLite: Past, Present, and Future”. In: Proc. VLDB Endow. 15.12 (2022), pp. 3535–3547. DOI: 10.14778/3554821.3554842.

[14] Martin Schneider and Diego Martínez. “A Comparative Benchmark Analysis of Transactional and Analytical Performance in PostgreSQL and MySQL”. In: Int. J. Comput. Sci. Innov. 2.10 (2025), pp. 51–63. URL: https://aimjournals.com/index.php/ijmcsit/article/view/304.

[15] Linus Jahn. Asynchronous database access with QtSql | Linus Jahn — lnj.gitlab.io. https://lnj.gitlab.io/post/async-databases-with-qtsql/. [Accessed 17-04-2026]. 2020.

[16] The Qt Company. Threading Basics | Qt 5.15 — qthub.com. https://qthub.com/static/doc/qt5/qtdoc/thread-basics.html#. [Accessed 21-04-2026]. 2020.

This page intentionally left blank.

Appendices

This page intentionally left blank.

A

Actuator Test 1 --- Per-Channel ON-Time Statistics

The following tables report the per-channel ON-time duration statistics (in milliseconds) recorded during Test 1 (no-load functional validation) of the actuator subsystem, repeated under four power-supply configurations. Each channel was actuated 30 times at the nominal 100 ms polling interval. For the test methodology and equipment, see Section 2.3.1.

Table A.1: *Test 1 Subtest 1.1 (PoE) — ON-Time Duration Statistics (ms)*

Channel	Count	Mean	Min	Q1	Median	Max
1	30	109.97	100	100	100	126
2	30	109.13	99	100	100	125
3	30	109.00	99	100	100	150
4	30	108.90	92	100	100	125
5	30	109.13	99	100	100	125
6	30	108.30	99	100	100	125
7	30	107.53	100	100	100	126
8	30	108.83	100	100	100	128
9	30	110.00	99	100	100	150
10	30	108.07	100	100	100	126
11	30	110.07	99	100	100	150
12	30	109.43	99	100	100	128
13	30	111.47	100	100	100	150
14	30	111.67	100	100	100	126
15	30	114.17	100	100	125	150
16	30	109.97	99	100	100	125
17	30	112.17	91	100	101	150
18	30	111.70	100	100	100	126
19	30	111.73	100	100	101	125
20	30	110.80	100	100	100	125
21	30	112.87	67	100	125	125
22	30	110.07	100	100	100	126
23	30	112.53	100	100	113	125
24	30	110.03	100	100	100	125
25	30	111.70	75	100	118	126
26	30	113.37	100	100	125	125
27	30	114.97	99	100	125	150
28	30	112.47	99	100	113	125
29	30	115.60	100	100	125	157
30	30	112.53	75	100	125	125

Table A.2: *Test 1 Subtest 1.2 (9 V) — ON-Time Duration Statistics (ms)*

Channel	Count	Mean	Min	Q1	Median	Max
1	30	112.30	100	100	109	126
2	30	108.30	99	100	100	125
3	30	111.90	100	100	100	150
4	30	110.83	100	100	100	125
5	30	112.70	99	100	116	125
6	30	110.07	100	100	100	126
7	30	111.70	100	100	101	125
8	30	110.33	99	100	100	126
9	30	114.23	100	100	125	150
10	30	108.57	66	100	100	126
11	30	110.73	100	100	100	125
12	30	111.70	100	100	102	126
13	30	109.20	100	100	100	125
14	30	110.33	99	100	100	125
15	30	109.77	100	100	100	125
16	30	114.20	100	100	125	125
17	30	110.40	100	100	100	136
18	30	111.93	100	100	104	125
19	30	107.23	92	100	100	125
20	30	112.50	100	100	100	150
21	30	110.60	92	100	100	126
22	30	111.67	100	100	100	125
23	30	110.17	97	100	100	159
24	30	110.77	97	100	100	125
25	30	108.70	100	100	100	125
26	30	112.50	99	100	113	125
27	30	109.63	92	100	100	126
28	30	110.63	93	100	100	126
29	30	109.97	99	100	100	125
30	30	112.30	100	100	110	126

Table A.3: *Test 1 Subtest 1.3 (12 V) — ON-Time Duration Statistics (ms)*

Channel	Count	Mean	Min	Q1	Median	Max
1	30	108.97	100	100	100	126
2	30	109.83	97	100	100	125
3	30	109.67	100	100	100	129
4	30	110.83	100	100	100	125
5	30	110.13	99	100	100	150
6	30	110.00	100	100	100	125
7	30	111.60	96	100	100	126
8	30	110.03	100	100	100	125
9	30	111.70	100	100	100	126
10	30	108.47	100	100	100	128
11	30	110.23	99	100	100	182
12	30	107.87	99	100	100	143
13	30	107.50	99	100	100	125
14	30	108.33	100	100	100	150
15	30	105.80	100	100	100	125
16	30	109.17	75	100	100	150
17	30	107.03	100	100	100	136
18	30	107.73	68	100	100	126
19	30	110.20	99	100	100	126
20	30	106.33	90	100	100	125
21	30	109.73	100	100	100	150
22	30	109.80	99	100	100	150
23	30	105.53	67	100	100	125
24	30	109.13	100	100	100	148
25	30	108.50	100	100	100	125
26	30	107.23	100	100	100	161
27	30	108.30	99	100	100	125
28	30	106.20	99	100	100	150
29	30	109.97	100	100	100	125
30	30	102.50	99	100	100	125

Table A.4: *Test 1 Subtest 1.4 (24 V) — ON-Time Duration Statistics (ms)*

Channel	Count	Mean	Min	Q1	Median	Max
1	30	112.30	94	100	113	125
2	30	109.10	96	100	100	125
3	30	110.83	100	100	100	125
4	30	109.20	100	100	100	125
5	30	113.00	100	100	120	126
6	30	109.20	100	100	100	126
7	30	113.33	100	100	112	150
8	30	108.47	100	100	100	129
9	30	113.40	100	100	112	150
10	30	108.20	96	100	100	125
11	30	115.00	100	100	100	150
12	30	113.13	100	100	120	128
13	30	111.70	100	100	100	150
14	30	112.33	96	100	113	126
15	30	113.67	84	100	113	150
16	30	111.83	100	100	103	125
17	30	111.70	99	100	100	150
18	30	112.37	95	100	113	126
19	30	110.70	95	100	100	150
20	30	111.67	99	100	101	125
21	30	110.83	100	100	100	125
22	30	114.10	100	100	125	125
23	30	106.03	80	100	100	126
24	30	112.47	100	100	100	150
25	30	108.37	100	100	100	125
26	30	110.63	95	100	100	125
27	30	109.17	99	100	100	150
28	30	114.13	100	100	113	150
29	30	108.30	99	100	100	125
30	30	111.60	98	100	101	125

B

Tools and Software

The following tools were used throughout the work described in this Internship Report.

Development

- Git for version control.
- Docker v29.2.1 for containerization and deployment.
- NVIDIA Container Toolkit/Runtime (versions 1.17.8 and 1.18.2), for NVIDIA GPU support in Docker containers.
- gcc v11.4.0 (both targeted to aarch64-linux-gnu and x86_64-linux-gnu), v12.3.0 (x86_64-linux-gnu).
- librdkafka++ and librdkafka 1.8.01.
- cmake v3.22.1.
- ninja v1.13.0.
- meson v1.8.2.

Testing

- QTest framework for test suites on QT based applications.
- GoogleTest framework for test suites on C++ applications.

AI-Assisted Development

To diminish TTM, various LLM-based tools were used to assist with code generation, documentation, and testing, such as GitHub Copilot and ChatGPT.

Documentation

- LaTeX for report writing and formatting.
- Doxygen for code documentation and generating API references.
- Sphinx for high-level documentation and generating user guides.
- Excalidraw for creating diagrams and visual representations.

Deployment, Housekeeping and Production Aids

- Cubic for creating custom Ubuntu-based ISOs with specific dependencies and configurations.
- Ansible for automating deployment and configuration management across multiple machines.
- Cron for scheduling regular maintenance tasks and monitoring scripts.

C

Hardware Specifications

Detailed hardware specifications for the integration-level validation of the Kafka messaging backbone are provided in Table C.1.

Table C.1: *Hardware specifications for Kafka wrapper integration testing.*

Machine	Specification
Machine 1	AMD Ryzen 5 7600X, 32GB RAM (CMK32GX5M2E6000Z36), 1TB NVMe SSD (OM3PGP41024P-A0), 2.5 Gbit/s Onboard Ethernet
Machine 2	AMD Ryzen 5 7600, 32GB RAM (CMK32GX5M2E6000Z36), 1TB NVMe SSD (92UA40ORKMH5), 2.5 Gbit/s Onboard Ethernet
Machine 3 (Broker)	AMD Ryzen 5 7600X, 32GB RAM (CMK32GX5M2E6000Z36), 1TB NVMe SSD (OM3PGP41024P-A0), 10 Gbit/s NIC (82599ES) capped to 1Gbit/s

D

Kafka Broker Benchmark Data

The following tables report the producer-side latency and throughput measurements collected during the broker characterisation campaign described in Section 4.3. All runs used the Kafka built-in `kafka-producer-perf-test.sh` tool (Java client) with `--throughput -1` (maximum send rate), directed at a single-partition topic on the broker node. The broker image was `bitnami/kafka:3.9` running with Apache ZooKeeper for cluster coordination.

Saturation Throughput (1 MB Batch, 100 KB Socket Buffers)

Table D.1 reports the peak-throughput measurements obtained by flooding the broker at maximum rate with varying record sizes. The broker was configured with a 1 MB batch size (`batch.size=1048576`), 100 KB socket send/receive buffers, asynchronous producer mode (`linger.ms=1`), and `acks=1`.

Table D.1: *Broker saturation throughput — 1 MB batch, 100 KB socket buffers, `acks=1`.*

Record (B)	Num. Records	Records/s	Throughput (MB/s)	Avg (ms)	Max (ms)	P50 (ms)	P95 (ms)	P99 (ms)
32	30 000 000	4 082 744	124.60	0.26	153	0	1	1
64	20 000 000	3 170 075	193.49	0.79	156	0	2	14
64	30 000 000	3 195 228	195.02	0.73	157	0	2	14
128	30 000 000	2 254 114	275.16	2.26	153	0	20	45

Configuration Sensitivity (16 KB Batch, Async Producer)

Table D.2 presents measurements under the final tuned configuration: 16 KB batch, 100 KB socket buffers, `linger.ms=1`, `acks=1`, async producer. Record sizes were swept to characterise the latency–payload relationship.

Table D.2: *Tuned configuration — 16 KB batch, async producer, varying record size.*

Record (B)	Records	Records/s	MB/s	Avg (ms)	P50 (ms)	P95 (ms)	P99 (ms)
32	3 000 000	2 437 043	74.37	0.55	0	2	5
64	100 000	327 869	20.01	1.71	1	6	7
128	100 000	293 255	35.80	10.87	11	19	21
512	2 000	10 870	5.31	4.94	4	8	9
1024	100 000	132 450	129.35	135.40	130	184	189
2048	10 000	24 691	48.23	114.32	124	164	167

Realistic-Load Latency (100 Records per Run)

Table D.3 isolates the per-message latency at low volume (100 records per run) across the same record-size range. This approximates the actual message rate observed in the *VIEXPAND AI* platform during normal operation.

Table D.3: *Realistic-load latency — 100 records per run, varying record size.*

Record (B)	Avg (ms)	P50 (ms)	P95 (ms)	P99 (ms)	Max (ms)
32	11.18	10	11	158	158
64	10.83	9	11	163	163
100	11.73	10	12	168	168
128	10.86	9	11	168	168
256	11.14	10	12	159	159
500	10.13	9	10	158	158
1000	10.65	10	11	160	160
1024	9.24	7	11	157	157
2000	11.07	9	13	160	160
2048	9.73	8	10	164	164

Buffer Under-Provisioning (1 KB Socket Buffers)

Table D.4 demonstrates the effect of severely constrained socket buffers (1 KB send/receive) with a 2 KB batch. The marked latency increase relative to Table D.2 justified the selection of 100 KB buffers in the production configuration.

Table D.4: *Under-provisioned buffers — 1 KB socket buffers, 2 KB batch,* `acks=0`.

Record (B)	Records	Records/s	MB/s	Avg (ms)	P50 (ms)	P95 (ms)	P99 (ms)
32	1 000	4 673	0.14	48.17	48	54	54
32	20 000	86 957	2.65	31.45	30	41	49
64	20 000	78 431	4.79	44.26	44	51	52

This page intentionally left blank.