INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Processamento de grandes volumes de dados em grafos

JOSÉ MANUEL FERREIRA DA CUNHA
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Processing large volumes of graph data

José Manuel Cunha

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Cybersecurity And Systems Administration

Advisor: Dr. Jorge Coelho

Porto, September 12, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 12, 2025

"Success is not final, failure is not fatal: It is the courage to continue that counts."

— Winston Churchill

Acknowledgement

I would like to begin by thanking my supervisor, Dr. Jorge Coelho, whose unwavering support, confidence, and encouragement made it all possible in the pursuit of this work. His expertise, patience, and constructive advice contributed in shaping the form and course of this thesis and, in fact, in my professional and educational development as well. I am sincerely grateful for the time he devoted in reading my work, for being ever accessible in order to put doubts straight, and in exhorting me towards greater reach whenever necessary.

Also, I would like to extend my deepest thanks to my university, ISEP – Instituto Superior de Engenharia do Porto, for giving me an intellectual environment, knowledge, and countless memorable memories and self-improvement years.

Specifically, I want to gratitude to Critical Techworks for continuing to motivate me to study and progress, and to my colleagues, who were so inquisitive about how my work was coming along and offered me great feedback and encouragement whenever necessary.

I wish to extend my gratitude to my family: my father, Eng. Manuel Cunha; my mother, Cristina Cunha; and my brother, Eng. João Cunha, whose unwavering support, encouragement, and faith in me, in spite of self-doubt, kept me on the rails and pushed me to be a better person and a better professional. Their constant support helped me aim at nothing less than the best and, eventually, to complete a Master's degree.

To all my friends, I express my profound gratitude. I appreciate your consistent presence during times when I needed to relax, your willingness to listen when I needed to express my frustrations, and also the joy, diversions, and motivational words that you offered significantly contributed to making this experience more manageable and enjoyable.

Lastly, my greatest and most enormous gratitude goes to my girlfriend, Filipa Nunes, my future attorney. You are my anchor and my strongest support. Your love, patience, and unwavering faith in me kept me going through the most difficult points in the master's. This achievement is not mine alone, but also yours.

Abstract

The increased complexity of product architectures in today's industrial environments demands the use of robust mechanisms to provide certainty and quality in real-time. Verification by manual means has become inefficient, and automated solutions must be accurate as well as speedy in handling high volumes of data and high rates of messaging. The challenge here is addressed by designing, implementing, and validating a quality verification service that can detect mismatches between expected and observed product trees, with the service seamlessly integrating into an already established infrastructure of messaging.

The suggested approach is founded on graph-oriented data models for the representation of product structures and employs algorithms designed to compare nodes, relationships, and attributes among various sources. A repository component has been created to maintain nodes and relationships, facilitating both sequential and parallel insertion modes. The integration of Apache Kafka was implemented to allow for real-time management of verification events. Validation of the system was conducted through unit and integration testing, augmented by performance assessment.

Performance evaluation focused on the ingestion paths, JSON file insertion and Kafka streaming, and was executed 50 times per scenario to ensure consistency. For JSON file inserts (Table 4.1), parallel processing reduced end-to-end runtime from 3–5 s to 150–180 ms at 1k records and from 140–150 s to 550–650 ms at 50k. For Kafka messaging (Table 4.2), sequential runtimes of 4–6 s (1k), 7–10 s (5k), 32–36 s (20k), and 115–121 s (50k) were cut to 0.8–0.9 s, 0.6–0.8 s, 10–12 s, and 38–42 s with parallel programming. These results demonstrate multi-fold speedups and robust scalability under high-throughput conditions.

In summary, the study illustrates that the real-time assessment of quality in intricate product configurations can be achieved through the integration of effective data structures, parallel processing techniques, and concurrent communication methods. The findings encompass a validated proof-of-concept that exhibits considerable enhancements in performance, an extensive testing framework to ensure accuracy, and a well-defined basis for prospective industrial application and scholarly investigation.

Keywords: Real-Time Quality Control, Graph Data Structures, Parallel Computing, Industry 4.0, Smart Manufacturing

Resumo

A crescente complexidade das arquiteturas de produtos nos ambientes industriais atuais exige o uso de mecanismos robustos para fornecer certeza e qualidade em tempo real. A verificação manual tornou-se ineficiente, e as soluções automatizadas devem ser precisas e rápidas no tratamento de grandes volumes de dados e altas taxas de mensagens. O desafio aqui é abordado através da conceção, implementação e validação de um serviço de verificação de qualidade que possa detetar discrepâncias entre as árvores de produtos esperadas e observadas, com o serviço a integrar-se perfeitamente numa infraestrutura de mensagens já estabelecida.

A abordagem sugerida baseia-se em modelos de dados orientados a grafos para a representação de estruturas de produtos e emprega algoritmos concebidos para comparar nós, relações e atributos entre as várias fontes. Foi criado um repositório para manter nós e relações, facilitando os modos de inserção sequencial e paralela. A integração do Apache Kafka foi implementada para permitir a gestão em tempo real de eventos de verificação. A validação do sistema foi realizada através de testes unitários e de integração, complementados por uma avaliação de desempenho.

A avaliação de desempenho focou-se nos caminhos de inserção, inserção offline de ficheiros em JSON e Kafka Streaming, sendo esta executada 50 vezes por cenário para garantir a consistência. Para inserções de ficheiros JSON (Tabela 4.1), o processamento paralelo reduziu o tempo de execução de ponta a ponta de 3–5 s para 150–180 ms em 1k registos e de 140–150 s para 550–650 ms em 50k. Para streaming Kafka (Tabela 4.2), os tempos de execução sequenciais de 4 a 6 s (1 mil), 7 a 10 s (5 mil), 32 a 36 s (20 mil) e 115 a 121 s (50 mil) foram reduzidos para 0,8 a 0,9 s, 0,6 a 0,8 s, 10 a 12 s e 38 a 42 s com a programação paralela. Esses resultados demonstram acelerações múltiplas e escalabilidade robusta em condições de alto rendimento.

Em resumo, o estudo ilustra que a avaliação em tempo real da qualidade em configurações complexas de produtos pode ser alcançada através da integração de estruturas de dados eficazes, técnicas de processamento paralelo e métodos de comunicação simultânea. As conclusões abrangem uma prova de conceito validada que exibe melhorias consideráveis no desempenho, uma estrutura de testes abrangente para garantir a precisão e uma base bem definida para aplicações industriais prospectivas e investigação académica.

Contents

List of Figures

Listings

List of Tables

List of Abbreviations

ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
ACK	**Ack**nowledge
ACM	**A**ssociation for **C**omputing **M**achinery
AI	**A**rtificial **I**ntelligence
AMQP	**A**dvanced **M**essage **Q**ueuing **P**rotocol
API	**A**pplication **P**rogramming **I**nterface
AQL	**A**cceptance **Q**uality **L**imit
AR	**A**ugmented **R**eality
ASSY	**Ass**embl**y**
ASYU	**As**sembl**y** **U**nit
AWS	**A**mazon **W**eb **S**ervices
BOM	**B**ill **o**f **M**aterials
BOMO	**B**ill **o**f **M**aterials **O**verlay
BSON	**B**inary **JSON**
CPS	**C**yber-**P**hysical **S**ystems
CPU	**C**entral **P**rocessing **U**nit
DAG	**D**irected **A**cyclic **G**raph
DB	**D**ata**b**ase
DBMS	**D**ata**b**ase **M**anagement **S**ystem
DCM	**D**ata **C**ollection **M**odule
DSR	**D**esign **S**cience **R**esearch
EBOM	**E**ngineering **B**ill **o**f **M**aterials
FURPS	**F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**upportability
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
IBM	**I**nternational **B**usiness **M**achines
ID	**Id**entifier
IoT	**I**nternet **o**f **T**hings
JSON	**J**ava**S**cript **O**bject **N**otation
MBOM	**M**anufacturing **B**ill **o**f **M**aterials
MQTT	**M**essage **Q**ueuing **T**elemetry **T**ransport
MRO	**M**aintenance, **R**epair and **O**perations
NACK	**N**egative **Ack**nowledge
NFR	**N**on-**F**unctional **R**equirement
NoSQL	**N**ot **o**nly **SQL**
OPC-UA	**O**LE for **P**rocess **C**ontrol — **U**nified **A**rchitecture
ORM	**O**bject-**R**elational **M**apping
POV	**P**oint **o**f **V**erification
PRISMA	**P**referred **R**eporting **I**tems for **S**ystematic **R**eviews and **M**eta-**A**nalyses
QVM	**Q**uality **V**erification **M**odule
RAC	**R**eal **A**pplication **C**lusters
RAM	**R**andom **A**ccess **M**emory

RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
REST	**Re**presentational **S**tate **T**ransfer
SBOM	**S**ervice **B**ill **o**f **M**aterials
SPC	**S**tatistical **P**rocess **C**ontrol
SQL	**S**tructured **Q**uery **L**anguage
UC	**U**se **C**ase
UI	**U**ser **I**nterface
UOM	**U**nit **o**f **M**easure
URL	**U**niform **R**esource **L**ocator
US	**U**ser **S**tory
WBS	**W**ork **B**reakdown **S**tructure
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

This chapter sets the context for the work with a description of the problem to be solved, the objectives to be achieved, the plan with the schedule of activities, the research methodology, and the structure in which this document is organised.

1.1 Context

Recently, the focus on quality control strategies has increased considerably and is based on the growing need for continuous improvement and product reliability [1].

Several factors have contributed to this growing need: the enormous pressure on production processes in modern industry, intensive competition in a global context and the increasing need for sustainable processes, given the rising prices of energy and raw materials [2].

In addition to the individual approaches mentioned, these demands have made it essential for all industries to reduce waste and improve efficiency, mainly through intelligent quality software and integrated data-based technologies as part of Industry 4.0 initiatives [2].

Moreover, real-time data monitoring assists in on-the-spot detection of any inefficiency in the processes involved in the manufacturing of any item. Their continuous analysis for resource usage and operational performance ensures the optimum use of energy and materials. As an example, real-time adjustment of production parameters can reduce energy consumption by 20%, which is quite economical for high resource-consuming industries. It does this by not only reducing wastage but also making production sustainable and economically viable [3].

Graph data structures represent relationships and dependencies between complex systems, such as production lines. They can help against those demands of efficient processing of large volumes of data that coop with the ever-increasing complexity and scale of industrial operations [4].

Graph-based insights into real-time monitoring systems provide the quickest identification of the root cause of defects. As a consequence, these advantages have guaranteed defect rate reductions of up to 30–50% in high-volume production lines [5]. Also, it accelerates decision-making, as reported, that such data integration reduces the time for making decisions by as much as 50%, compared to traditional systems [3].

This approach ensures that each step is interconnected and enables an overview of the operations taken during the production. With this, graph data processing is essential in industrial steps from the need to maintain consistent quality and minimize defects in high-volume production [6].

1.2 Problem

In modern production lines and especially in more complex scenarios, it is critical to determine whether a specific product meets quality standards during the manufacturing processes [7].

This problem, in its real essence, will involve comparing, in real-time, a product's characteristics at critical points along the production line with their desired ones. Such a comparison should be both initiated and completed within a short time. Any product that does not meet the desired characteristics must be removed from the production line without wasting more time. It saves waste and assures the restoration of non-conforming products.

The challenge is to determine how to perform such a comparison efficiently using enhanced parallelism and concurrency techniques. The usage of such techniques will show how the process can be accelerated and how the implications on the functionality of the production line can be minimised. Adding another layer of complexity in managing and analysing data, the product tree needs to be created in real-time using a scalable data-streaming solution.

Another relevant point is the introduction of a *product tree* — a hierarchical structure that represents the expected product along with its subcomponents and relationships. The user will be required to manually construct and destroy an expected product tree, which they will need to enter on a dedicated platform. To ensure efficiency, this process should be supported by choosing one database and technology stack that offers the best possible performance to manage this type of hierarchical data.

Furthermore, it is imperative to acknowledge that the various data types inherent to this process necessitate disparate persistence approaches. It is evident that data originating from the production line, including measurements and physical characteristics of products, is inherently dynamic and high-frequency in nature. This necessitates immediate processing to ensure the efficacy of the data. In such cases, the utilisation of real-time ingestion and analysis mechanisms, such as Kafka, is particularly advantageous.

Conversely, the anticipated product configuration, which is entered manually by the user, is more static in nature and needs mechanisms that ensure integrity, versioning and efficient queries. The selection of a database management system (DBMS) depends on the complexity of the structure in question. Relational, graph, or non-relational DBMS can be utilised, depending on the specific requirements of the database. It is therefore evident that divergent objectives within the production process necessitate disparate data persistence technologies.

As already described, real-time quality checking has important effects on energy usage, waste minimisation, and prompt decision-making. Research reveals that production facilities can decrease energy use by 20% or less with real-time control, and scrap tracking can decrease material waste by between 15-23%. Additionally, IoT-based real-time quality solutions have been shown to achieve up to 18% reduced energy use, a 15% increase in resource efficiency, and greenhouse gas emissions decreases above 20%. Therefore, prompt detection of deviations not only guarantees better use of resources but also greatly minimises material waste and allows corrective action to be implemented within a far shorter timeframe than previously [8].

Designing an efficient and scalable solution for real-time quality verification that ensures integrity in the process of production while optimising computing resource utilisation is the key issue in this problem.

1.2.1 Objectives

By analysing the problem, a main objective was identified: It is imperative to ensure that each product meets predefined quality standards.

This objective gives rise to several technical and architectural challenges. In essence, the process involves the real-time comparison of a product's characteristics at critical points along the production line with the desired specifications. To undertake such a comparison, it is necessary to ensure that it is initiated and completed within a very short timeframe. Products that fail to meet the expected characteristics must be removed from the production line without delay to avoid further resource consumption and ensure prompt restoration.

To achieve this objective, several tasks must be addressed:

1. Literature search and review:
    - Study of the evolution of the Industry.
    - Study of where Bill of Materials and Event Streaming enter in the Key Technologies of the Industry 4.0.
    - Study about the importance of Quality Control.
    - Comparison between relational, non-relational and graph databases.
2. Event Streaming Integration:
    - Use an event stream to simulate real-time data in simultaneous production lines.
    - The product tree must be dynamically created in real-time, which requires managing high-throughput data streams and maintaining responsiveness.
3. Real-Time Quality Verification System:
    - Identify the technical and operational requirements of the quality verification system.
    - Mechanism that allows for instant assessment of product attributes against a set of quality standards, by employing parallel and/or concurrent processing methodologies to reduce latency in the system's design.
4. Management of Expected Product Structures:
    - Manual entry of the expected product tree by users on a dedicated platform. This necessitates the selection of a database and technology stack that provides optimal performance for storing and querying this data.

1.3 Work Breakdown Structure

Appendix A contains the detailed Work Breakdown Structure (WBS), which covers the hierarchical decomposition of the project into workable sections. The WBS graphically shows how the project will be organized internally in a detailed structure, showing task relationships from high-level objectives to specific deliverables.

These range from **Data Collection & Preparation** (1.1), consisting of Identification of Articles (1.1.1), Cleaning of Data (1.1.2), and Final Selection (1.1.3). Further, **Project Planning** (1.2) consists of Project Charter (1.2.1), Work Breakdown Structure (1.2.2), and Planning (1.2.3).

Further, under **Literature Review** (1.3) is the State of the Art (1.3.1) analysis within the concerned domain. **Analysis and Design** (1.4) are then conducted through Problem Domain (1.4.1) definition, Requirements (1.4.2) Outline, and presenting the design structure through Architecture (1.4.3.1) and Levels of Design (1.4.3.2).

Then comes the development of the **Solution** (1.5) as its realization under Backend (1.5.1.1) and Front-end (1.5.1.2). In **Evaluate** (1.6), Testing (1.6.1) and Solution Validation (1.6.2) will ensure the solution's quality and its objective alignment.

Last but not least, **Conclusions** (1.7) are reported as Final Report (1.7.1) and communicated through Final Presentation (1.7.2).

1.4 Approach

An Agile (Iterative) methodology is the selected strategy that will be applied to this work. It shall be implemented through two-week sprints in order to provide flexibility, adaptability, and constant movement on the main tasks. It provides iterative development of deliverables with feedback loops and incremental improvements.

The nature of the adopted method will make the project dynamic and responsive to changes, and it will ensure the quality of the outcome at the end of each sprint by validating with the advisor, and finally at the deadline of the whole project.

1.5 Project Schedule

In each of these sprints, as you can see on the Appendix B, the Work Breakdown Structure (WBS) has certain work activities that guarantee small, sharp steps in the activity's development. This is the critical schedule for the sprints that allows all milestones to be completed in a timely fashion without losing the flexibility that will enable every change that may be necessary at whatever juncture of the project timeline. The Gantt chart of the project schedule is attached to Appendix C.

1.6 Research Methodology and Strategy

The Research Methodology and Strategy section outlines the approach adopted for conducting this study. The Design Science Research (DSR) methodology was selected and applied to ensure rigorous analysis and practical relevance by aligning the industry practices with academic rigour [9].

1.6.1 Design Science Research (DSR)

Design Science Research focuses on developing and assessing artefacts to put forward and help solve real-world problems. In this regard, the notion of an artefact may refer to constructs, models, methods, and instantiations, including social innovations and things not in existence before, such as new properties of resources [10].

The underlying philosophy in design science is that the understanding and solving of a design problem are achieved by constructing and applying a purposeful artefact. Such an artefact should be useful in providing utility, novel, and properly evaluated to prove its contribution [11].

Alen Hevner conceptualized DSR as a process comprising three interconnected cycles, namely [10]:

- **Relevance Cycle**: Links the research project to its contextual environment.
- **Design Cycle**: Iterates between building and evaluating the artifact.
- **Rigor Cycle**: Links the research activities to the existing knowledge base.

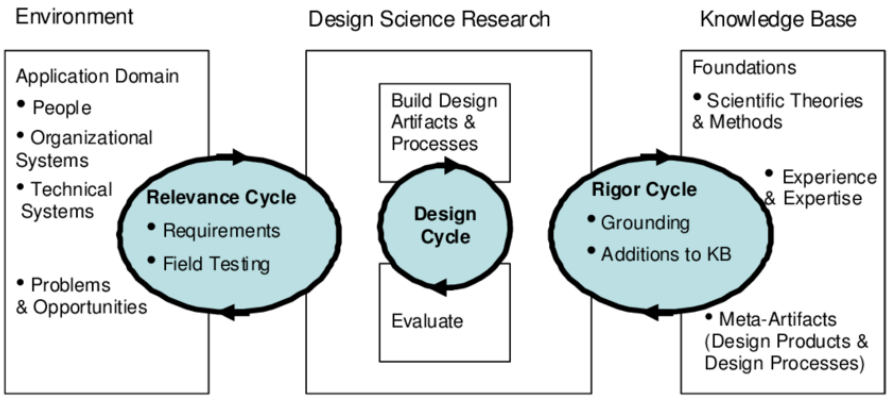

Figure 1.1: Design Sciences Research Cycles (retrieved from [12]).

Other scholars provided other useful guidelines to perform DSR projects, that includes:

- **Ken Peffers** outlined six activities that comprise problem identification, defining the objectives, design and development, demonstration, evaluation, and communication [13].
- **Roelf J. Wieringa** described two cycles: the **Design Cycle** for developing and evaluating the artifact and the **Empirical cycle** for scientific investigation [14].
- **Per Runeson** emphasized problem conceptualization, artefact design, and validation, with recommendations for practice framed as technological rules [15].

These forms of DSR differ in the role of empirical studies. Whereas some confine empirical methods to the evaluation of the artefact, others propose using them to understand the problem or to develop the artefact [11].

Phillip Offermann proposes structuring DSR into three problems: identification, solution design, and evaluation supported by specific research methods such as literature reviews and expert interviews. Such a phased approach fits well with industry-academia collaboration in software engineering [16].

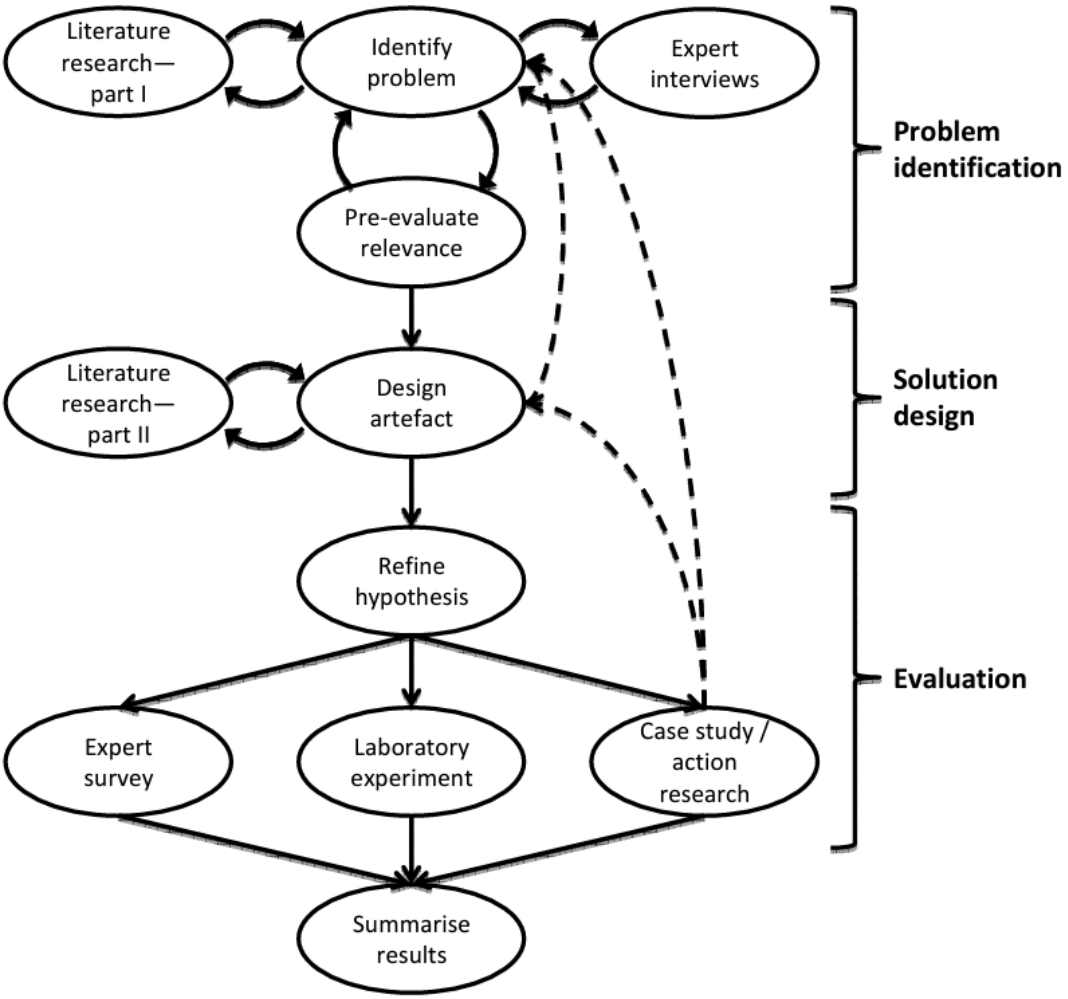

Figure 1.2: Proposed Research Process by Offermann (retrieved from [16]).

1.6.2 Applied Methodology

By applying these cycles to the dissertation, the following is achieved:

1. **Relevant Cycle**:
    - The gathering of requirements regarding the high-volume data performance of the database, integrated with Kafka, while ensuring that the user interface remains user-friendly.

2. **Design Cycle**:
    - Clearly develop the necessary requirements for database performance, Kafka integration, and the user interface.
    - Create the artefacts and test and refine them iteratively in a controlled environment.
3. **Rigor Cycle**:
    - Conduct a systematic study of available methodologies and technologies concerning relational versus non-relational database performance and parallel processing techniques.

1.6.3 Research Question and Data sources

This dissertation focuses on the field of manufacturing technology development, including data structure, performance analysis of database, and parallel processing methodology that would contribute to improving productivity.

Against the above background, the following research questions (RQs) are identified:

1. How will the modern quality control strategy be further improved for high-volume and complicated production lines, keeping view of product reliability and sustainability?
2. What differences in performance in real-time production data management are observed in relational databases and in non-relational databases regarding their performance, speed, and fault tolerance?
3. In what ways is it possible to adopt an advanced parallel and concurrent processing technique to perform real-time quality verification of a product in the course of its manufacturing process?

Identifying the data sources for the systematic review is the initial phase of the research process. Table 1.1 lists the selected electronic databases.

Table 1.1: Electronic Database

Identifier	Database	URL
ED1	Google Scholar	https://scholar.google.com/
ED2	IEEE Xplore	https://ieeexplore.ieee.org/Xplore/home.jsp
ED3	ScienceDirect	https://www.sciencedirect.com/
ED4	ACM Digital Library	https://dl.acm.org/
ED5	Springer Nature Link	https://link.springer.com/

1.6.4 Search Query, Keywords and Criteria

Given that the investigation focuses on real-time quality control, database efficiency, and the application of graph data structures for the administration of product trees, two questions have been put together. Each question specifically addresses a different dimension of the current study and, as such, methodically covers the relevant literature.

SQ1 - Real-Time Quality Control and Database Perfomance

This first search query (SQ) covers real-time quality control, the performances of relational and non-relational databases within manufacturing systems, parallel processing, relational databases, non-relational databases, NoSQL, Kafka, and manufacturing analytics.

It has been formulated as follows:

"(Real-time quality control OR manufacturing analytics) AND (parallel processing OR concurrency) AND (relational databases OR non-relational databases OR NoSQL) AND Kafka"

The inclusion criteria of this search query are as follows:

- The source addresses either a methodology of real-time quality control or the performance of a database, such as relational or non-relational, in an industrial context.
- In a comparison paper, relational and non-relational databases were compared with reference to the performance parameters-latency and throughput.
- Discussion on recent parallel/concurrent processing methodologies for data-intensive applications.
- Practical examples of Kafka with reference to its manufacturing application to achieve real-time data streaming.

All the sources which do not address issues related to manufacturing, do not contain practical examples, or were published in any language other than English or Portugues will be excluded.

SQ2 - Graph Data Structures and Product Tree Interfaces

The second query seeks to apply graph data structures in modelling relationships and dependencies, especially in managing the product tree. Besides, it includes the development of interfaces for the visualization of hierarchical data and its update in real-time. Thus, keywords are graph database, graph data processing, product tree modeling, hierarchical data structure, real-time update, and user interface design.

The query goes as:

"(Graph databases OR graph data processing) AND (product tree modelling OR hierarchical data structures) AND (real-time updates OR user interface design)"

One can confirm that, by inclusion criteria, the source belongs to the category below based on:

- Graph data structures applied in modelling relationships within the manufacturing process.
- Designing, enhancing, or improving the user interfaces applied in the management of product trees.
- Examining the effects that will be brought about by real-time updates in implementing and integrating data in these manufacturing systems.
- Evaluation of the implementation of real-time updates and data integration into manufacturing systems.

Studies not applied, studies not about graph data structures applied in manufacturing systems, and studies whose publications are not in English or Portuguese will be excluded.

1.6.5 Data Collection Process

With the presentation of the previous step, and to obtain the applicable articles Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA) process was employed [17]. It consists of a process in three stages:

1. Identification involved all the documents from the data sources that met the criteria covered. It was discovered a total of 173 relevant articles.
2. Screening, further dividing into two sections:
   (a) It started with an abstract screening carried out on all found articles. Within this section, the articles were classified as either "Relevant", "Possibly Relevant" or "Irrelevant". Then, "Possibly Relevant" were subjected to a secondary screen to be classified as either "Relevant" or "Irrelevant". This gave a total of 79 news items labelled as "Relevant".
   (b) Then, it was a critical reading of some relevant articles selected. During a research effort aimed at finding literature appropriate to research questions. Finally, 36 of these articles were discovered to provide a response to at least one research question.
3. Inclusion includes collecting all data from other articles which could be used for research.

By using this cautious manner, it was possible to ensure the choice of reliable literature, increasing the validity and credibility of the study findings.

1.7 Document Structure

This document is organized into several chapters to show how the conducted research flows logically.

The first chapter, *Introduction*, provides a succinct overview of the research topic. It states the problem and defines the objectives the study will tackle. In addition, this chapter develops a detailed plan that describes the structure and progress of the research by identifying major development stages and including specific milestones that correspond to the set goals. A methodology of the research has been added to emphasise which methodological framework will be selected and how it has been applied.

The second chapter, *State of the Art*, goes in-depth into the industry's developments from Industry 1.0 to date, presenting the most important technological advancements. Furthermore, it explains in detail the Bill of Materials (BoM) concept, presenting it as an important structure for the manufacturing sector and showing how it can be represented in a graph so that relationships and interdependencies during production are visible. The chapter also looks into the performance differences between relational, non-relational databases and graph databases. Parallel processing techniques are also discussed as essential tools to improve computational efficiency in handling large datasets.

The third chapter, *Analysis and Design of the Solution*, defines the system requirements by utilising use cases and both functional and non-functional specifications. It puts forward the FURPS+ classification scheme, clarifies the quality attributes, and explains the design of the solution being put forward. The architectural design is explained with respect to the layers it is composed of (presentation, backend, and data), whilst processes such as development of expected product trees and quality verification workflow are shown by the use of diagrams.

The fourth chapter, named *Solution Implementation*, describes the process of translating the design into a working system. It explains the combination of the hardware components, like sensors and Programmable Logic Controllers (PLCs), with the software structure, the implementation of basic modules (Data Consumer, Quality Verification Module, API Gateway, and Frontend), and the use of technologies like Angular, Quarkus, Neo4j, and Kafka. Also, it explains the inclusion of parallel programming at the repository layer. The chapter ends with a subsection dedicated to test and validation, including unit, integration, and performance tests.

The fifth chapter, *Conclusions*, provides a summary of the objectives met, presents limitations and validity threats, and gives directions for future research work. It ends by providing the final evaluation of the targeted solution and its applicability to the industry.

Chapter 2

State of the Art

This chapter discusses the evolution in industrial technology, charting the path from Industry 1.0 through to Industry 4.0, specifically through the lens of the emergence of smart manufacturing and the central presence of the Bill of Materials (BoM). In such a context, event streaming has emerged as a predominant solution, facilitating real-time data interaction between heterogeneous systems and enabling seamless integration, monitoring, and adaptive actions in production environments. Subsequent discussion on data handling in the modern world sets relational databases against NoSQL and graph-based approaches. The evaluation of performance issues focuses on how different database models manage structured, semi-structured, and data-heavy relationships under different workloads. The chapter ultimately highlights the need for parallel and concurrent computing to enhance processing capacity.

2.1 Evolution of Industry

This section addresses the industrial landscape evolution, which has been segmented into four distinct phases:

- **Industry 1.0**: symbolized by the mechanization era;
- **Industry 2.0**: characterized by the mass production concept;
- **Industry 3.0**: identified by the automation and computerization wave;
- **Industry 4.0**: characterized by intelligent, data-driven manufacturing.

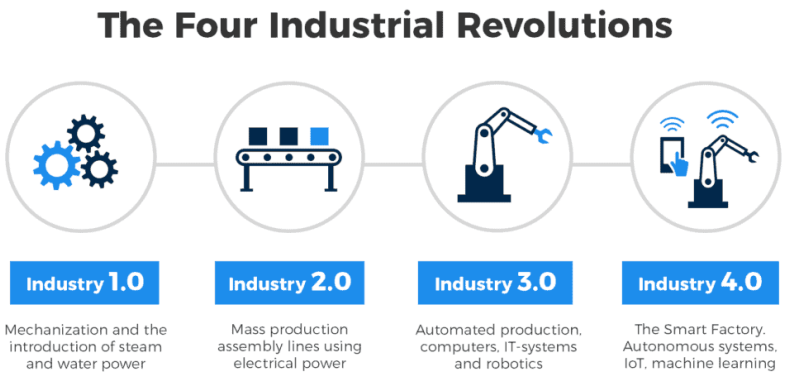

Figure 2.1: Industry Evolution (retrieved from [18]).

2.1.1 Industry 1.0

The First Industrial Revolution was between approximately 1760 and sometime between 1820 and 1840, when steam power and the mechanization of production processes were heavily used [19]. With high mechanization, productivity became much higher, sometimes many times greater, than traditional methods. For instance, simple spinning wheels were vastly accelerated with mechanical spinning systems [20].

While steam power was nothing new, its application for industrial purposes constituted a breakthrough in the tremendous gain of human productivity. It was a fundamental transformation from the conventional manual methods of production to machine-based manufacturing processes. The characteristic elements that made this era of rapid industrial growth possible included steam power, hydropower, and many technological and infrastructural innovations [20].

The revolution catalyzed progress in a variety of sectors, including textile manufacturing, the iron industry, machine tools, and chemical production. Further, considerable development occurred in cement, gaslighting, glassmaking, agriculture, and paper machinery. In terms of transportation infrastructure, massive improvements were reflected in canal construction and improved waterways, railways, and roads. All these laid the foundation for significant economic and social changes [21].

2.1.2 Industry 2.0

The Second Industrial Revolution was a great change because new technologies and industries were introduced throughout the late 19th and early 20th centuries. In contrast to the First Industrial Revolution, the introduction of electricity, chemicals, and steel production characterises this period. It fundamentally transformed manufacturing and transportation. Several key inventions — such as the internal combustion engine and electric power generation — radically changed the means of production and daily life. Such inventions enabled mass production and the creation of new industries that transformed economies and ways of life across the world [22].

Besides this, the value of electricity was well realized for industries in this era, whereas revolutionary communication media such as the telephone and telegraph emerged. In the chemical industry, one witnessed the dawn of the development of synthetic materials like plastics and the perfection of fertilisers for higher agricultural productivity. The innovations in the production of steel, including the Bessemer process, made many construction and infrastructure projects that were erstwhile unviable and created cheaper alternatives. In the last technological leaps, the direct result has been urbanisation and a developed network of transportation—railways, automobiles, and later aviation. [23].

2.1.3 Industry 3.0

The Third Industrial Revolution is the one that occurred in the 1970s of the 20th century and is also called the Digital Revolution because the era was booming with electronics, information technology, and telecommunications. During that era, it shifted toward digital technologies from mechanical and electrical systems. It unleashed so many industries with huge automation, reducing human labor input within the production process. Of them, central ones were computers, microprocessors, and the Internet in the communication and transformation of industrial operation [20].

Above all, at the core of this revolution was the use of Programmable Logic Controllers, among other forms of automation, which enabled industries to attain more accuracy and efficiency. Computers began replacing manual control in many procedures while software systems started to perform inventory, logistics, and resource planning tasks. The semiconductor industry, driven by the integration of silicon and germanium, was one of the bedrocks that drove this revolution since this made the production of integrated circuits and digital devices [21].

2.1.4 Industry 4.0

Industry 4.0, also called the Fourth Industrial Revolution, refers to the digitalization of production and manufacturing processes due to higher-order technological integration. Building on the digitalization born from the Third Industrial Revolution, with a focus on establishing more connections, data flow, and smart automation, Industry 4.0 basically finds embodiment in constituents like CPS, IoT, and cloud computing [21].

The key technologies driving Industry 4.0 are [24]:

1. **Internet of Things (IoT)**: Machine, device, and system communication for effective real-time data interchanging and decision-making;
2. **Artificial Intelligence and Machine Learning**: Predictive maintenance, quality control, and process optimization are improved through the inference of large volumes of data;
3. **Big Data Analytics**: Large volumes of data from various sources are gathered and transformed into action through strategic decisions for better operation efficiencies;
4. **Cyber-Physical Systems**: Computation, networking, and physical processes are integrated in such a way that digital and physical components may be allowed to interact smoothly;
5. **Cloud Computing**: Scalable resources and storage solutions that enable collaboration, including data access across the enterprise;
6. **Advanced Robotics and Automation**: Employ intelligent robots for complicated tasks that require hardly any human interference. This enhances productivity and precision;
7. **Additive Manufacturing**: In short, 3D printing makes rapid prototyping and customized production, reducing time to market and material wastes;
8. **Augmented Reality and Virtual Reality**: They help design, train, and conduct maintenance while digitally visualizing information in the real world;
9. **Cybersecurity**: Protect interconnected systems and data against cybersecurity attacks, thus assuring the integrity and reliability of the operation.

However, Industry 4.0 adoption also presents quite a number of challenges with huge investments in new technologies, specifically skilled workforces that are competent in the use of digital tools, and concern for security and privacy issues. Such challenges have to be overcome if Industry 4.0 is to work and its benefits are to materialise [25].

In addition to these core technologies, balanced integration and alignment of data flows in distributed platforms are also important in realizing Industry 4.0 capabilities in its full sense. Since intelligent factories generate immense amounts of data simultaneously from IoT devices, robots, sensors and control systems, there is a great need for fault-tolerant data streaming and messaging platforms. This is where messaging brokers like Apache Kafka [26], RabbitMQ [27], and other such platforms become critical in providing the underlying foundation for real-time communication and data-driven automation.

2.1.5 Real-Time Messaging and Event Streaming in Industry 4.0

In the context of Industry 4.0 and modern management of production lines, the real-time exchange of data is essential to drive production efficiencies, reduce downtime durations, and to allow autonomous responses to events within the manufacturing landscape. The manufacturing environment requires the uninterrupted data flow between machines, sensors, quality management systems, and enterprise systems with low latency [28].

The Role of Real-Time Messaging

Sending messages in real time makes it possible to create precision, flexibility and resilience in production environments. An event, which could be the detection of an anomaly by a sensor, the completion of a task by machines or the activation of a quality control notification, needs to be transmitted to the relevant systems in good time.

This type of communication can provide operators with relevant information, for example that operations are interrupted or that downstream processing is changed immediately. Message brokers provide a critical infrastructure for communication, abstracting data producers (machines and sensors) from consumers (control panels and instrumentation) and providing reliable, high-speed and horizontally scalable interactions [28].

Examples of Real-Time Messaging Applications in the Industry

Apache Kafka: is a distributed data store optimised for ingesting and processing streaming data in real time, used primarily for using streaming data, particularly in real time, and applications that adapt to that data. It combines messaging, storage and stream processing to enable storage and analysis of both real-time and historical data [26].

Figure 2.2: Apache Kafka Logo

RabbitMQ: is an open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It acts as a messenger between applications, facilitating communication by sending and receiving messages. Developed in the Erlang programming language, RabbitMQ is known for its efficiency and ability to handle a large number of concurrent connections [27].

Figure 2.3: RabbitMQ Logo

MQTT: Message Queuing Telemetry Transport (MQTT) is a publish-subscribe messaging protocol designed for efficient communication between devices, particularly in the Internet of Things (IoT) ecosystem. The protocol follows a publish-subscribe architecture, where clients can publish messages to a broker, and other clients can subscribe to specific topics to receive relevant messages [29].

Figure 2.4: MQTT Logo

2.2 The Bill of Materials (BOM) and Graphs Data Structures

2.2.1 Bill of Materials (BoM)

A bill of materials, also known as a BOM, is a structured list of all materials, components, subassemblies, or intermediate parts that go into creating a final product. It gives a detailed inventory and blueprint that must be followed to understand the hierarchy of components and their relationship with each other, thereby stating what will be required to make a product, the relationship of components, and in what quantity. The bills of material are basic to most manufacturing, software development, and maintenance management, as they enable production planning, inventory management, cost estimation, and lifecycle tracking [30].

Types of BoM

There are many types of BOMs, and each serves a different purpose at different points in life and according to industry needs. The major ones are explained as follows:

- **Traditional and Engineering BOM (EBOM)**

  In manufacturing contexts, a bill of materials implies the product structure listing all the items that constitute a product. This normally applies in design and production contexts [31].

  The EBOM is developed during the product design phase and describes the components, assemblies, and relationships that comprise the product. For instance, the study on transforming the EBOM into a Maintenance BOM shows that the EBOM is important in mapping the product design to other lifecycle functions [31].

- **Maintenance Bill Of Material (MBOM)**

  The MBOM is then represented, which is obtained from EBOM but modified to capture the pragmatic needs of maintenance, repair and overhaul (MRO). The additional attributes can be Service History, Demand for Spare parts, or Physical Usage of equipment [32].

- **Software Bill of Materials (SBOM)**

  The SBOM is a special type of BOM utilized within software systems. It lists all the constituents, dependencies, and sub-dependencies of a software product with the aim of guaranteeing safety and transparency within software supply chains [33].

  SBOMs are some of the most important things. They are key to trying to mitigate things like supply chain attacks because they give very vital details about the makeup of the software. Some of the commonly used formats within SBOMs include SPDX, CycloneDX, and SWID Tagging [33].

- **Generic Bill of Materials and Operations (BOMO)**

  This is more holistic, integrating traditional BOMs with routing and operational information to support integrated product and process management in high-variability production systems. BOMO facilitates the production planning and costing processes and change management by providing a common structure [31].

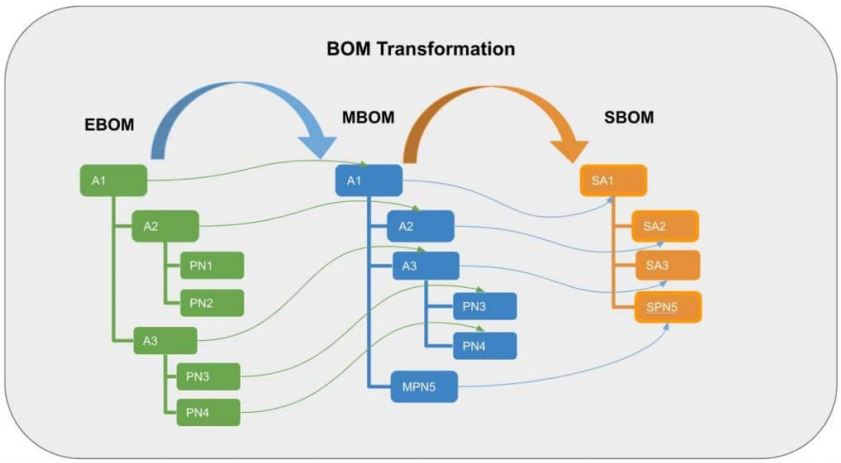

Figure 2.5: BOM Transformation (retrieved from [34]).

2.2.2 Role in Industry 4.0

The Bill of Materials is turning out to be a major influence in modern manufacturing, especially under the wings of Industry 4.0. In this way, BoM changes from a component list into a living, data-driven tool, enabling a series of stages within the ambit of the manufacturing process.

Foundation to Smart Manufacturing

Operating capability for digital twins and smart systems industries in the core of the BoM. Indeed, BoM began with simple lists denoting material and components, but today they have grown to return something dynamic to IoT, digital twins or other smart technology. More about what materials or parts/assemblies in production are able to provide and collect, updated information at the hands of automatic systems, allowing in-line or real-time control and improvements to optimize the operational process of an organization in order to reach one goal: accuracy [35].

It follows that digital twins, virtual models of physical assets, processes, or systems drive their functionality with more and more accurate and timely BoM data. A BoM in Industry 4.0 does not just list the components needed for a product and connects those listed parts to their real-life equivalents to build an accurate digital twin. Such a model is used for simulations, monitoring, and optimising processes to help manufacturers predict and prevent problems before they occur [35].

Optimization of Production Flexibility and Agility

In contrast, one of the core messages of Industry 4.0 is about how enterprises would have to respond much faster to shifting market conditions and changing consumer demand. In a strong, perceptible manner, the BoM supports its endeavour toward a flexible and agile manufacturing system by supplying comprehensive, timely information about the materials and components used in manufacturing. This will result in the easy and swift reconfigurations of production lines or processes, if necessary, to enable quick responses to new product designs or revised production schedules [36].

Supporting Sustainability Goals

In recent times, sustainability has become one of the major concerns of manufacturers in the era of Industry 4.0. The BoM is important when considering the optimization of the use of resources, reduction of waste being generated, and minimization of the environmental impact during production. By tracking the life cycle of every material, raw material to end-of-life manufacturing can be optimized to consume less energy and produce less waste, all in line with sustainability goals [35, 36].

Traceability Assurance and Quality Control

The traceability of components during production is fundamental in ensuring that quality thresholds are met by the products and that the products adhere to set regulations. Linking the BoM to real-time monitoring systems during manufacture ensures that each component is tracked accurately from assembly to the testing of the final product. It enhances quality control because defects or discrepancies are identified rather quickly and allows predictive maintenance of equipment from real-time data [35].

2.2.3 BoM modelled as Graphs Data Structure

In order to be able to fully understand the representation of a Bill of Materials (BoM) using graphs, it is necessary to have a basic understanding of graph basics as they relate to data structures.

Graphs in data structures

A graph is a powerful and efficient method to present how various objects are related, linked, or structured. Different from simple linked lists or arrays, which consist of straight lines, graphs have the capabilities to represent more complex things such as a Bill of Materials (BoM), a file system, a map, or a network of individuals where objects may be linked in various ways. A graph is composed of nodes or vertices and lines or edges that join the nodes. The lines may be directed or undirected and may also contain additional information, such as weight that may indicate distance or cost [37].

A graph (G) is normally presented as G(V, E), where [37]:

- **V (Vertices/Nodes)**: Distinct points or sections on the graph. Examples: locations on a map, individuals on a social network, or components on a Bill of Materials.
- **E (Edges)**: connections or ties that bind two ends. Each edge possesses two ends and may be directed or weighted.

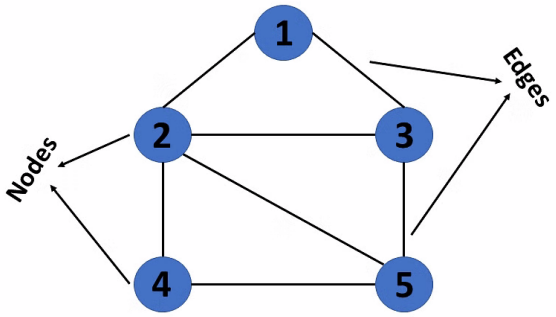

Figure 2.6: Example of a Graph

Types of graphs

There is a lot of types of graphs, but these are commonly used in real-world applications [37]:

- **Undirected Graph**: Edges have no direction and relationships are bidirectional.
- **Directed Graph**: Each edge has a direction from one vertex to another.
- **Weighted Graph**: Each edge has a numeric weight (e.g., distance, cost, time).
- **Unweighted Graph**: All edges are equal (no weights).
- **Cyclic Graph**: Contains at least one cycle (a path where the start and end nodes are the same).
- **Acyclic Graph**: Contains no cycles.
- **Directed Acyclic Graph (DAG)**: A directed graph with no cycles.

Convert BoM to Graph Data Structure

There are several ways a Bill of Materials could be modelled into a graph. A BOM is somewhat tree-like - it represents a structure of parents/granular level components and sub-assemblies that constitute or build a given product. If translated to a graph, this then becomes a direct and acyclic graph (DAG) [38].

One such realization could be done like [38]:

1. **Nodes**: Represent components, sub-assemblies, or raw materials.
2. **Edges**: Represent the relationships or dependencies between parent and child components
3. **Weighting Edges**: The weights of the edges can represent quantities of materials or components required

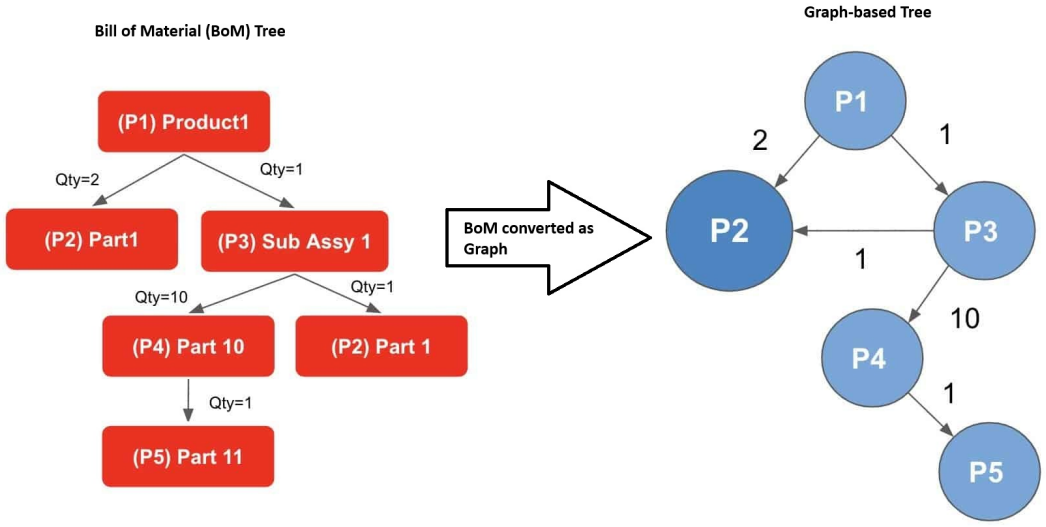

Figure 2.7: BoM converted as Graph (retrieved from [39]).

2.3 Quality Control

Quality control is the backbone of every form of production, making quality consistent and efficient and setting safety and performance standards that depend directly on it. In modern production, quality control thus immediately influences logistics, productivity, and customer satisfaction since industries in every field depend on strong quality assurance systems to be able to stay competitive in world markets [40].

Here are key reasons why quality control is vital [41]:

- **Assurance of Uniformity in Products and Customer Satisfaction**: Quality control ensures product conformance to specified standards and consequently minimises variance in production. Uniformity generates consumer confidence and helps in brand image development and long-term competitiveness.
- **Reduces Waste and Improves Resource Efficiency**: With early defect detection in the production process, quality control efficiently reduces rework requirements, minimises scrap, and reduces downtime. This translates into better utilisation of resources, lower production costs, and a direct boost to sustainable manufacturing practices.
- **Enhances Compliance with Regulations and Risk Avoidance**: Several industries operate subject to rigorous quality and protection standards. Quality control makes compliance with these requirements possible, reduces liability risks, and helps companies avoid costly recalls, penalties, or harm to their image.

Quality control is not just defect detection but also system responsiveness optimization. Truly integrated quality control systems and production logistics can ease the operation of production in the lines, and also can smooth various buffers and inspection points, addressing issues related to quality. As seen from the analysis of production design and influence on the spread of variability across manufacturing stages [40].

Manufacturing companies face ever-increasing competition today, and raw material costs continue to rise. These are factors that companies cannot control, but they need to focus on what they can control: their processes. They must work towards continually improving it to enhance quality, efficiency and reduce costs. Many companies still rely only on inspection after production to detect quality issues. By monitoring the performance of a process in real-time, the operator can detect trends or changes in the process before they result in nonconforming products and scrap [42].

2.4 Database Performance

2.4.1 Relational Databases

Relational databases are those kinds of databases that store and provide access to interrelated data, organized as tables. Each record in a table is uniquely identified by a primary key, while columns contain the attributes of the data. Such an organization makes it easy for one to understand and manage relationships between different sets of data [43].

The relational model distinguishes between logical data structures, such as tables, views, and indexes, and physical storage setups. This allows administrators to manage the physical storage independently of logical data access. Moreover, relational databases use integrity constraints to guarantee the validity and consistency of data, including mechanisms to prevent the occurrence of duplicate rows in a table [43].

Consider a small business using two tables to process product orders:

- Customer Information Table:
  - Columns: Name, Address, Contact Information, etc.
  - Primary Key: Customer ID
- Customer Orders Table:
  - Columns: Customer ID, Product, Quantity, etc.
  - Foreign Key: Customer ID (linking to the Customer Information Table)

The database correlates orders to their respective customers through the customer ID, enabling efficient and accurate order processing.

2.4.2 Benefits of the Relational Database

Relational databases have a few advantages that make them popular for managing structured data, including:

1. **Data Consistency and Integrity**: Relational databases store data in interrelated tables, hence encouraging accuracy and coherence of data in the system. Such an organization structure limits redundancy and maintains data integrity through predefined constraints and relationships [44].
2. **Easy Access and Update of Data**: The use of SQL allows very efficient querying, updating, and general management of data. SQL has very powerful commands to filter, sort, and summarize data and enables complex analysis and reporting [45].
3. **Support for ACID Transactions**: Relational databases support the Atomicity, Consistency, Isolation, and Durability (ACID) properties to have sound transaction processing. It would mean that all the database transactions are processed reliably and maintain the integrity of data even in the event of any failure in the system [45].
4. **Scalability and Flexibility**: Relational databases handle large volumes of data and hence are scalable for growth. They provide flexibility in designing database schemas that can scale with changing business needs [46].

5. **Data Security**: RDBMS offers robust security features such as user access controls and privilege management, also it allows access to data or manipulation of the data to every user as permission is granted to them [45].
6. **Data Independence**: The fact that logical data structures exist independently of physical storage in relational databases provides reality to changing storage without impairing data accessibility and vice-versa, hence maintaining systems that are easily maintainable [45].
7. **Advanced Query and Join Support**: Relational databases support the integration of data from multiple tables using joins, leading to in-depth data analysis and reporting [45].

2.4.3 Examples of RDBMS

Oracle Database

Oracle Database is one of the most common relational database management systems, distinguished by its scalability and performance and suitable for enterprise environments. It supports large applications; hence, organizations can manage large volumes of data with reliability and efficiency. It has high functionality, with options such as Real Application Clusters (RAC) for high availability and Oracle Data Guard for disaster recovery. This database is used in the worlds of finance, supply chain management, and customer relationship management for applications that are mission-critical [47].

Figure 2.8: Oracle Database Logo

MySQL

MySQL is an open-source relational database system widely used with Web applications and in small- to medium-sized projects. MySQL attracts developers because the setup of the software is really easy and it has good community support. Quite a number of famous sites and platforms have chosen MySQL as their default database, such as WordPress, Joomla, and Drupal. MySQL runs different storage engines along with replication or clustering mechanisms to facilitate high availability and horizontal scaling of growing applications [48].

Figure 2.9: MySQL Logo

Microsoft SQL Server

Microsoft SQL Server is a relational database management system for an enterprise environment. It works very well with other Microsoft products such as Azure, Power BI, and Excel and hence forms a very important part in the organization using the Microsoft ecosystem. SQL Server can give advanced analytics, data integration, and reporting for business intelligence applications. High availability and data security will be ensured for the critical systems by features such as Always On Availability Groups and encryption [49].

Figure 2.10: Microsoft SQL Server Logo

PostgreSQL

PostgreSQL is an open-source, free-of-cost, relational database system that does support many strong features with flexibility. It supports complicated queries, indexing, and transactions compliant with ACID principles: Atomicity, Consistency, Isolation, Durability. The strongest feature of PostgreSQL is its great extensibility. Here, the developers can define their custom types and functions. It primarily supports modern advanced data structures like JSON and XML and thus becomes fit for serving a variety of modern applications in which structured and semi-structured handling of data is needed. PostgreSQL finds broad application in academic research, startups, and enterprises where reliability and performance are needed [50].

Figure 2.11: PostgreSQL Logo

2.4.4 Non-Relational Databases

The general meaning of a NoSQL, or non-relational, database is to forget the conventional rows and columns method of storing and retrieving data. It normally has data stored in flexible structures such as documents, key-value pairs, wide-column stores, or graph models. Hence, this is perfectly adapted to deal with unstructured, semi-structured, or big volumes of data. A perfect match comes from new, modern applications like real-time analytics, content management, and IoT systems [51].

In a non-relational database, the storage format of the data can be in document form like JSON/BSON, key-value pairs, column families, or graph structures. The databases are either schema-less or schema-flexible, hence allowing evolution of data models with time. Most of the non-relational databases support distributed architecture for horizontal scaling and high availability [51].

2.4.5 Benefits of NoSQL Databases

This has made the NoSQL databases a favorite among developers for modern applications, because of their [52]:

1. **Flexible Data Model**: Their data models can be flexible enough to hold structured, semi-structured, and unstructured data. Because of this, storing a wide range of data types and adjusting the structure in case of requirement changes is very easy without major interruptions of services.
2. **Scalability**: With horizontal scaling in mind, NoSQL databases can store large volumes of growing data and serve many more users. It is a way of scaling horizontally across multiple servers in order to support consistently high performance for applications.
3. **Performance**: There are some NoSQL databases that offer high-speed operations by renormalizing data retrieval and storage mechanisms. That makes them most applicable within applications requiring real-time access and processing of data.
4. **Reliability and High Availability**: The majority of NoSQL databases were designed to offer replication and distribution of data. Because of this, failure in hardware or network doesn't result in the unavailability of the data. That supports the continuous availability and reliability of such architecture.
5. **Ease of Development**: Complying with agile development principles, changes in the application requirements can be accommodated very fast since the NoSQL databases have their schemas and data models flexible.

Thus, a number of reasons, such as flexibility, scalability, and performance, make NoSQL databases appropriate for applications that require these features.

2.4.6 Examples of NoSQL Databases

MongoDB

MongoDB is an open-source document-oriented database intended for modern web applications, for which high scalability and flexibility are required. It does not store data in tables, as in other databases, but it uses collections and documents, with each document being a JSON-like object. Such a schemaless design of MongoDB allows for highly flexible storage and querying of data; changes are accommodated without modification in the database schema. It may be appropriate for applications involving large amounts of data-for example, content management systems, IoT platforms, and real-time analytics. It also supports horizontal scaling, replication, and native sharding for distributed systems [53].

Figure 2.12: MongoDB Logo

Cassandra

Apache Cassandra is designed to handle huge volumes of data across many low-cost commodity servers with no single point of failure. This storage uses a wide-column model for fast data retrieval and efficient storage of huge amounts of data. Highly scalable, Cassandra provides linear scaling and fault tolerance; hence, it finds a perfect application in demanding availability requirements. Netflix, Instagram, and Spotify run on Cassandra for logging, real-time analytics, and huge volume user data [54].

Figure 2.13: Apache Cassandra Logo

Redis

Redis is an in-memory data structure store that can be used as a database, used as a cache, and as a message broker. The support of various data structures in Redis includes strings, hashes, lists, sets, and more, really making it versatile for many different use cases. Because Redis operates mostly in memory, read and write speeds are extremely fast, which is great for caching, session storage, and real-time analytics. It also has features like replication and persistence to disk for high availability and durability [55].

Figure 2.14: Redis Logo

Couchbase

Couchbase is a NoSQL database imbued with powers from both a document-oriented database and a key-value store. Optimized for applications requiring high throughput and low latency, highly interactive applications have been their target. Following distributed architecture, Couchbase features in-memory caching, sync capabilities, and hence much better means of scaling up mobile and IoT applications. Typical use cases for such a product range from e-commerce and gaming to the healthcare sector because these have highly performant services concerning user experience [56].

Figure 2.15: Couchbase Logo

2.4.7 Graph Databases

A graph database is a unique class of NoSQL database in which graph structures that include nodes, edges, and properties are applied for data representation and storage. The architectural model has been seen to have greater effectiveness in handling complex and interconnected data through the explicit modeling of relations, thus making it highly applicable in social networks, recommendation systems, fraud detection, supply chain management, etc. Compared to the traditional relational databases based on tables and joins, graph databases provide a more natural and effective way of querying relations [57].

2.4.8 Benefits of Graph Databases

These are some of the advantages of using graph databases in today's applications [57]:

1. **Native Relationship Management**: Unlike relational databases, which rely on JOIN operations to link entities in their schema, graph databases treat relationships as objects. These are stored natively as a data element which allows for optimised data flow.
2. **Flexible Schema Design**: Graph databases allow data structures to evolve over time without disrupting existing data. New nodes, relationships or properties can be added even after the data has been entered, without the need to rewrite existing data. Graph databases support agile or iterative development processes.
3. **High Performance on Connected Queries**: Graph databases are optimised for queries and for querying any existing data structure under a reasonable connection between the data. In a way that most other traditional relational databases can't do: traversals across connections that can be performed with low-latency performance.
4. **Intuitive Data Modeling**: The graph data model closely resembles real-world networks and relationships. This often makes it easier for programmers, analysts and stakeholders who are closer to the real scenario to visualise the data structure and design data models.
5. **Improved analysis**: Graph databases support advanced analysis, especially with complex entities, which can include path finding, centrality detection or matching patterns of relationships and circumstances that promote a better understanding of activities in various disciplines, such as supply chain optimisation, knowledge graphs and network analysis.

2.4.9 Examples of Graph-based Databases

AWS Neptune

AWS Neptune is a graph database offered by Amazon Web Services. It supports two models, namely, the property graph model with Gremlin, and the RDF model with SPARQL. It assists numerous applications such as knowledge graphs, identity graphs, and network security. It is designed for high availability and durability with options of replication, automatic backup, and integration with other AWS services. It is appropriate for businesses that have graph workloads in the cloud [58].

Figure 2.16: AWS Neptune Logo

Neo4J

Neo4j is the best graph database. It is a robust platform that is transactional and ACID compliant, and this is why it is suitable for graph data management. It supports a query language, Cypher, that enables developers to code queries in a manner that is consistent with how humans perceive relationships. It is mostly applied in applications such as fraudulent detection, recommendation systems, and managing critical data due to its excellent tools, vibrant community, and proper support for visualization [59].

Figure 2.17: Neo4j Logo

ArangoDB

ArangoDB is a multi-model database that seamlessly combines graph, document, and key-value models in one engine. Its flexible architecture allows users to model complex relationships together with unstructured data with precision. Using its own Arango Query Language (AQL), ArangoDB allows for high-level queries between different models without ever requiring a technology switch. Therefore, it is especially suited for applications requiring both hierarchies and relationships, for instance, content management systems or logistics systems [60].

Figure 2.18: ArangoDB Logo

RedisGraph

RedisGraph is a graph database module intricately constructed on top of Redis's in-memory data store. Adopting the Cypher query language, RedisGraph is carefully designed for ultra-low-latency access, providing full support for real-time analytics. Through the use of the power of sparse matrices and linear algebra, it provides high-velocity graph processing, positioning itself as the perfect partner for high-throughput applications like recommendation engines, session graphs, or social media timelines. Its inherent simplicity, paired with incredible acceleration, makes it an outstanding choice for applications that require immediate responses [61].

Figure 2.19: RedisGraph Logo

2.4.10 Performance Comparison

As previously said, relational databases, like PostgreSQL, have full optimization for structured data with much more complicated query operations. They do well where consistent relational and transactional integrity is necessary. For example, PostgreSQL can do better with things like complex SELECT queries or even DELETEs, since it relies on indexed relational schemas. NoSQL databases are schema-less, therefore, MongoDB can be fast when it comes to insert operations. Meanwhile, graph databases like Neo4j are optimized for relationship-centric data. They outperform traditional databases in traversing multiple levels of relationships or performing graph-like queries, thanks to native graph storage and constant-time relationship lookups. This can be seen in the following performance metrics [62, 63]:

Table 2.1: Performance Comparison

Operation	**PostgreSQL (ms)**	**MongoDB (ms)**	**Neo4J (ms)**
Insert (10k records)	800	400	900 (est.)
Select (simple)	5	8	7-13
Select (complex)	20	25	9-15
Recursive Query	200–300	>300	2–3
Aggregation	22–118	20–90	15–86
Pattern Matching	49	>50	1–5

As is evident in Table 2.1, MongoDB performs well in raw insertion, with the insertion of 10,000 records in roughly half the time that PostgreSQL requires. PostgreSQL, on the other hand, performs better on executing structured SELECT and DELETE queries. In comparison, Neo4j delivers significant performance advantages in recursive queries—displaying up to 146 times increased speeds—and in pattern-matching queries, with up to 10-fold enhancement compared to relational models. Such queries normally involve several joins in SQL, but Neo4j accomplishes these requirements efficiently through its graph traversal mechanism and the Cypher query language.

The integration of SQL or NoSQL databases with Kafka results in acceptable latency. However, when the amount of data rises, the latency of synchronization increases linearly with it. This affects performance when dealing with large throughput. For queries involving several joins or nested conditions, PostgreSQL outperforms MongoDB. At large volumes of data, PostgreSQL always returns stable times for the queries, while MongoDB has a slight performance degradation at such a load, justifying the database orientation in handling semi-structured or unstructured data [64, 65].

MongoDB support horizontal scaling where distributed systems support load increases more efficiently and at much cheaper costs compared to PostgreSQL that support vertical scaling and hence depend on hardware scaling to boost performance. The somewhat expensive hardware scaling thus makes SQL databases less practical for real-time systems that deal in enormous datasets. Due to these many reasons, NoSQL is ideal for real-time analytics at scale [64].

The schema-less nature of MongoDB lets it adapt to rapidly changing data structures-a feature very important for real-time systems operating on dynamic data. This is opposed to PostgreSQL, which requires a predefined schema and thus may not be able to keep pace when data models change frequently [64].

Neo4j is really good for real-time work, especially when dealing with data that's very connected. Its design is built for graphs, so it can quickly move through many levels of relationships. This lets it run deep and complex searches faster than other database types. Studies show that Neo4j can be more than 100 times faster at these searches than traditional databases. This makes it great for finding fraud, suggesting things to users, and managing large amounts of information where connections matter a lot [63].

In conclusion, PostgreSQL is most suitable for structured data and transactional operations. MongoDB is ideal for dynamic, high-ingestion environments with unstructured data. Neo4j emerges as the top performer in relationship-centric data, especially for complex traversals, hierarchical dependencies, and pattern discovery tasks.

2.5 Parallel Computing and Concurrency Techniques

2.5.1 Parellel Computing

Parallel computing is a type of computing wherein the big computational problems are divided into smaller tasks, which are executed simultaneously by many processors. It is contrasted with serial computing, in which a program is executed one line at a time by a single processor, usually resulting in longer execution times [66].

All processors in parallel computing communicate via shared memory. Then, the various solutions from each processor are integrated by using particular algorithms. It assures greater speed and efficiency in computation and, hence, is able to tackle complex problems much better. This has played a great role in bringing about the realization of recent technologies such as supercomputers, AI, and IoT [66].

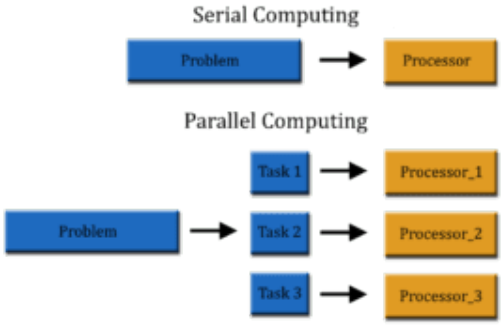

Figure 2.20: Serial vs Parallel Computing

The architecture of parallel computing systems may be different. Generally, there are three types of parallel computing systems [66, 67]:

- **Shared Memory Systems**: One common memory is shared between several processors. This, while offering very effective communication among processors, calls for close monitoring in order to avoid memory conflicts.
- **Distributed Memory Systems**: Every processor has its private memory. Processors communicate through the passing of messages. This architecture will be easy to scale up but may suffer because of communication loss.
- **Hybrid Systems**: These incorporate elements of both shared and distributed memory architectures so as to make use of the respective advantages of both.

Generally, the whole point of parallel computing is to solve big computations more quickly and efficiently than traditional serial computing. In that respect, parallel computing becomes an essential factor when substantial computational powers are required, as is the case with scientific simulations, data analysis, real-time processing, AI and machine learning [66].

2.5.2 Concurrency

In computing, concurrency or concurrent computing is the style of computing wherein many computing tasks occur together or at overlapping time intervals. Such processing can be done, for instance, by independent computers given applications or within networks. Concurrent computing in big data environments is an approach massively used to handle very

huge data sets. This demands a well-coordinated effort between systems and across the architectures of big data concerning task scheduling, data exchange, and memory allocation so that things work efficiently and effectively [68].

2.5.3 Key Differences

Concurrency deals with several tasks that might overlap in their starting, execution, and finishing times. It deals more with the structure of a program that can support several tasks amply, not necessarily running simultaneously. On a single processor, concurrency is usually emulated by time-slicing among several tasks, giving an impression of contemporaneous execution of tasks [69].

On the other hand, parallelism involves the real execution of more than one task at the same instant of time. It utilizes many processors or cores and is mainly used to enhance computation speed through the splitting of tasks into independent subtasks that can contemporarily run [69].

Here are the key differences [69]:

- **Simultaneity**: Concurrency deals with the handling of multiple tasks within overlapping time slices, where for parallelism, more than one task has to be executed in actual time simultaneously.
- **Resource Utilization**: Concurrency can be achieved even on a single processor through time-slicing. Whereas parallelism requires many processors or cores.
- **Design vs. Execution**: Concurrency is a design issue that deals with structuring a program such that it has multiple tasks. Wherein, parallelism is an execution technique involving doing many computations in parallel.

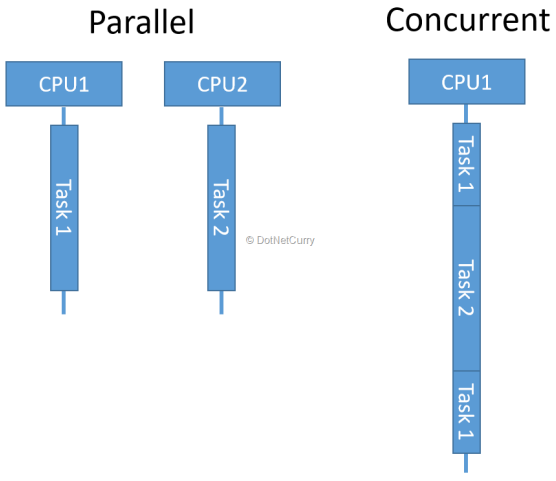

Figure 2.21: Parallel vs Concurrent Computing (retrieved from [70])

2.5.4 Parallel computing in quality control context

Quality control systems gathers data from sensors, cameras, or other IoT devices that monitor the production process. This will allow for the ability to analyze such data in parallel computing, allowing for real-time detection of defects or irregularities, hence aiding in the prevention of defective products moving further into production.

Parallel computing frameworks will be able to process the data coming from various stages of production, detect whatever kinds of deviations from major quality parameters like temperature, pressure, material composition, or amounts, and act immediately to maintain quality and reduce waste.

Those frameworks can work on distributed nodes, offering scalability while assuring quality control systems that their efficiency will not falter with volumes of data on the rise.

Chapter 3

Analysis and Design of the Solution

This chapter presents the design and the evaluation of a solution to the problems of the real-time checking of quality for the large-scale complex production situations. The system suggests the usage of the most recent data processing methods and the graph technologies so that the fast and efficient comparison of the actual and the targeted product properties becomes possible. The argumentation proceeds from the establishment of the functional and the non-functional requirements and proceeds further the consideration on the technology selection and on the architecture of the system. All the above make possible the highly scalable and performance-oriented solution allowing for the decisions in the real-time and the waste reduction in all the industry procedures.

3.1 Requirement Analysis

Requirements analysis establishes the required capability and constraint toward the goal of an efficient and scalable solution to the contemporary production line real-time quality inspection. What the system needs to do (its functional requirements), to what extent it needs to do it (its non-functional requirements), and encapsulating all the system expectation in the FURPS+ model complete the requirements of the design and technology selection enumerated thereafter.

FURPS+ is a categorisation model used for software requirements classification during the five main quality areas of Functionality, Usability, Reliability, Performance, and Supportability. The "+" signifies the other problems of design limitations, implementation needs, interface needs, and physical issues. The model offers a balanced signal on the things the system must do and the manner the system must function on differing eventualities [71].

3.1.1 Use Cases

This section presents key use cases that describe the main system interactions for real-time quality verification and data handling, and you can see the diagram in the Figure 3.1.

- **UC1**: Ingest Real-Time Production Data.
- **UC2**: Compare Product Against Expected Specification.
- **UC3**: Alert on Quality Deviation.
- **UC4**: Manage Expected Product Tree.

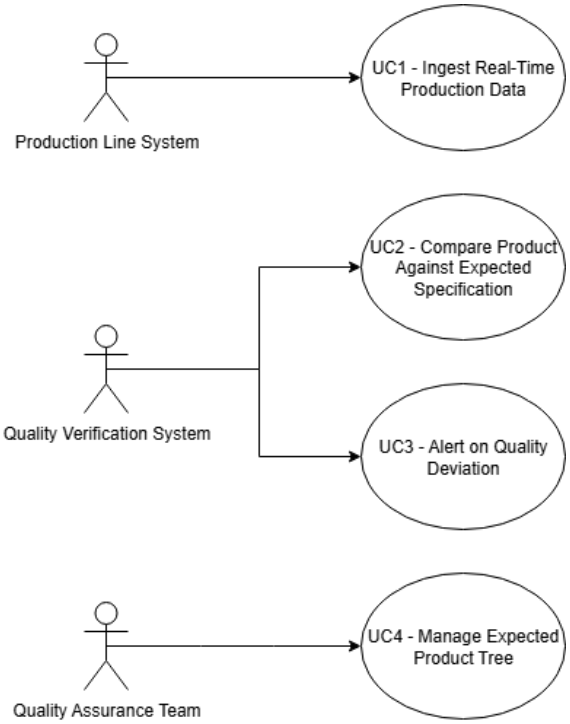

Figure 3.1: Use Case Diagram

The Use Case 1 (UC1) ensures that production line data is continuously ingested and processed, which enables live monitoring. The UC2 makes the system functionality compare the production product properties with the specified standards so the difference gets calculated. The UC3 ensures all the identified non-conformities receive alerts on the immediate basis to the quality team so the correction gets possible. The UC4 usage scenario ensures operators get to define or change the expected product hierarchy manually so the same becomes the master for the quality check.

3.1.2 Functional Requirements

Functional requirements detail the internal behaviours the system must accomplish to support real-time quality validation. The requirements originate from the main use cases and define the behaviours anticipated and the user and system component interactions. The next table defines the main functional capabilities that must exist in order to support the efficient manipulation of data, product validation, and quality validation.

Table 3.1: Functional Requirements

ID	Functional Requirement
FR-01	The system shall ingest real-time product data from the production line using Kafka.
FR-02	The system shall process and store incoming product data for analysis.
FR-03	The system shall compare actual product characteristics with expected values in real time.
FR-04	The system shall detect and identify deviations from expected product specifications.
FR-05	The system shall allow users to manually create, retrieve and delete expected product trees.
FR-06	The system shall store expected product configurations for use during quality checks.
FR-07	The system shall trigger alerts to remove the product when non-conformities are detected.

3.1.3 Non-Functional Requirements

Non-functional requirements specify the behavior of the system and not the goal it is to accomplish. The following comprise the performance, scalability, availability, security, and usability requirements that shall make the system reliable, efficient, and maintainable in all instances. The following table provides an overview of the most important non-functional requirements that shall be responsible for the solution being proposed.

Table 3.2: Non-Functional Requirements

ID	Attribute	Non-Functional Requirement
NFR-01	Performance	The system shall process real-time data with minimal latency to avoid production delays.
NFR-02	Performance	The system shall support horizontal scalability to handle increased data throughput and additional production lines.
NFR-03	Reliability	The system shall ensure high availability to maintain continuous operation during production.
NFR-04	Reliability	The system shall ensure data consistency and integrity across streaming and stored data.
NFR-05	Usability	The system shall provide an intuitive user interface for managing expected product trees.
NFR-06	Supportability	The system shall allow easy maintenance and integration of future components or services.

3.2 Solution Design

This section presents the technologies selected for implementing the proposed solution and details the system architecture following a three-level (three-tier) design: Presentation, Application, and Data layers.

3.2.1 Technology Selection

Technologies chosen for the project are most directly influenced by the particular technical needs of the solution and the state-of-the-art outcome. Objectives included simplifying support for compatibility with a real-time processing environment, scalability, maintainability, and developer productivity. The technologies chosen represent state-of-the-art production-quality tools well suited to the deployment of a three-tier architecture.

- **Frontend – Angular:** The user interface is going to be developed with Angular due to the strength of its component-oriented design, adequate reactive-programming support, and efficient dynamic updating of master and detail information. Its being a mature ecosystem and natively having form handling and state management support makes it suitable for intensive UI interaction handling [72].
- **Backend – Java with Quarkus:** Quarkus, the next-gen Java framework built for optimal performance on Kubernetes and cloud-native application runtimes, is going to be utilised for the construction of the backend services. Fast boot time and minimal usage of the RAM for Quarkus align well within the scenario of the application being in real-time. The choice also aligns with prior experience on the Java ecosystem and makes the building process simple [73].
- **Event Streaming – Apache Kafka:** Apache Kafka was the platform of choice for event streaming as it was capable of handling real-time and high-throughput data pipelines. This helps facilitate the scalable and reliable ingestion of the production line events, before the continuous monitoring and analysis.
- **Database - Neo4j:** In the scenario of modeling and querying the anticipated product tree structure, I've opted for Neo4j due to its inherent support for graph processing and fast traversal algorithms.
- **Communication:** Communication between the Quarkus backend and the Angular frontend is achieved using the RESTful APIs so that the separation of concerns remains clean and the application remains maintainable.

3.2.2 System Architecture

The system it's structured into a three-layer modular structure with Presentation Layer, Backend Layer, and Data Layer. This type of structure allows for a specific separation of responsibilities with an increase in scalability, maintainability, as well as real-time data processing. The architectural structure is specifically aimed at enabling prompt quality evaluation in large production lines. The system consists of separate modules with each module undertaking specific tasks:

- **Presentation Layer**: This layer, built using Angular, acts as an interface for user interaction. It integrates several functionalities such as entering manually the expected product trees, graphical representation of the production line's condition, and real-time notifications.
- **Backend Layer**: The layer, built using Quarkus, holds the system's business logic and controls the data flows. It includes:
  - **Data Consumer Module:** Interacts with live production events coming from Apache Kafka Streams, systematically examining and verifying the data before it proceeds to the verification stage.
  - **Quality Verification Module:** This module checks the production data obtained against the expected product specifications stored in Neo4j, allowing real-time detection of deviations.
  - **API Gateway Module:** controls RESTful endpoints that allow interaction with the frontend and controls all expected operations related to the product tree, including creation, retrieval, and deletion operations in Neo4j.
- **Data Layer**: This layer is responsible for persisting all relevant system data using Neo4j graph database. As such:
  - Expected product tree structures for reference during verification.
  - Real-time and historical product measurements linked to the corresponding nodes.

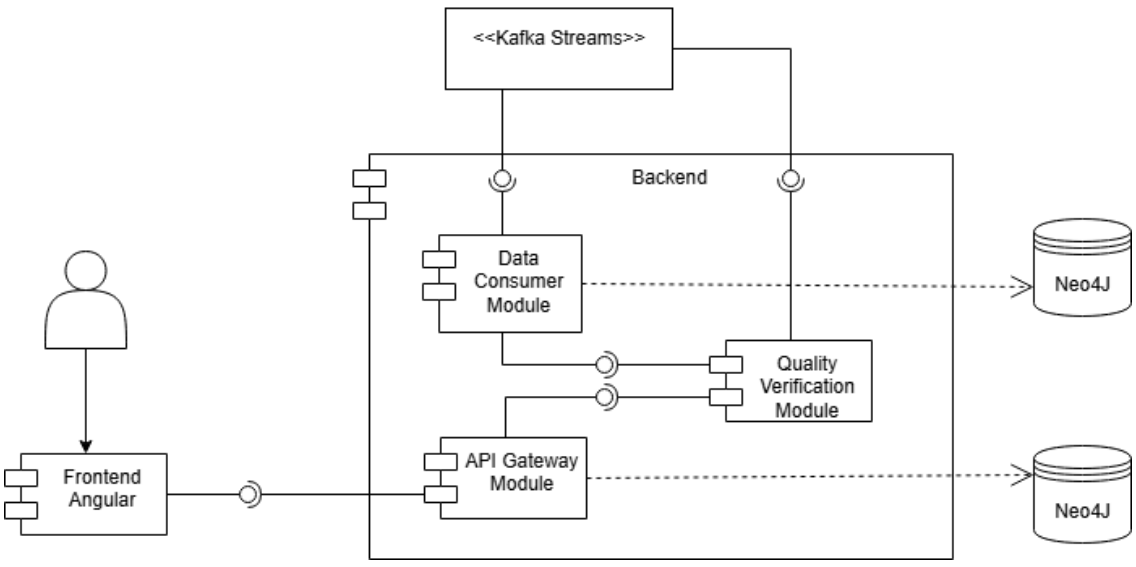

Figure 3.2: System Architecture Diagram

3.2.3 System Process View

System process view describes the system's temporal behaviour by defining the relationships between components within a specified scale of time to enable attainment of specified goals. Analysis centers on two main processes being portrayed, owing to their major contribution towards enabling solution findings:

- **Expected Product Tree Creation Process**: Describes the steps involved when a user creates or updates the expected product tree through the frontend, and how this information is stored in the Neo4j database via the API Gateway Module.
- **Quality Verification Process**: Describes the real-time flow of production data from the ingestion of events via Kafka, through the backend modules for parsing, verification, and deviation detection, until the delivery of alerts to the frontend.

Expected Product Tree Creation Process

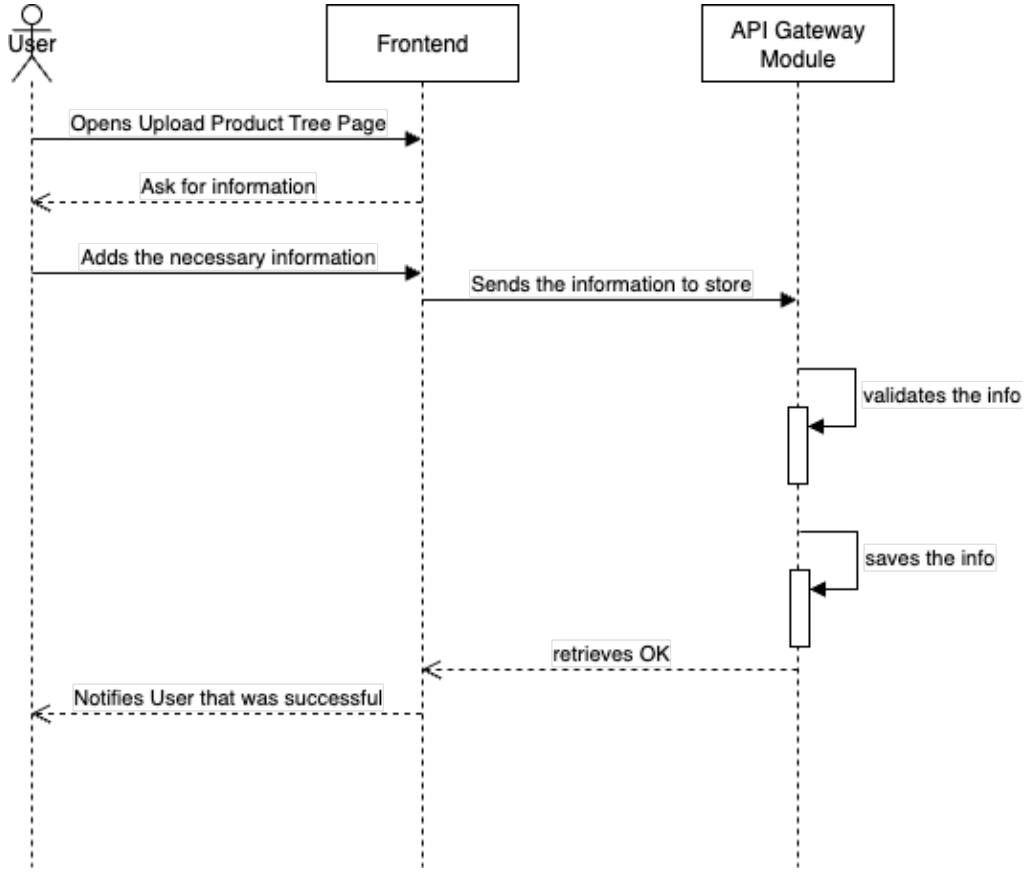

Figure 3.3: Expected Product Tree Creation Process Diagram

Quality Verification Process

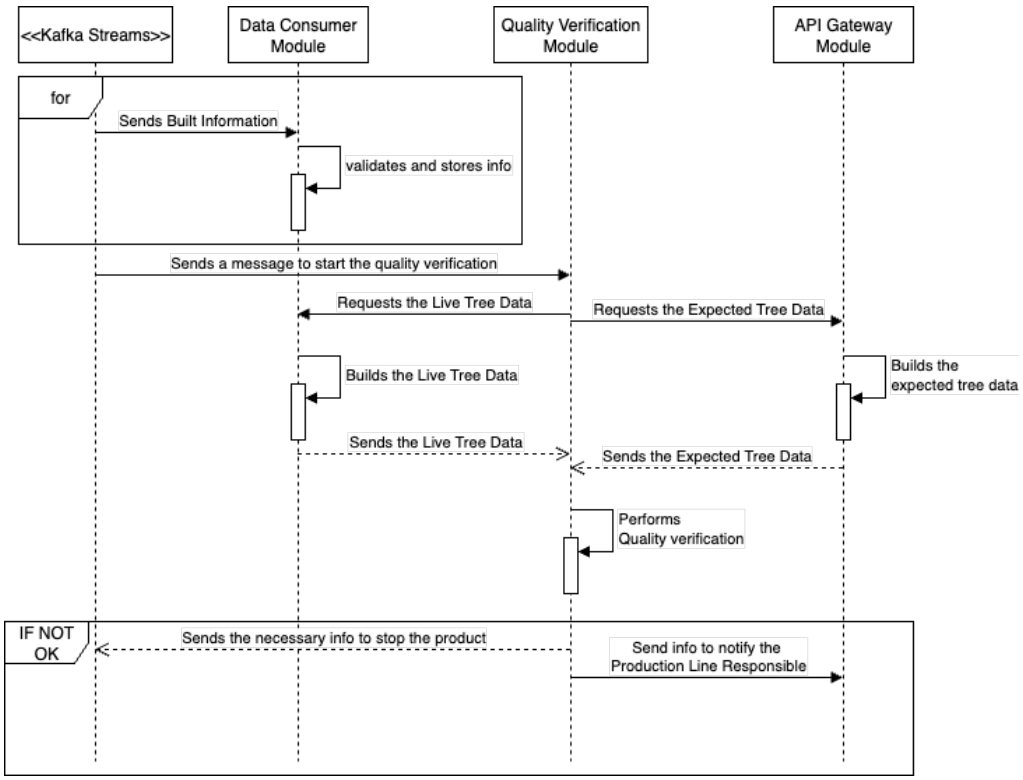

Figure 3.4: Quality Verification Process Diagram

Chapter 4

Solution Implementation

This proposed solution was implemented using a method that transformed the design requirements into an operational system. The development phase entailed the building of key modules, the assembling of auxiliary ones, and the enabling of communication between those modules. Testing and validation were conducted through integration testing and performance analysis to assess the accuracy, reliability, and robustness of the system under various conditions.

4.1 Hardware Integration

To confirm that the implemented solution is deployable in a real-world industrial setup, it is crucial to consider not only the backend services but also the hardware on the production line.

Therefore, the architecture includes sensors (e.g., RFID readers, barcode scanners, and vision systems), programmable logic controllers (PLCs) that collect and manage machine signals, and Internet of Things (IoT) gates or industrial computers (IPCs) that are responsible for translating shop-floor protocols (e.g., OPC-UA and Modbus) to stream events.

These add data to the Kafka messaging system, which connects the operational level to the Quarkus services and the Neo4j data store found in the software ecosystem. The Data Consumer Module (DCM) will consume the production events (assembly or creation) of products within the production line and feed the Neo4j database with the relevant data. The Quality Verification Module (QVM) consumes quality verification events, selects the expected values from the API Gateway Module, and retrieves the live product trees from the DCM. It sends block commands when discrepancies are discovered.

The comprehensive integration of production hardware with backend services is exemplified in Figure 4.1, which illustrates the information flow from the shop floor to the software systems.

This schematic emphasises the manner in which hardware and software components collaborate to constitute a cyber-physical system, facilitating real-time identification of discrepancies and prompt reactions to quality-related concerns.

4.2 Software Implementation

Implementation of the project proceeded on a step-by-step basis, starting with the core backend services. As a first step, the Data Consumer module was built to enable connectivity

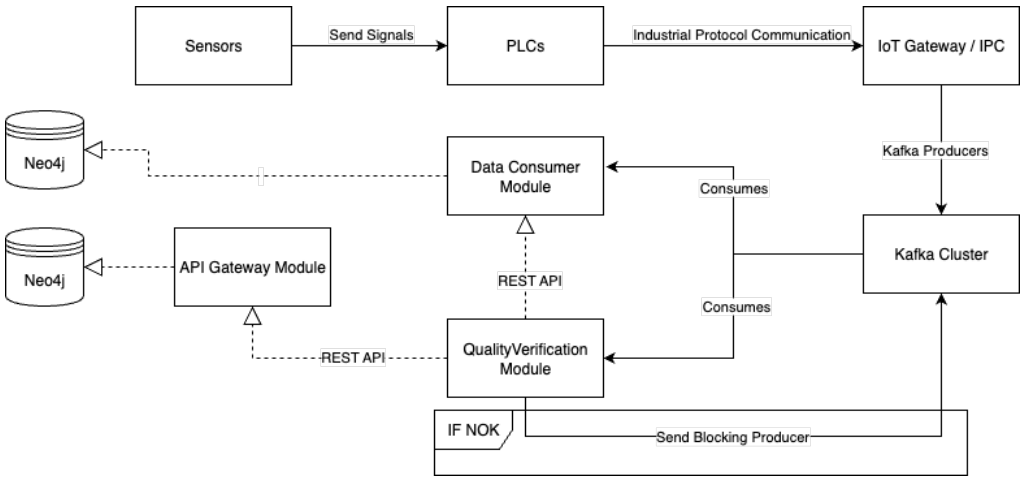

Figure 4.1: Hardware Integration Diagram

with Apache Kafka and thus ensure the reliable ingestion and deserialization of production events in real-time. Next, the module for Quality Verification was implemented with Quarkus for carrying out queries on the Neo4j graph database and for implementing a comparison between actual product data and expected details. Thereafter, API Gateway came into existence as a REST service with endpoints for managing expected product trees and for facilitating communication with the Angular frontend. On the client side, the frontend was developed with reactive forms for supporting manual tree creation with the help of real-time dashboards to display alerts and deviations on the fly. After module development was completed, integration focused on achieving smooth communication between multiple layers. Kafka messages proceeded efficiently via the backend and got processed via the verification logic and stored or compared as part of Neo4j, and Angular interface accessed REST APIs supporting configuration management and displaying live status of notifications. Integration test validated end-to-end data flows to ensure that the system functioned as an end-to-end solution for quality verification in real-time.

4.2.1 Data Consumer Module

As the very first backend module implemented, the Data Consumer module forms the entry point of all production line data entering the system. Its primary role revolves around building an association with the Apache Kafka cluster, subscribing to the interested topics, and de-serialising incoming production events for further processing. Each message acquired comes with a series of product measurements continuously being streamed from the production line. The implementation of the execution was done with Quarkus, utilising its `SmallRye Reactive Messaging` extension for declarative integration with Kafka. This approach ensured minimal boilerplate code and allowed the system to process high-throughput event streams with lower latency. Incoming messages were automatically deserialised as domain objects that encapsulated product data and forwarded to the Quality Verification module for further verification.

This is the implemented consumer listed in 4.1. In our case, the function annotated with `@Incoming` listens for the `parts` topic, stores the event received, and passes it on for verification and insertion.

```
1 @ApplicationScoped
2 public class PartsKafkaConsumer {
```

```
3     private static final Logger LOG = Logger.getLogger(
      PartsKafkaConsumer.class);
4     private static final String HEADER_TYPE = "type";
5
6     @Inject
7     PartServiceInterface partService;
8
9     @Incoming("parts")
10    @Acknowledgment(Acknowledgment.Strategy.MANUAL)
11    @Blocking(ordered = false)
12    public CompletionStage<Void> consume(Message<String> record){
13        try {
14            var md = record.getMetadata(IncomingKafkaRecordMetadata.
      class).orElseThrow();
15            String type = getHeader(md.getHeaders());
16
17            if (type == null || type.isBlank()) {
18                LOG.error("Missing required kafka header 'type'");
19                throw new IllegalArgumentException("Missing required
      kafka header 'type'");
20            }
21            partService.process(type, record.getPayload());
22
23            return record.ack();
24        } catch (Exception e) {
25            return record.nack(e);
26        }
27    }
```

Listing 4.1: Kafka Consumer Implementation in Quarkus

To facilitate simultaneous message processing and enhance overall throughput, supplementary Kafka client configurations were established. Specifically, there was an increase in the number of consumer instances, as well as an increase in the upper limit of messages retrieved per request. The implemented configuration is presented in 4.2.

```
1 # Enabling concurrency on Kafka
2 mp.messaging.incoming.parts.max.poll.records=500
3 mp.messaging.incoming.parts.fetch.min.bytes=1048576
4 mp.messaging.incoming.parts.fetch.max.wait.ms=200
5
6 # Enable currency on Neo4j
7 quarkus.neo4j.pool.max-connection-pool-size=100
8 quarkus.neo4j.pool.connection-acquisition-timeout=30S
```

Listing 4.2: Kafka Consumer Configuration for Concurrency

Kafka Message Types

The messages that are being sent on `parts` topic are JSON payloads keyed by the logical entity and tagged via a Kafka header (`"type", b"..."`), and two types are produced:

Creation

This type of message represents a part or assembly vertices of the product tree. The Kafka key is the `partCode`, and only the root node includes `patternKey`. The `pointOfVerification` (PoV) is computed per tree as `<Location>-POV-<ZoneX>`.

```
{
  "partCode": "ASSY-ENG-0007-ROOT",
  "id": "ASSY-ENG-0007-ROOT",
  "name": "Top Assembly",
  "description": null,
  "category": null,
  "type": null,
  "revision": null,
  "lifecycle": null,
  "isAssembly": true,
  "isTopLevel": true,
  "active": true,
  "uom": "EA",
  "stdCost": null,
  "currency": null,
  "weight": null,
  "weightUnit": null,
  "length": null,
  "width": null,
  "height": null,
  "dimensionUnit": null,
  "defaultSupplier": null,
  "effectiveFrom": null,
  "effectiveTo": null,
  "amount": null,
  "position": null,
  "reference": null,
  "variantCode": null,
  "optional": null,
  "scrapPct": null,
  "pointOfVerification": "FactoryA-POV-Zone3",
  "patternKey": "ASSY-ENG-0007-ROOT:FactoryA-POV-Zone3"
}
```

Listing 4.3: Creation message (part/assembly node)

Assembly

This type of message represents parent->child relationships of the product tree. The Kafka key is the `parentId`. Relationship properties include `qty`, `position`, and a `reference` string.

```
{
    "parentId": "ASSY-ENG-0007-ROOT",
    "childId": "PART-ENG-0007-L1N0001",
    "relType": "CONTAINS",
    "properties": {
        "qty": 2,
        "position": 1,
        "reference": "ASSY-ENG-0007-ROOT->PART-ENG-0007-L1N0001"
        },
    "pointOfVerification": "FactoryA-POV-Zone3"
}
```

Listing 4.4: Assembly message (edge/relationship)

Headers summary

Message	**Header** `type`	**Kafka key**
Creation (node)	`"creation"`	`partCode`
Assembly (edge)	`"assembly"`	`parentId`

Product Tree Resource

This REST resource presents a read endpoint for accessing an entire product tree from its *pattern key*. For ease of scalability on handling of concurrent requests, the method itself is marked `@RunOnVirtualThread` such that each request runs on a lightweight virtual thread and so improves concurrency without consuming the platform threads on blocking I/O operations.

```
    @GET
    @Path("/{patternKey}")
    @RunOnVirtualThread // <-- parallel requests to scale
    public Response byPatternKey(@PathParam("patternKey") String
    patternKey) {
        return service.getTreeByPatternKey(patternKey)
                .map(Response::ok)
                .orElseGet(() -> Response.status(Response.Status.
    NOT_FOUND)
                .entity(new ErrorMessage("Tree not found for patternKey=
    " + patternKey)))
                .type(MediaType.APPLICATION_JSON)
                .build();
    }
```

Listing 4.5: Product tree REST resource using virtual threads

This resource can be accessed through a `GET` request, which returns the serialised product tree for the given key. If the tree exists, the service returns an HTTP 200 message with a JSON body. In those cases wherein the referenced key doesn't match any existing tree, the endpoint will return an HTTP 404 message with a minimal error JSON.

4.2.2 Quality Verification Module

The Quality Verification Module (QVM) is a critical component of the backend that is responsible for ensuring that the *live product tree* generated on the production line aligns with the *expected product tree*. The main purpose of the module is to confirm that every assembly is according to the design specification to facilitate quick identification and action on any deviations.

The QVM sets up a bidirectional link with Apache Kafka. On one side, it consumes commands issued on the `quality-start` topic, which get issued whenever some quality verification procedure has to be invoked. Every command comes with some metadata like the `patternKey`, the source of the message, and a correlation ID. The consumer annotated with `@Incoming` is configured to listen to `quality-start` messages and then call the verification service.

```
1     @Incoming("quality-start")
2     @Acknowledgment(Acknowledgment.Strategy.MANUAL)
3     @Blocking
4     public CompletionStage<Void> onStart(Message<String> msg) {
5         StartCommand cmd = new StartCommand();
6         try {
7             cmd = mapper.readValue(msg.getPayload(), StartCommand.class)
    ;
8             if (cmd.patternKey == null || cmd.patternKey.isBlank()) {
9                 throw new IllegalArgumentException("Missing patternKey
    in StartCommand");
10            }
11
12            var expected = expectedTreeService.fetch(cmd.patternKey);
13            var live = liveTreeService.fetch(cmd.patternKey);
14            CompareResultDTO result = qualityVerificationService.compare
    (cmd.patternKey, expected, live);
15
16            if (!result.ok) {
17                BlockEvent event = new BlockEvent();
18                // set event properties
19                blockProducer.send(event, event.blockedPartCodes.
    getFirst(), event.severity);
20            }
21            return msg.ack();
22        } catch (Exception e) {
23            LOG.info("Tree with Pattern Key" + cmd.patternKey + " and
    Part Code " + cmd.patternKey.split("_")[1] + " was blocked");
24            BlockEvent event = new BlockEvent();
25            // set event properties
26            blockProducer.send(event, event.blockedPartCodes.getFirst(),
     event.severity);
27
28            return msg.nack(e);
29        }
30    }
```

Listing 4.6: Quality Start Consumer

From the opposite side, once the verification has been finished, the QVM can write events on the `quality-block` topic if some inconsistencies have been discovered, thus avoiding defective assembly pieces from moving further down the production chain.

```
    @Channel("quality-block")
    Emitter<String> emitter;

    public void send(BlockEvent event, String key, String severity) {
        try {
            var headers = new RecordHeaders()
                    .add("type", "quality-block".getBytes(
    StandardCharsets.UTF_8))
                    .add("severity", severity.getBytes(StandardCharsets.
    UTF_8));

            var metadata = OutgoingKafkaRecordMetadata.<String>builder()
                    .withKey(key)
                    .withHeaders(headers)
                    .build();

            String json = mapper.writeValueAsString(event);
            emitter.send(Message.of(json).addMetadata(metadata));
        } catch (Exception e) {
            throw new RuntimeException("Failed to send block event", e);
        }
    }
```

Listing 4.7: Block Part Producer

Central to the module is the Quality Verification service, shown in Listing 4.8. This service contains the business logic inherent in tree comparison. It is given both the expected and actual structure and will perform a recursive walk of its nodes and the relationships between them.

During the course of the walk, it checks attributes like *part codes*, *amounts*, *positions*, and various assembly metadata, and also checks that all the child components present in the expected tree exist within the actual tree. The result of this process is a `CompareResultDTO`, which gathers information on the success of verification, along with, if it fails, a listing of the discrepancies that were discovered, the severity of the discrepancy, and which parts were affected.

This modular structure enables the verification logic to be independently evolved in relation to the infrastructure of the messages, thereby opening the possibility of adding more advanced validation rules or heuristics in the future.

```
    public CompareResultDTO compare(String patternKey, TreeDTO expected,
    TreeDTO live) {
        CompareResultDTO r = new CompareResultDTO();
        if (expected == null || live == null) {
            r.ok = false ;
            return r;
        }

        var exp = flatten(expected.root);
        var liv = flatten(live.root);

        List<String> problems = new ArrayList<>();

        // Nodes
        for (String n : diff(exp.props.keySet(), liv.props.keySet())) {
            problems.add("Missing in live: " + n);
        }
        for (String n : diff(liv.props.keySet(), exp.props.keySet())) {
            problems.add("Unexpected in live: " + n);
        }

        // Edges
        var parents = union(exp.edges.keySet(), liv.edges.keySet());
        for (String p : parents) {
            var eKids = exp.edges.getOrDefault(p, Set.of());
            var lKids = liv.edges.getOrDefault(p, Set.of());
            for (String c : diff(eKids, lKids)) // Add Missing Child
            for (String c : diff(lKids, eKids)) // Add Unexpected Child
        }

        var common = inter(exp.props.keySet(), liv.props.keySet());
        for (String n : common) {
            var eProps = exp.props.get(n);
            var lProps = liv.props.get(n);
            // Unit of Measure
            Object eUom = eProps.get("uom");
            Object lUom = lProps.get("uom");
            if (!Objects.equals(eUom, lUom))
                problems.add("UOM mismatch on " + n);
            // Assembly Type
            Boolean eAsm = toBool(eProps.get("isAssembly"));
            Boolean lAsm = toBool(lProps.get("isAssembly"));
            if (!Objects.equals(eAsm, lAsm))
                problems.add("isAssembly mismatch on " + n);
            // Quantity
            Object eQty = eProps.get("qty");
            Object lQty = lProps.get("qty");
            if (!Objects.equals(eAsm, lAsm))
                problems.add("Quantity mismatch on " + n);

        }
        if (problems.isEmpty()) return CompareResultDTO.ok(patternKey);
        r.ok = false ;
        return r;
    }
```

Listing 4.8: Quality Verification Service

4.2.3 Main API Module

The Main API Module (MAM) is the Gateway not only to the frontend but also to the backend system, and it offers a collection of RESTful services that facilitate interaction of the production knowledge. It differs essentially from the other modules that predominantly work asynchronously via Apache Kafka, since the MAM gives synchronous access to some of the use cases.

The MAM provides effortless handling of HTTP requests with low latency and also allows supporting reactive backpressure semantics where needed. The endpoints, shown in 4.9 are organised around the *TreeNode* model, shown in Listing 4.10. All patterns are represented in Neo4j as a tree of relationships and nodes that has the assembly semantics built in and that guarantees all subsequent checks depend on a single trusted source of truth. The endpoints return formatted JSON outputs and mesh perfectly with the service and repository layers that need to execute the necessary Neo4j queries.

```
@Path("/api/patterns")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public interface PatternResourceInterface {

    // Upload and persist a product pattern tree (Parallel Programming).
    // @param req request containing location, fatherComponentNumber and pattern JSON
    // @return 200 OK if stored, 400 for validation errors
    @POST
    Response uploadPattern(@Valid UploadPatternRequest req);

    // Inactivate a pattern by its key.
    // @param patternKey unique pattern identifier
    // @return 200 OK if inactivated, 404 if not found
    @POST
    @Path("/{patternKey}/inactivate")
    Response inactivate(@PathParam("patternKey") String patternKey);

    // Retrieve the full tree of a pattern (root + all children recursively).
    // Only active patterns are considered.
    // @param patternKey identifier of the pattern
    // @return 200 OK with TreeNode, or 404 if not active/not found
    @GET
    @Path("/{patternKey}")
    Response getTree(@PathParam("patternKey") String patternKey);

    // Retrieve a list of summaries for all patterns.
    // @return 200 OK with an array (possibly empty)
    @GET
    @Path("/")
    Response list();

    // Retrieve a list of summaries for all active patterns.
    // @return 200 OK with an array (possibly empty)
    @GET
    @Path("/active")
    Response listActive();
}
```

Listing 4.9: PatternResource

```java
public class TreeNode {

    @JsonProperty("partCode")
    public String partCode;

    @JsonProperty("name")
    public String name;

    @JsonProperty("description")
    public String description;

    @JsonProperty("category")
    public String category;

    @JsonProperty("type")
    public String type;

    @JsonProperty("revision")
    public String revision;

    @JsonProperty("lifecycle")
    public String lifecycle;

    @JsonProperty("isAssembly")
    public Boolean isAssembly;

    @JsonProperty("isTopLevel")
    public Boolean isTopLevel;

    @JsonProperty("active")
    public Boolean active;

    @JsonProperty("patternKey")
    public String patternKey;

    @JsonProperty("uom")
    public String uom;

    // Other specification of the TreeNode

    @JsonProperty("amount")
    public Double amount;

    @JsonProperty("children")
    public List<TreeNode> children = new ArrayList<>();
}
```

Listing 4.10: TreeNode Model

Parallel Programming for the Pattern Storage

Throughout the process of implementation of the Pattern Repository, it became obvious that the sequential insertion of large hierarchical structures into Neo4j constituted a significant constraint within the system. Each product tree may comprise thousands of nodes and edges, and the commitment of these structures through a single-threaded approach resulted in prolonged transaction durations and suboptimal utilisation of the existing CPU and database resources. In answer to this challenge, the repository has been redesigned to use parallel programming techniques by way of batch inserts.

Worker Threads

To ensure flexibility and scalability, the number of writer threads that can run concurrently and the transaction-batch size can be parameterised. With this ability, the system can be tuned to the particular environment: the machine of a developer may have fewer threads running, while a production server can take advantage of larger concurrency levels. The use of configuration instead of hardcoding facilitates the versatility of the implementation.

```
1     @ConfigProperty(name = "pattern.writer.threads", defaultValue = "4")
2     int writerThreads;
3
4     @ConfigProperty(name = "pattern.writer.batch-size", defaultValue = "
      500")
5     int batchSize;
```

Listing 4.11: Thread Configuration

Parallel batching executor

The element of parallel programming is the parallelBatches method. This divides an input workload into evenly sized chunks and passes them to a thread pool to be executed. Individual batches get independent processing of their own transaction, thus providing atomicity. Futures provide synchronisation and collect errors to the point where the entire operation will wait till the end of successful completion of all the batches.

```
1  private <T> void parallelBatches(List<T> all, Consumer<List<T>> writer)
       {
2      if (all.isEmpty()) return;
3      List<List<T>> chunks = chunk(all, Math.max(1, batchSize));
4      ExecutorService pool = Executors.newFixedThreadPool(Math.max(1,
       writerThreads));
5      try {
6          List<Future<?>> futures = new ArrayList<>(chunks.size());
7          for (List<T> c : chunks) {
8              futures.add(pool.submit(() -> writer.accept(c)));
9          }
10         for (Future<?> f : futures) f.get();
11     } catch (InterruptedException ie) {
12         Thread.currentThread().interrupt();
13         throw new RuntimeException("Interrupted writing batches", ie);
14     } catch (ExecutionException ee) {
15         throw new RuntimeException("Batch failed", ee.getCause());
16     } finally {
17         pool.shutdown();
18     }
19 }
```

Listing 4.12: Parallel Batch

Parallel Batch Division

As said previously, the `parallelBatches` method is in charge of dividing the input workload into smaller portions and running them, in parallel, on a thread pool. The amount of portions C created is a function of the total number of elements N, the batch size configured B, and it follows the following expression:

$$C = \left\lceil \frac{N}{B} \right\rceil$$

For instance, if $N = 10\,000$ items are added in a batch size of $B = 500$, there are produced $C = \lceil 10000/500 \rceil = 20$ chunks. Then, every chunk is dealt with by a group of `writerThreads` threads. This design ensures a balanced distribution of work and allows for parallel processing, thereby scaling well while maintaining deterministic characteristics through idempotent operations in the database layer.

```
private static <T> List<List<T>> chunk(List<T> list, int size) {
    List<List<T>> out = new ArrayList<>();
    for (int i = 0; i < list.size(); i += size) {
        out.add(list.subList(i, Math.min(i + size, list.size())));
    }
    return out;
}
```

Listing 4.13: Parallel Batch

Batch Node Insertion

Nodes are inserted in batches by using the UNWIND construct of Neo4j. It ensures that multiple rows get executed in one query and that around-trip latency is minimised. Using MERGE makes the operation idempotent, i.e., parallel executions will not lead to the creation of multiple nodes, but will instead update those that already exist seamlessly.

```
private void writeNodesBatch(List<NodeRow> rows) {
    if (rows.isEmpty()) return;
    try (Session s = driver.session()) {
        s.executeWrite(tx -> {
            tx.run("""
                UNWIND $rows AS row
                MERGE (x:Part {patternKey: row.pk, partCode: row.pc})
                SET  x += row.props,
                     x.pointOfVerification = coalesce(row.pV, x.
    pointOfVerification),
                     x.patternName         = coalesce(row.pN, x.
    patternName),
                     x.updatedAt = datetime(),
                     x.createdAt = coalesce(x.createdAt, datetime())
                """, Map.of("rows", rows.stream().map(r -> Map.of(
                    "pk", r.pk, "pc", r.pc, "props", r.props, "pV", r.pV
    , "pN", r.pN
                )).toList()));
            return null;
        });
    }
}
```

Listing 4.14: Batch Node Insertion

Batched Relationship Insertions

Following the addition of nodes, relationships between these nodes are also added in concurrent batches. Again, the MERGE operation ensures that concurrent executions remain consistent. Every relation is enriched with the attributes of quantity, reference, and timestamps, simultaneously being added or modified idempotently.

```
private void writeRelsBatch(List<RelRow> rows) {
    if (rows.isEmpty()) return;
    try (Session s = driver.session()) {
        s.executeWrite(tx -> {
            tx.run("""
                UNWIND $rows AS row
                MATCH (p:Part {patternKey: row.pk, partCode: row.pp})
                MATCH (c:Part {patternKey: row.pk, partCode: row.cp})
                MERGE (p)-[rel:HAS_CHILD]->(c)
                SET rel += row.r,
                    rel.updatedAt = datetime(),
                    rel.createdAt = coalesce(rel.createdAt, datetime())
                """, Map.of("rows", rows.stream().map(r -> Map.of(
                    "pk", r.pk, "pp", r.pp, "cp", r.cp, "r", r.r
                )).toList()));
            return null;
        });
    }
}
```

Listing 4.15: Batch Relationship Insertion

High-level Orchestration

The upsertTree method manages the parallel execution workflow by first checking the existence of the constraint. Then it flattens the tree by converting the hierarchical tree structure into a linear form, writes nodes and relationships with parallel programming, and then synchronises children as needed. This separation of duties lowers contention and provides a clear, deterministic workflow.

The flattening process is crucial, as it reduces the hierarchical tree structure to two separate lists: relRows (representing relationships) and nodeRows (representing nodes). Since primary keys, as well as merge constraints, are used, the maintenance of a stringent parent–child sequence becomes unnecessary, as sequential dependencies among the constituent nodes have been eliminated. The only ordering that is left with significance is with respect to the phase level: firstly verifying the constraints, then listing the nodes, finally listing the relations, with both the latter operations being done in parallel.

```
public void upsertTree(TreeNode root, String pov, String patternName,
    boolean replaceChildren) {
    try (Session session = driver.session()) {
        session.executeWrite(tx -> { createConstraints(tx); return null;
    });
    }

    Flattened flat = flatten(root, pov, patternName);

    // Parallel writes (nodes, then relationships)
    parallelBatches(flat.nodeRows, this::writeNodesBatch);
    parallelBatches(flat.relRows,  this::writeRelsBatch);

    // Optional reconciliation
    if (replaceChildren && !flat.keepByParent.isEmpty()) {
        List<DeleteRow> dels = flat.keepByParent.entrySet().stream()
            .map(e -> new DeleteRow(root.patternKey, e.getKey(), new
    ArrayList<>(e.getValue())))
            .toList();
        parallelBatches(dels, this::writeReplaceChildrenBatch);
    }
}
```

Listing 4.16: High-Level Orchestration

Adding parallel batch processing to the tree insertion workflow enables the system to achieve greater scalability and improved throughput when writing large hierarchical data structures to Neo4j.

4.2.4 Frontend

The Pattern Uploader frontend system provides the workflows exposed to the user for producing and dealing with the expected product trees (patterns). As said before, the frontend was developed in Angular, designing the system to also be very testable, extensible, and highly modular. Four key activities are facilitated by the UI, which act as the mirror for the backend functionality: upload an original pattern (reading the file on the client and sending its JSON payload), visualise existing patterns (list of patterns), disable a pattern, and download the JSON stored version of a pattern.

Upload a Pattern

In this subsection, the process by which the frontend interface creates a new pattern is shown. The user will enter the name of the pattern, the parent component number and the location (verification point), then select one JSON file representing the expected product tree. This file is thoroughly read in the browser platform, and the data is sent to the backend with the form input data as an integral JSON payload.

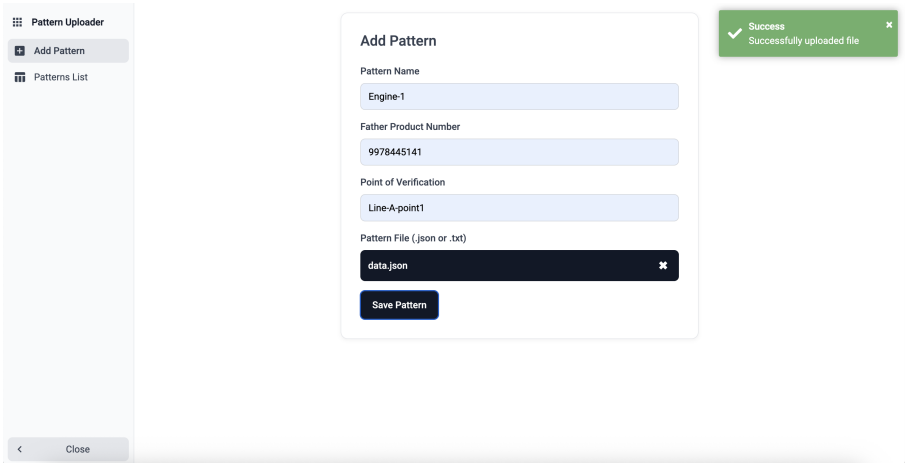

Figure 4.2: Upload a Pattern Page with Data

After a successful operation, a green toast appears to confirm that the pattern was saved in the database. Also, the toast can show errors when the file is not a .json or .txt, as you can see in Figure 4.3.

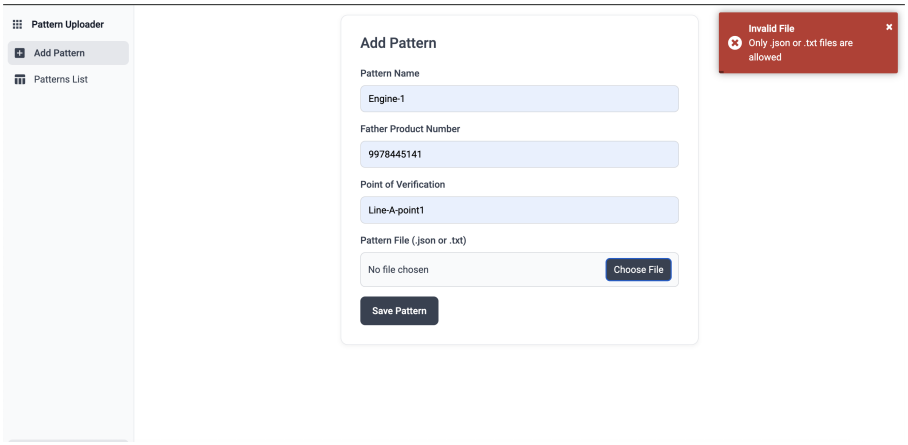

Figure 4.3: File is not valid

The frontend can also show when the content of the file is not a valid JSON structure.

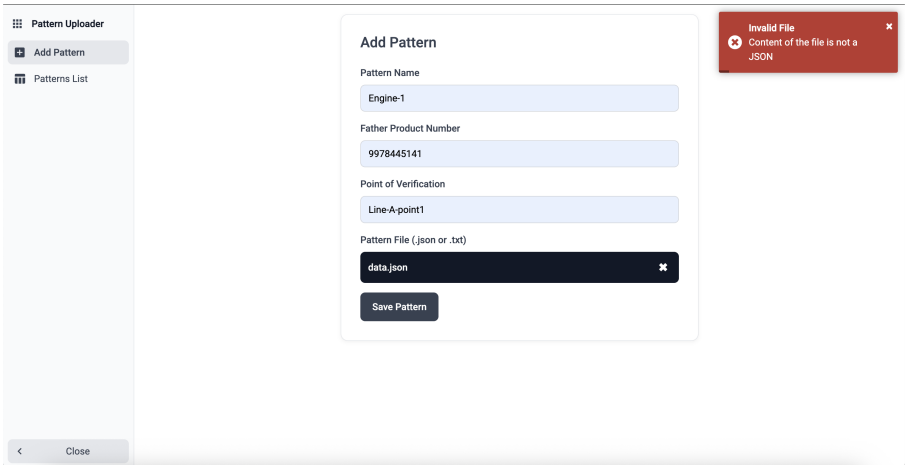

Figure 4.4: Content of the file is not a JSON

List of Patterns

This subsection introduces the table of patterns that facilitates the browsing and searching of current entries. The grid delineates the essential attributes (Pattern Name, Father Product Number, Point of Verification, Creation Date, Status) and incorporates pagination and resizing features to enhance navigational efficiency. Visual indicators of status are utilised to signify whether a pattern is active or inactive, while an Actions column reveals common operations. The Figure 4.5 illustrates the default interface.

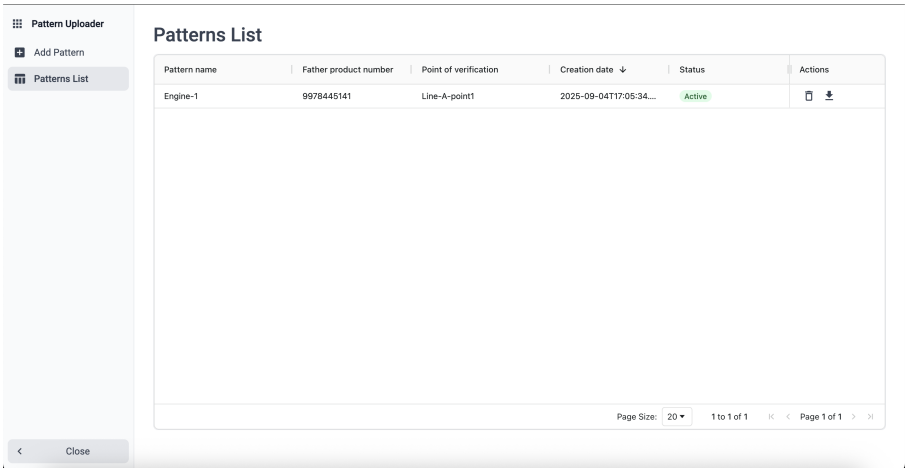

Figure 4.5: Pattern's List

Disable a Pattern

This subsection demonstrates how an inactivation of a pattern occurs from the list view. The user selects the *Set Inactive* (shown as a bin icon) action for the table, which provides a lightweight request for the server to update the status of the pattern. On success, the row immediately updates and the status pill is displayed as "Inactive," with the action disabling itself to avoid duplicate actions. Non-blocking toasts pass both validation errors and success. Figures demonstrate the action button, the confirmation feedback, and the new row state.

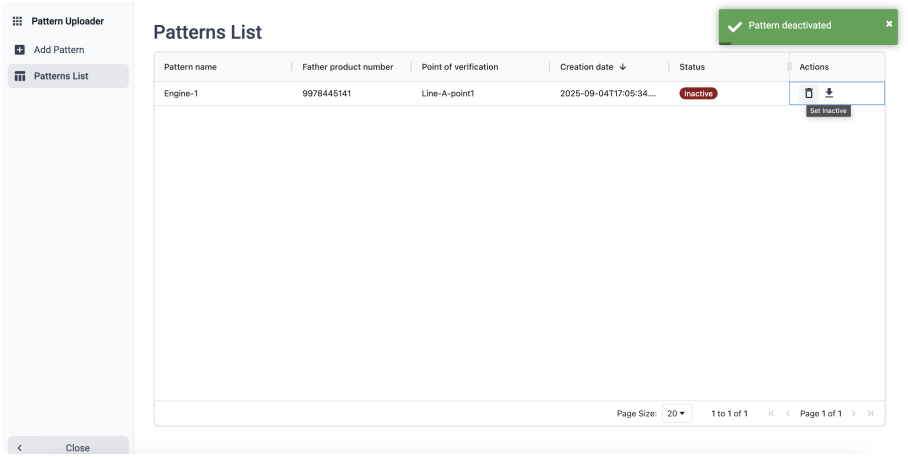

Figure 4.6: Pattern Deactivated

Download a Pattern

This subsection describes the flow by which the frontend accesses the saved JSON for a chosen pattern. The user initiates the download through the Actions column, which requests the backend endpoint, resulting in the JSON as an attachment. The browser then provides the file to save with the name of the *patternKey*, thereby allowing traceability between the list entry and the exported artefact. The Figure 4.7 illustrates the action entry point and resulting download prompt.

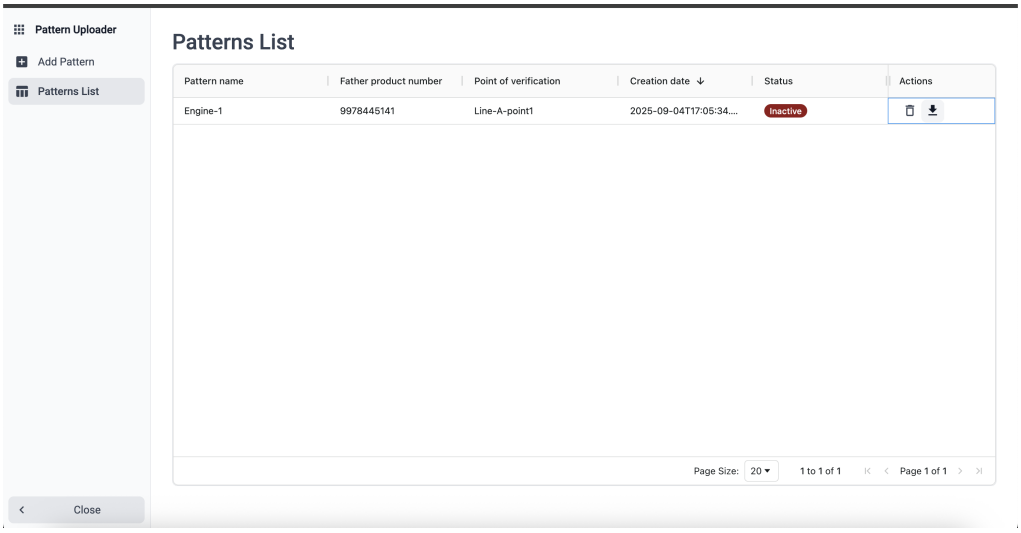

Figure 4.7: Download a pattern

4.3 Testing and Validation

In order to guarantee the scalability, reliability, and accuracy of the solution put forward, it became necessary to adopt a thorough test and validation strategy.

The system consists of various modules, like Kafka-based producers and consumers, the Quality Verification Service, and the repositories supported by Neo4j databases, whose functionality needed to be evaluated on various levels.

As such, the validation effort has been divided into three complementary domains: unit, integration tests, and performance results. These cover different system assurance levels, ranging from the low-level component correctness to the overall business compliance, and therefore together provide an integrated view of the system quality.

4.3.1 Unit Tests

Unit testing was adopted to confirm the correctness and robustness of the system's base logic. Unit tests focus on validating the independent component functionality, such as the repositories, services, and utility functions, and therefore their expected behaviour both under normal and edge-case conditions.

Quality Verification Service: Structural and Property Comparison Tests

Test 1: Identical Trees ⇒ OK

- `ok = true` when trees are equal;
- expected/live node counts match;
- the `problems` list is absent or empty.

Test 2: Missing/Unexpected Elements and Property Mismatches

1. a missing node (`B`) and an unexpected node (`C`);
2. altered parent–child edges (`A`→`B` missing; `A`→`C` unexpected);
3. property mismatches on the root (`uom` and `isAssembly`).

Annex The full Java test code for this section is provided in Annex D.1.

PatternRepository: Parallel Batched Writing Tests

Test 1: Parallelism and Completeness

- measures distinct thread IDs to confirm concurrent execution (>1 threads used, and $\leq$ configured maximum);
- counts produced batches and matches the expected number;
- verifies that all items are processed exactly once (no loss, no duplicates).

Test 2: Clean Error Propagation

- the method rethrows a `RuntimeException` with message "`Batch writing failed`";
- the original cause (`IllegalStateException`) is preserved and accessible via `getCause()`.

Annex The complete test implementation is presented in Annex D.2.

4.3.2 Integration Tests

Integration tests further extend the validation by checking the module-to-module interactions and external dependencies (e.g., the Neo4j and Kafka), which ensure that the data is properly moved across the interfaces.

Quality Verification Consumer: Message Handling and Blocking Logic

Test 1: Non-OK Comparison ⇒ Block Event and ACK

- *acknowledge* the message (no `nack`);
- emits a `BlockEvent` mirroring the comparison outcome (problems, severity, node counts, correlation ID);
- derive the producer key as the suffix after ":" (`"CODE"`) and pass the severity unchanged.

Test 2: Exception Path ⇒ Error Block Event and NACK

- *nack* the message (no `ack`);
- emit a fallback `BlockEvent` with severity `"High"` and `blockedPartCodes` computed as the suffix after "_" (`"ABC"`);
- use that suffix as the producer key.

Annex The complete Java test code for this section is provided in Annex E.1.

4.3.3 Performance Results

Performance analysis was conducted to investigate the system's capacity for handling broad event streams and committing intricate patterns in Neo4j. Experiments explored latency for high-demand cases, centred on the impact of simultaneous writing approaches in the database.

For ensuring consistency and minimising variability, every scenario was repeated 50 times, and the output was taken over these iterations. These explorations enabled the identification of system weaknesses, validation of parallel programming models, and verification of the design's compliance with industry-level scalability needs.

JSON Files Inserts

To verify the offline/backfill path, we consume events from JSON files (NDJSON) directly into Neo4j and capture end-to-end runtime. Unlike the Kafka pipeline, this workload is CPU-bound with disk I/O and JSON parsing dominating other operations, with validation, transformation, and batched MERGE/CREATE operations following. I contrasted a single-threaded reader/writer with the parallel configuration that shards files (or big files) across worker nodes while maintaining per-entity ordering. The table 4.1 summarises total ingest time per input size with read+parse, schema checks, and commit latency included.

Table 4.1: Runtime comparison - sequential vs parallel - JSON Files

Input size	Sequential (ms)	Parallel (ms)
1,000	3-5 sec	150-180 ms
5,000	17-21 sec	200-250 ms
20,000	68-73 sec	300-350 ms
50,000	140-150 sec	550-650 ms

Messaging Kafka

To assess the influence of concurrency within the Kafka - Neo4j framework, I conducted an evaluation comparing a single-threaded writer/consumer with a multithreaded setup that enhances consumer parallelism and increases fetch/poll thresholds while proportionately scaling the Neo4j connection pool. The benchmark repeats 1k–50k events in the same condition, as in Table 4.2.

Table 4.2: Runtime comparison - sequential vs parallel - Messaging Kafka

Input size	Sequential (ms)	Parallel (ms)
1,000	4-6 sec	800-900 ms
5,000	7-10 sec	600-800 ms
20,000	32-36 sec	10-12 sec
50,000	115-121 sec	38-42 sec

Chapter 5

Conclusions

5.1 Objectives Achieved

The primary aim of the current dissertation was the formation of a sequence of objectives aimed at creating a robust, efficient, and scalable solution that can be used to verify product structures in an industrial environment. These objectives were exceeded in many cases, indeed, by meeting them via a systematic combination of design, implementation, and test processes.

To start with, the system design was successfully deployed to build and compare anticipated and actual product trees. It entailed specifying suitable data structures and an algorithm that can detect discrepancies on various levels: absence of or unexpected nodes, relation mismatches, and property mismatches. The service was also proven against descriptive unit tests that guaranteed correctness on clearly defined scenarios, ranging from the same structure to intentionally modified ones.

Second, the offline/backfill ingest path from JSON files was timed under sequential and parallel batching. As tabulated in Table 4.1, sequential runtimes were 3–5 s (1k), 17–21 s (5k), 68–73 s (20k), and 140–150 s (50k), while parallel setting completed in 150–180 ms (1k), 200–250 ms (5k), 300–350 ms (20k), and 550–650 ms (50k). These values confirm substantial speedups with batched writes and concurrent parsing over input sizes.

Third, a stream path was included through Apache Kafka, with producer/consumer logic subjected to normal and stress cases such that errors in data-fetching or comparison emerged as informative blocking events. We see from Table 4.2 that sequential end-to-end runtimes were 4–6 s (1k), 7–10 s (5k), 32–36 s (20k), and 115–121 s (50k), with the parallel consumer/writer taking 0.8–0.9 s, 0.6–0.8 s, 10–12 s, and 38–42 s, respectively. It thereby exhibits steady multi-fold wall-clock time reductions through parallel ingestion.

In conclusion, the complete system underwent thorough validation via extensive testing procedures. Unit tests guaranteed the accuracy of functions, integration tests verified the congruity of components when assembled, and performance assessments evaluated efficiency under various loads. Collectively, these elements offer robust confidence in the reliability and efficacy of the proposed design.

5.2 Limitations, Threats and Future Work

In spite of the achievement of the objectives, the dissertation has some limitations and perceived threats to its validity that deserve mention.

There is a significant limitation that concerns the experimental setup. Testings were conducted against a controlled environment that had access to limited hardware resources and relatively small datasets compared to those normally facing production settings. While the result reflects significant improvement, the scalability may bring more challenges like database contention, network performance fluctuations, or larger memory requirements. Another limitation is the highly simplified nature of the datasets that were used. Trees and messages used during the test were representative but did not essentially reflect the diversity and unpredictability that comes with the data of the industrial kind.

The parallelisation process, which has been successful in providing increased throughput, comes with its own compromises. As the complexity of the workloads increases, the synchronisation, thread scheduling, and resource contention behaviours can deviate considerably. Optimising the batch sizes and thread pools will be very important to bring it to a successful production implementation. Similarly, the Kafka message experiments were restricted to a single-cluster case and didn't account for issues like partition rebalancing, node failure, or continuous resiliency under variable-loading conditions.

These limitations point to several directions of future work. Extending the assessment of performance to cover larger data sets, on the scale of the industry, would provide more meaningful evidence of scalability. Investigating adaptive techniques for repository batching could help better balance latency and throughput. Extending the comparative algorithm to support tolerance ranges, incremental verification, and richer property semantics would enhance its usability. For the purpose of messaging, tests within a distributed system using fault-injection scenarios would shed more light on resilience and reliability by actual operational scenarios.

5.3 Final Assessment

In summary, it has been shown by this dissertation that it is feasible to unify the graph-based product tree verification, parallel data persistence, and concurrent messaging to construct a holistic and highly effective real-time quality control methodology.

The primary goals were thus reached: the system saw implementation, validation, and careful performance measurement. Unit, integration, and performance testing together produced great confidence in correctness as well as efficacy. The results hold particular promise: repository insertions gained speed by more than an order of magnitude, comparisons took place within tens of milliseconds, and message-processing time decreased from the range of minutes to the range of seconds upon the exercise of concurrency.

While there remain limitations and unanswered questions to be addressed by future studies, the value of the research is clear-cut. It allows for a validated proof-of-concept that can serve simultaneously as a foundation of industrial deployment and a model of further academic research. The design choices, supported by experimental evidence, show that quality verification on the fly within complex product architectures is not just achievable but can be made with high efficacy and reliability.

Bibliography

[1] "Quality Management: Practices, Tools, and Standards". In: *Fundamentals of Quality Control and Improvement*. John Wiley & Sons, Ltd, 2008. Chap. 2, pp. 93–146. isbn: 9781118491645. doi: https://doi.org/10.1002/9781118491645.ch3. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118491645.ch3. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118491645.ch3.

[2] Albert Albers et al. "Procedure for Defining the System of Objectives in the Initial Phase of an Industry 4.0 Project Focusing on Intelligent Quality Control Systems". In: *Procedia CIRP* 52 (2016). The Sixth International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2016), pp. 262–267. issn: 2212-8271. doi: 10.1016/j.procir.2016.07.067. url: https://www.sciencedirect.com/science/article/pii/S2212827116308666.

[3] Qingwen Li, Waifan Tang, and Zhaobin Li. "Leveraging Industry 4.0 for Sustainable Manufacturing: A Quantitative Analysis Using FI-RST". In: *Applied Sciences* 14.20 (2024). issn: 2076-3417. doi: 10.3390/app14209545. url: https://www.mdpi.com/2076-3417/14/20/9545.

[4] F.M. Bono, L. Radicioni, and S. Cinquemani. "A novel approach for quality control of automated production lines working under highly inconsistent conditions". In: *Engineering Applications of Artificial Intelligence* 122 (2023), p. 106149. issn: 0952-1976. doi: 10.1016/j.engappai.2023.106149. url: https://www.sciencedirect.com/science/article/pii/S0952197623003330.

[5] The MPI Group. *Digitization Delivers for Manufacturers: Industry 4.0 Increases Productivity, Revenues, and Profitability around the Globe*. Executive Summary. The MPI Group, July 2021. url: https://www.mpi-group.com.

[6] Wei Dai et al. "Online quality inspection of resistance spot welding for automotive production lines". In: *Journal of Manufacturing Systems* 63 (2022), pp. 354–369. issn: 0278-6125. doi: 10.1016/j.jmsy.2022.04.008. url: https://www.sciencedirect.com/science/article/pii/S0278612522000589.

[7] M.A. Montironi et al. "Adaptive autonomous positioning of a robot vision system: Application to quality control on production lines". In: *Robotics and Computer-Integrated Manufacturing* 30.5 (2014), pp. 489–498. issn: 0736-5845. doi: 10.1016/j.rcim.2014.03.004. url: https://www.sciencedirect.com/science/article/pii/S0736584514000209.

[8] CarbonMinus Editorial Team. *Why Most Manufacturing Plants Waste 40% Energy (And How to Fix It)*. en-US. Aug. 2025. url: https://carbonminus.com/why-most-manufacturing-plants-waste-energy-and-how-to-fix-it/ (visited on 08/05/2025).

[9] Claes Wohlin and Per Runeson. "Guiding the selection of research methodology in industry–academia collaboration in software engineering". In: *Information and Software Technology* 140 (2021), p. 106678. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106678. url: https://www.sciencedirect.com/science/article/pii/S0950584921001361.

[10] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004). Publisher: Management Information Systems Research Center, University of Minnesota, pp. 75–105. issn: 02767783. url: http://www.jstor.org/stable/25148625.

[11] Claes Wohlin and Per Runeson. "Guiding the selection of research methodology in industry–academia collaboration in software engineering". In: *Information and Software Technology* 140 (2021), p. 106678. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106678. url: https://www.sciencedirect.com/science/article/pii/S0950584921001361.

[12] Alan R Hevner. "A three cycle view of design science research". In: *Scandinavian journal of information systems* 19.2 (2007), p. 4.

[13] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". en. In: *Journal of Management Information Systems* 24.3 (Dec. 2007), pp. 45–77. issn: 0742-1222, 1557-928X. doi: 10.2753/MIS0742-1222240302. url: https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302.

[14] Roelf J. Wieringa. *Design science methodology for information systems and software engineering*. Undefined. 10.1007/978-3-662-43839-8. Germany: Springer, 2014. isbn: 978-3-662-43838-1. doi: 10.1007/978-3-662-43839-8.

[15] Per Runeson, Emelie Engström, and Margaret-Anne Storey. "The Design Science Paradigm as a Frame for Empirical Software Engineering". In: *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos. Cham: Springer International Publishing, 2020, pp. 127–147. isbn: 978-3-030-32489-6. doi: 10.1007/978-3-030-32489-6_5. url: https://doi.org/10.1007/978-3-030-32489-6_5.

[16] Philipp Offermann et al. "Outline of a design science research process". en. In: *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology - DESRIST '09*. Philadelphia, Pennsylvania: ACM Press, 2009, p. 1. isbn: 978-1-60558-408-9. doi: 10.1145/1555619.1555629. url: http://portal.acm.org/citation.cfm?doid=1555619.1555629.

[17] Matthew J Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021). Publisher: BMJ Publishing Group Ltd _eprint: https://www.bmj.com/content/372/bmj.n71.full.pdf. doi: 10.1136/bmj.n71. url: https://www.bmj.com/content/372/bmj.n71.

[18] Admin. *Short history of manufacturing: from Industry 1.0 to Industry 4.0*. en-US. Jan. 2021. url: https://kfactory.eu/the-industrial-revolution-short-history-of-manufacturing/ (visited on 12/22/2024).

[19] E. Wrigley. "Reconsidering the Industrial Revolution: England and Wales". In: *The Journal of Interdisciplinary History* 49 (June 2018), pp. 9–42. doi: 10.1162/jinh_a_01230.

[20] K. Vinitha et al. "Review on industrial mathematics and materials at Industry 1.0 to Industry 4.0". en. In: *Materials Today: Proceedings* 33 (2020), pp. 3956–3960. issn: 22147853. doi: 10.1016/j.matpr.2020.06.331. url: https://linkinghub.elsevier.com/retrieve/pii/S2214785320348045.

[21] Aeshita Mathur, Ameesha Dabas, and Nikhil Sharma. "Evolution From Industry 1.0 to Industry 5.0". en. In: *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. Greater Noida, India: IEEE, Dec. 2022, pp. 1390–1394. isbn: 978-1-6654-7436-8. doi: 10.1109/ICAC3N56670.2022.10074274. url: https://ieeexplore.ieee.org/document/10074274/.

[22] James Jull. "The second industrial revolution. The history of a concept". In: *Rivista internazionale di storia della storiografia* 36 (1999), pp. 81–90.

[23] Stephanie Muntone. "Second industrial revolution". In: *Education. com. The McGraw-Hill Companies. Retrieved* 14 (2013).

[24] *Industry 4.0: The Future of Manufacturing*. English. url: https://www.sap.com/products/scm/industry-4-0/what-is-industry-4-0.html (visited on 12/22/2024).

[25] *6 Industry 4.0 Challenges and Risks*. en. url: https://www.oracle.com/industrial-manufacturing/industry-4-challenges/ (visited on 12/22/2024).

[26] *What is Kafka? - Apache Kafka Explained - AWS*. en-US. url: https://aws.amazon.com/what-is/apache-kafka/ (visited on 05/05/2025).

[27] *Introduction to RabbitMQ*. en-US. Section: GBlog. url: https://www.geeksforgeeks.org/introduction-to-rabbitmq/ (visited on 03/05/2025).

[28] Nico Braunisch, Sven Schlesinger, and Robert Lehmann. "Adaptive Industrial IoT gateway using kafka streaming platform". In: *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*. 2022, pp. 600–605. doi: 10.1109/INDIN51773.2022.9976153.

[29] *MQTT - The Standard for IoT Messaging*. url: https://mqtt.org/ (visited on 03/05/2025).

[30] *What Is a Bill of Materials (BOM)?* en. url: https://www.investopedia.com/terms/b/bill-of-materials.asp (visited on 12/22/2024).

[31] Jianxin Jiao et al. "Generic Bill-of-Materials-and-Operations for High-Variety Production Management". In: *Concurrent Engineering* 8.4 (2000), pp. 297–321. doi: 10.1177/1063293X0000800404. eprint: https://doi.org/10.1177/1063293X0000800404. url: https://doi.org/10.1177/1063293X0000800404.

[32] Min Liu, Jianbo Lai, and Weiming Shen. "A method for transformation of engineering bill of materials to maintenance bill of materials". In: *Robotics and Computer-Integrated Manufacturing* 30.2 (2014), pp. 142–149. issn: 0736-5845. doi: https://doi.org/10.1016/j.rcim.2013.09.008. url: https://www.sciencedirect.com/science/article/pii/S0736584513000677.

[33] Boming Xia et al. "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 2630–2642. doi: 10.1109/ICSE48619.2023.00219.

[34] Oleg Shilovitsky. *Bill of Materials Transformation (EBOM-MBOM-SBOM)*. en. Sept. 2022. url: https://www.openbom.com/blog/bill-of-materials-transformation-ebom-mbom-sbom (visited on 12/22/2024).

[35] Jiewu Leng et al. "Digital twins-based smart manufacturing system design in Industry 4.0: A review". In: *Journal of Manufacturing Systems* 60 (2021), pp. 119–137. issn: 0278-6125. doi: https://doi.org/10.1016/j.jmsy.2021.05.011. url: https://www.sciencedirect.com/science/article/pii/S0278612521001151.

[36] Markus Hammer et al. "Profit Per Hour as a Target Process Control Parameter for Manufacturing Systems Enabled by Big Data Analytics and Industry 4.0 Infrastructure". In: *Procedia CIRP* 63 (2017). Manufacturing Systems 4.0 – Proceedings of the 50th CIRP Conference on Manufacturing Systems, pp. 715–720. issn: 2212-8271. doi: https://doi.org/10.1016/j.procir.2017.03.094. url: https://www.sciencedirect.com/science/article/pii/S2212827117302408.

[37] *Introduction to Graph Data Structure*. en-US. Section: Graph. url: https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/ (visited on 03/06/2025).

[38] Oleg Shilovitsky. *Graphs, Networks, and BOMs - Part 1*. en. Apr. 2020. url: https://www.openbom.com/blog/graphs-networks-and-boms-part-1 (visited on 12/24/2024).
[39] Oleg Shilovitsky. *OpenBOM: Graphs, Networks, and Bill of Materials - Part 3*. en. Apr. 2020. url: https://www.openbom.com/blog/openbom-graphs-networks-and-bill-of-materials-part-3 (visited on 12/24/2024).
[40] M. Colledani and T. Tolio. "Impact of Quality Control on Production System Performance". In: *CIRP Annals* 55.1 (2006), pp. 453–456. issn: 0007-8506. doi: https://doi.org/10.1016/S0007-8506(07)60457-0. url: https://www.sciencedirect.com/science/article/pii/S0007850607604570.
[41] Douglas Brauer and John Cesarone. *Total Manufacturing Assurance: Controlling Product Quality, Reliability, and Safety*. en. 2nd ed. Boca Raton: CRC Press, Feb. 2022. isbn: 978-1-003-20805-1. doi: 10.1201/9781003208051. url: https://www.taylorfrancis.com/books/9781003208051 (visited on 09/02/2025).
[42] *SPC | Statistical Process Control | Quality-One*. en-US. Dec. 2016. url: https://quality-one.com/spc/ (visited on 12/26/2024).
[43] *What is a relational database?* en. url: https://www.oracle.com/pt/database/what-is-a-relational-database/ (visited on 12/28/2024).
[44] *7 Advantages of Relational Database Management Systems | AppMaster*. en. url: https://appmaster.io/blog/advantages-of-relational-database-management-systems (visited on 12/28/2024).
[45] *What is a relational database? | IBM*. en. Oct. 2021. url: https://www.ibm.com/think/topics/relational-databases (visited on 12/28/2024).
[46] Priya Pedamkar. *Relational Database Advantages*. en-US. Feb. 2020. url: https://www.educba.com/relational-database-advantages/ (visited on 12/28/2024).
[47] *Oracle Database*. en. url: https://www.oracle.com/database/ (visited on 12/29/2024).
[48] *MySQL*. url: https://www.mysql.com/ (visited on 12/29/2024).
[49] *Microsoft SQL Server*. en-US. url: https://www.microsoft.com/en-us/sql-server (visited on 12/29/2024).
[50] PostgreSQL Global Development Group. *PostgreSQL*. en. Dec. 2024. url: https://www.postgresql.org/ (visited on 12/29/2024).
[51] *What Is A Non-Relational Database?* en-us. url: https://www.mongodb.com/resources/basics/databases/non-relational (visited on 12/29/2024).
[52] *Advantages Of NoSQL*. en-us. url: https://www.mongodb.com/resources/basics/databases/nosql-explained/advantages (visited on 12/29/2024).
[53] *MongoDB: a plataforma de dados para desenvolvedores*. pt-br. url: https://www.mongodb.com/ (visited on 12/29/2024).
[54] *Apache Cassandra | Apache Cassandra Documentation*. en. url: https://cassandra.apache.org/ (visited on 12/29/2024).
[55] *Redis - The Real-time Data Platform*. en. url: https://redis.io/ (visited on 12/29/2024).
[56] *Couchbase: Best Free NoSQL Cloud Database Platform*. en-US. url: https://www.couchbase.com/ (visited on 12/29/2024).
[57] *What is a graph database - Getting Started*. en. url: https://neo4j.com/docs/getting-started/graph-database/ (visited on 03/13/2025).
[58] *Managed Graph Database - Amazon Neptune - AWS*. en-US. url: https://aws.amazon.com/neptune/ (visited on 03/16/2025).
[59] *Neo4j Graph Database & Analytics – The Leader in Graph Databases*. en. May 2025. url: https://neo4j.com/ (visited on 03/16/2025).

[60] *Managed Graph Database - Amazon Neptune - AWS*. en-US. url: https://aws.amazon.com/neptune/ (visited on 03/16/2025).

[61] *RedisGraph*. en. url: https://redis.io/docs/latest/operate/oss_and_stack/stack-with-enterprise/deprecated-features/graph/ (visited on 03/16/2025).

[62] Pieter Willem Jordaan. "A novel Decision Support System for modern Relational and Non-Relational Database Systems". PhD thesis. North-West University (South Africa)., 2023.

[63] Thi-Thu-Trang Do et al. "Query-based Performance Comparison of Graph Database and Relational Database". en. In: *The 11th International Symposium on Information and Communication Technology*. Hanoi Vietnam: ACM, Dec. 2022, pp. 375–381. isbn: 978-1-4503-9725-4. doi: 10.1145/3568562.3568648. url: https://dl.acm.org/doi/10.1145/3568562.3568648 (visited on 05/16/2025).

[64] Cornelia A. Győrödi et al. "Implementing a Synchronization Method between a Relational and a Non-Relational Database". In: *Big Data and Cognitive Computing* 7.3 (2023). issn: 2504-2289. doi: 10.3390/bdcc7030153. url: https://www.mdpi.com/2504-2289/7/3/153.

[65] Alperen Sayar et al. "High-Performance Real-Time Data Processing: Managing Data Using Debezium, Postgres, Kafka, and Redis". tr. In: *2023 Innovations in Intelligent Systems and Applications Conference (ASYU)*. Sivas, Turkiye: IEEE, Oct. 2023, pp. 1–4. isbn: 979-8-3503-0659-0. doi: 10.1109/ASYU58738.2023.10296737. url: https://ieeexplore.ieee.org/document/10296737/.

[66] Roman Trobec, Marián Vajteršic, and Peter Zinterhof, eds. *Parallel Computing*. en. London: Springer London, 2009. isbn: 978-1-84882-408-9 978-1-84882-409-6. doi: 10.1007/978-1-84882-409-6. url: http://link.springer.com/10.1007/978-1-84882-409-6 (visited on 09/02/2025).

[67] *What is parallel computing? | IBM*. en. July 2024. url: https://www.ibm.com/think/topics/parallel-computing (visited on 12/31/2024).

[68] *What is Concurrency? | Teradata*. en. Jan. 2022. url: https://www.teradata.com/glossary/what-is-concurrency-concurrent-computing (visited on 12/31/2024).

[69] Simon Marlow. "Parallel and Concurrent Programming in Haskell". In: *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. Ed. by Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 339–401. isbn: 978-3-642-32096-5. doi: 10.1007/978-3-642-32096-5_7. url: https://doi.org/10.1007/978-3-642-32096-5_7.

[70] *Concurrent Programming in .NET Core | DotNetCurry*. url: https://www.dotnetcurry.com/dotnet/1360/concurrent-programming-dotnet-core (visited on 12/31/2024).

[71] COEPD. *What is FURPS+?* en. Aug. 2014. url: https://businessanalysttraininghyderabad.wordpress.com/2014/08/05/what-is-furps/ (visited on 05/31/2025).

[72] *What is Angular?* en. url: https://angular.dev/overview (visited on 07/02/2025).

[73] *What is Quarkus?* en. url: https://quarkus.io/about/ (visited on 07/02/2025).

Appendix A

Appendix A - Work Breakdown Structure (WBS)

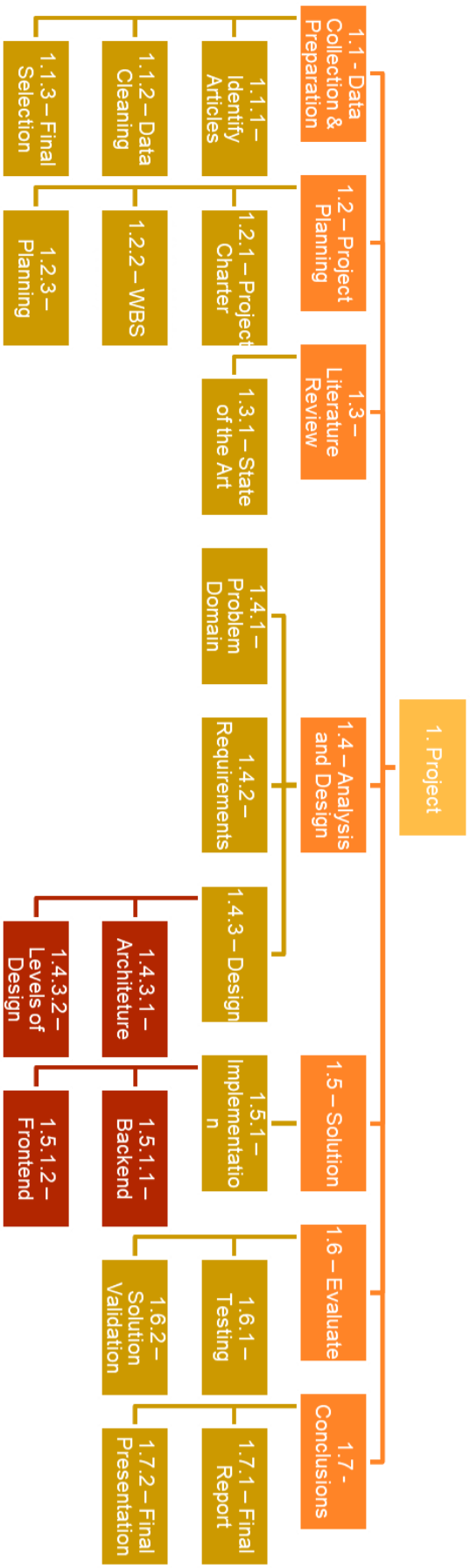

Figure A.1: Work Breakdown Structure

Appendix B

Appendix B - Sprint Detail

Table B.1: Sprint Details

Sprint	Start Date	End Date	Task ID	Task Name and Objective
Sprint 1	November 4, 2024	November 18, 2024	**1.1.1**	Identify Articles: Define search criteria, and conduct initial searches
			1.2.1	Project Charter: Start drafting the Project Charter, including objectives, etc
Sprint 2	November 19, 2024	December 3, 2024	**1.1.3**	Final Selection: Finalize the selection of articles
			1.2.1	Project Charter: Finalize and validate the Project Charter with advisor
			1.2.2	WBS: Begin creating the Work Breakdown Structure (WBS)
Sprint 3	December 4, 2024	December 18, 2024	**1.2.2**	WBS: Complete and validate the WBS
			1.3.1	State of the Art: Continue summarizing findings
Sprint 4	December 19, 2024	January 4, 2025	**1.2.3**	Planning: Finalize the project plan
			1.3.1	State of the Art: Complete the summary of key findings
			1.7.1	Mid-term Report: Prepare, validate, and deliver
			1.7.2	Mid-term Presentation: Prepare, validate, and deliver
Sprint 5	February 24, 2025	March 3, 2025	**N/A**	Adjust schedules and timelines as necessary
Sprint 6	March 3, 2025	March 17, 2025	**1.4.1**	Problem Domain: Identify key challenges
			1.4.2	Requirements: Complete the summary of key findings
			1.4.3	Initial Design: Begin creating architectural diagrams
Sprint 7	March 18, 2025	March 31, 2025	**1.4.3**	Detailed Design: Refine architecture and define system components
			1.4.3	Detailed Design: Validate the design with advisor
Sprint 8	April 1, 2025	April 14, 2025	**1.5.1.1**	Backend: Develop initial modules and establish the database structure
Sprint 9	April 15, 2025	April 28, 2025	**1.5.1.1**	Backend: Complete core backend functionality
			1.5.1.2	Frontend: Begin creating user interfaces
Sprint 10	April 29, 2025	May 12, 2025	**1.5.1.1**	Backend: Finalize all modules
			1.5.1.2	Frontend: Continue developing components and connect with backend
Sprint 11	May 13, 2025	May 26, 2025	**1.5.1.2**	Frontend: Finalize the developments
			1.6.1	Testing: Conduct functional testing of integrated components
			1.6.2	Solution Validation: Validate if the solution meets the initial requirements
Sprint 12	May 27, 2025	June 9, 2025	**1.6.2**	Solution Validation: Perform final testing and address remaining issues
			1.5	Solution: Make any final adjustments
Sprint 13	June 10, 2025	June 23, 2025	**1.7.1**	Final Report: Draft and finalize the project report
			1.7.2	Final Presentation: Develop and refine the presentation materials

Appendix C

Appendix C - Project Schedule - Gantt Chart

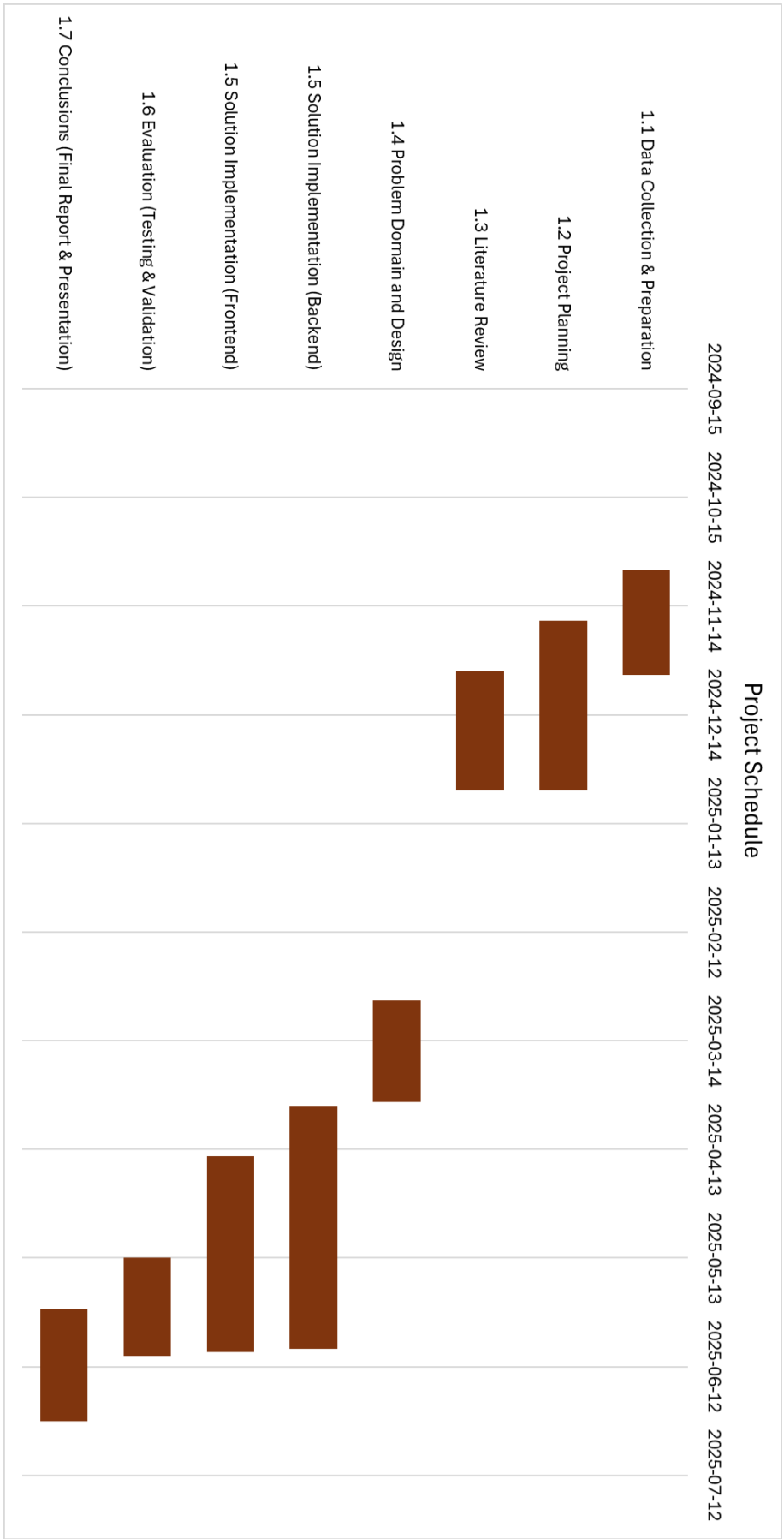

Figure C.1: Project Schedule (Gantt Chart)

Appendix D

Appendix D - Unit Testing Java Code

D.1 Quality Verification Service Tests

```
1     @Test
2     void compare_identicalTrees_returnsOk() {
3         TreeDTO.Node b = node("B", Map.of("uom", "kg", "isAssembly",
    false, "qty", 2));
4         TreeDTO.Node c = node("C", Map.of("uom", "kg", "isAssembly",
    false, "qty", 1));
5         TreeDTO.Node a = node("A", Map.of("uom", "kg", "isAssembly",
    true, "qty", 1), b, c);
6
7         TreeDTO expected = tree(a);
8         TreeDTO live = tree(
9                 node("A", Map.of("uom", "kg", "isAssembly", true, "qty",
     1),
10                        node("B", Map.of("uom", "kg", "isAssembly",
    false, "qty", 2)),
11                        node("C", Map.of("uom", "kg", "isAssembly",
    false, "qty", 1))
12                )
13        );
14
15        CompareResultDTO r = svc.compare("PK-1", expected, live);
16
17        assertTrue(r.ok, "eEqual Tree should return OK");
18        assertEquals(3, r.expectedNodeCount);
19        assertEquals(3, r.liveNodeCount);
20        if (r.problems != null) {
21            assertTrue(r.problems.isEmpty(), "It shouldn't have a
    problem");
22        }
23    }
```

```
    @Test
    void compare_detectsMissingUnexpectedNodesEdgesAndPropMismatches() {
        TreeDTO expected = tree(
                node("A", Map.of("uom", "kg", "isAssembly", true, "qty", 1),
                        node("B", Map.of("uom", "kg", "isAssembly", false, "qty", 2))
                )
        );

        TreeDTO live = tree(
                node("A", Map.of("uom", "g", "isAssembly", false, "qty", 1),
                        node("C", Map.of("uom", "kg", "isAssembly", false, "qty", 2))
                )
        );

        CompareResultDTO r = svc.compare("PK-2", expected, live);

        assertFalse(r.ok, "Should fail with differences");
        assertEquals(2, r.expectedNodeCount);
        assertEquals(2, r.liveNodeCount);

        var problems = r.problems;
        assertNotNull(problems);
        assertTrue(problems.stream().anyMatch(s -> s.equals("Missing in live: B")));
        assertTrue(problems.stream().anyMatch(s -> s.equals("Unexpected in live: C")));
        assertTrue(problems.stream().anyMatch(s -> s.equals("Missing child in live: A -> B")));
        assertTrue(problems.stream().anyMatch(s -> s.equals("Unexpected child in live: A -> C")));
        assertTrue(problems.stream().anyMatch(s -> s.startsWith("UOM mismatch on A:")));
        assertTrue(problems.stream().anyMatch(s -> s.startsWith("isAssembly mismatch on A:")));

        assertEquals("MEDIUM", r.severity, "With ~6 problems, the severity is MEDIUM");
    }
```

D.2 Pattern Repository Tests

```
    @Test
     void parallelBatches_processesAllInParallel() throws Exception {
         PatternRepository repo = new PatternRepository();

         int total = 100;
         List<Integer> items = new ArrayList<>(total);
         for (int i = 0; i < total; i++) items.add(i);
         var processed = new ConcurrentLinkedQueue<Integer>();
         var threadsUsed = new ConcurrentSkipListSet<Long>();
         var batchCount = new AtomicInteger();

         int expectedBatches = (int) Math.ceil((double) total / (Math.max
    (1, repo.batchSize)));
         var ready = new java.util.concurrent.CountDownLatch(Math.min(2,
    expectedBatches));
         var go = new java.util.concurrent.CountDownLatch(1);

         new Thread(() -> {
             try {
                 ready.await();
                 try { Thread.sleep(20); } catch (InterruptedException
    ignored) { Thread.currentThread().interrupt(); }
             } catch (InterruptedException ignored) {
                 Thread.currentThread().interrupt();
             } finally {
                 go.countDown();
             }
         }).start();

         Consumer<List<Integer>> writer = batch -> {
             if (ready.getCount() > 0) ready.countDown();
             try {
                 go.await();
             } catch (InterruptedException e) {
                 Thread.currentThread().interrupt();
                 throw new RuntimeException(e);
             }
             batchCount.incrementAndGet();
             threadsUsed.add(Thread.currentThread().getId());
             try { Thread.sleep(25); } catch (InterruptedException
    ignored) { Thread.currentThread().interrupt(); }
             processed.addAll(batch);
         };

         var m = PatternRepository.class.getDeclaredMethod("
    parallelBatches", List.class, Consumer.class);
         m.setAccessible(true);
         m.invoke(repo, items, writer);

         assertEquals(total, processed.size());
         assertEquals(total, processed.stream().distinct().count());
         assertEquals(expectedBatches, batchCount.get());
         assertTrue(threadsUsed.size() > 1);
         assertTrue(threadsUsed.size() <= 4,);
     }
```

```
1       @Test
2      void parallelBatches_propagatesErrorsCleanly() throws Exception {
3          PatternRepository repo = new PatternRepository();
4
5          List<Integer> items = List.of(1,2,3,4);
6
7          Consumer<List<Integer>> writer = batch -> {
8              if (batch.contains(3)) throw new IllegalStateException("boom
       ");
9          };
10
11         Method m = PatternRepository.class.getDeclaredMethod("
       parallelBatches", List.class, Consumer.class);
12         m.setAccessible(true);
13
14         RuntimeException thrown = assertThrows(RuntimeException.class,
       () -> {
15             try {
16                 m.invoke(repo, items, writer);
17             } catch (java.lang.reflect.InvocationTargetException ite) {
18                 if (ite.getTargetException() instanceof RuntimeException
        re) throw re;
19                 throw new RuntimeException(ite.getTargetException());
20             }
21         });
22
23         assertEquals("Batch writing failed", thrown.getMessage(), "
       Unexpected Error Message");
24         assertTrue(thrown.getCause() instanceof IllegalStateException, "
       Unexpected cause");
25         assertEquals("boom", thrown.getCause().getMessage());
26     }
```

Appendix E

Appendix E - Integration Testing Java Code

E.1 Quality Verification Service Tests

```
    @Test
    void onStart_whenCompareHasProblems_sendsBlockEventAndAck() throws Exception {
        String json = "{\"patternKey\":\"PREF:CODE\",\"correlationId\":\"corr-123\"}";
        StartCommand cmd = new StartCommand();
        cmd.patternKey = "PREF:CODE";
        cmd.correlationId = "corr-123";

        when(mapper.readValue(eq(json), eq(StartCommand.class))).thenReturn(cmd);
        when(expectedTreeService.fetch("PREF:CODE")).thenReturn(new TreeDTO());
        when(liveTreeService.fetch("PREF:CODE")).thenReturn(new TreeDTO());

        CompareResultDTO result = new CompareResultDTO();
        result.ok = false;
        result.patternKey = "PREF:CODE";
        result.problems = List.of("Something failed");
        result.severity = "MEDIUM";
        result.expectedNodeCount = 3;
        result.liveNodeCount = 2;

        when(qualityVerificationService.compare(eq("PREF:CODE"), any(), any())).thenReturn(result);

        Message<String> msg = mockMessage(json);
        CompletionStage<Void> stage = consumer.onStart(msg);
        stage.toCompletableFuture().join();

        verify(msg, times(1)).ack();
        verify(msg, never()).nack(any());

        BlockEvent ev = eventCap.getValue();
        assertNotNull(ev);
        assertEquals("PREF:CODE", ev.patternKey);
        assertEquals(List.of("Something failed"), ev.problems);
        assertEquals("MEDIUM", ev.severity);
        assertNotNull(ev.blockedAt);
```

```
        assertEquals("corr-123", ev.correlationId);
        assertEquals("CODE", keyCap.getValue());
        assertEquals("MEDIUM", sevCap.getValue());
    }
```

```
    @Test
    void onStart_whenExceptionOccurs_sendsErrorBlockEventAndNack()
    throws Exception {
        String json = "{\"patternKey\":\"PK_ABC\",\"correlationId\":\"c
    -777\"}";
        StartCommand cmd = new StartCommand();
        cmd.patternKey = "PK_ABC";
        cmd.correlationId = "c-777";

        when(mapper.readValue(eq(json), eq(StartCommand.class))).
    thenReturn(cmd);
        when(expectedTreeService.fetch("PK_ABC")).thenThrow(new
    RuntimeException("boom fetching expected"));

        Message<String> msg = mockMessage(json);
        CompletionStage<Void> stage = consumer.onStart(msg);
        stage.toCompletableFuture().join();

        verify(msg, times(1)).nack(any());
        verify(msg, never()).ack();

        ArgumentCaptor<BlockEvent> eventCap = ArgumentCaptor.forClass(
    BlockEvent.class);
        ArgumentCaptor<String> keyCap = ArgumentCaptor.forClass(String.
    class);
        ArgumentCaptor<String> sevCap = ArgumentCaptor.forClass(String.
    class);

        verify(blockProducer, times(1)).send(eventCap.capture(), keyCap.
    capture(), sevCap.capture());

        BlockEvent ev = eventCap.getValue();
        assertEquals("PK_ABC", ev.patternKey);
        assertEquals(List.of("ABC"), ev.blockedPartCodes);
        assertEquals("High", ev.severity);
        assertNotNull(ev.blockedAt);
        assertEquals("ABC", keyCap.getValue());
        assertEquals("High", sevCap.getValue());
    }
```